FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

Mitigation of Advanced Code Reuse Attacks

Paul Ioan Muntean

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK
LEHRSTUHL FÜR IT SICHERHEIT

Mitigation of Advanced Code Reuse Attacks

Paul Ioan Muntean

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Jens Grossklags, Ph.D.
Prüfer der Dissertation:	
	1. Prof. Dr. Claudia Eckert
	2. Prof. Dr. Zhiqiang Lin

Die Dissertation wurde am 04.11.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 09.03.2021 angenommen.

Acknowledgements

I would like to take the opportunity to thank, give gratitude and appreciation to the main persons, who supported me during my Ph.D. studies.

First, of all I would like to thank Prof. Dr. Claudia Eckert, for giving me the opportunity of pursuing a Ph.D. and for helping me during the years whenever I needed her help. Without her support, this dissertation would not have seen the light. Further, I would like to thank Prof. Zhiqiang Lin (Ph.D.) who decided to support this work by being my second thesis supervisor.

Second, I would like to give thanks to my main research collaborators. Prof. Jens Grossklags (Ph.D.), Technical University of Munich (TUM), Germany; Prof. Zhiqiang Lin (Ph.D.), The Ohio State University, USA; Prof. Gang (Gary) Tan (Ph.D.), Penn State University, USA; and Prof. Dr. Martin Monperrus, KTH Royal Institute of Technology, Sweden. Their support helped me bring my research to the next level.

Third, I would like to thank my current and former colleagues with whom I had the opportunity to work over the years and to share interesting research ideas. These are first of all the persons at the chair for IT security, from TUM and the Fraunhofer Institute for Applied and Integrated Security (AISEC). In no particular order, I would like to address my thanks to the following persons: Julian Kirsch, Peng Xu, Dr. Bojan Kolosnjaji, Dr. Huang Xiao, and Dr. Julian Schütte (Google, DE). Further, I would like to thank all students from TUM with whom I collaborated over the last years during my research projects.

Fourth, I want to also thank other researchers who helped mainly with constructive discussions. Dr. Benjamin Johnson, TUM, Germany; Dr. Dimitar Bounov, University of California, San Diego, USA; Dr. Thurston Dang, Massachusetts Institute of Technology (MIT), USA; Dr. Istvan Haller, Microsoft Research, UK; Dr. Felix Schuster, Microsoft Research, UK; Dr. Jake Weidmann, Google, USA; Dr. Hao Sun, Alibaba, P. R. China; Dr. Xi Cheng, Tsinghua University, P. R. China; Artur Janc, Google, CH; Jaroslav Sevcik, Google, DE; Haohuang Wen, The Ohio State Univ., USA; and Chandler Carruth, Google, USA; and Thomas Dullien (Halvar Flake), Google, CH. All these people sacrificed countless hours to help improve my work.

Finally, my special appreciation and gratitude goes to my family and to my parents, who always believed in me and continuously encouraged me to follow my dreams. Thanks to my brother for his positive thoughts. Without their love, support, and encouragement I could not have reached this milestone in my life.

Abstract

The people around the world use more and more software in their daily lives, as such the boundary between digital and human is about to vanish. A natural observation is that the interest of many type of actors (*e.g.,* nation state actors, state sponsored actors, *etc.*) rises exponentially as this software success story unfolds. As digital data is the new oil of the economy, these actors are driven by many incentives to get to this data. As such, comparable efforts are undertaken to reach the point in which private devices are no longer private but rather controlled and owned by others, in many cases without the user knowledge. This is achieved by performing an attack which uses a certain system weakness such as a program code-based vulnerability. As such, it is important to have a deep understanding of code reuse attacks (CRAs) as these are often used by these actors to reach their above-mentioned goals. Thus, this thesis provides tools and techniques to mitigate these problems by offering approaches to address CRAs along two main lines of research based on static and dynamic code analysis.

In the first part of this thesis, we present a static symbolic execution based framework INTDETECT, which can detect integer overflows in `C` source code programs, as integer overflows often lead to memory corruptions, and even to CRAS. INTDETECT can reliably detect integer overflows and does not suffer from false negatives for the tested programs. We integrate it in the Eclipse IDE which is a well-established and widely used Integrated Development Environment (IDE). Next, we extend the static `C` source code analysis framework on which INTDETECT relies by implementing an integer overflow detection and repair generation tool, called INTREPAIR, on top of it. INTREPAIR generates `C` source code repairs that help a programmer to automatically repair a previously detected integer overflow. INTREPAIR can efficiently remove a fault, automatically validate a repair and does not introduce unwanted program behavior. Further, we extend our static source code analysis framework in order to not only detect and repair integer overflows, but also to detect and repair buffer overflows which are in most cases one of the main prerequisites for performing CRAs. For this purpose, we provide a tool, called BUFFREPAIR, which automatically generates buffer overflow repairs, does not suffer from false negatives, and can also validate a repair.

In the second part of this thesis, we focus on the detection of dynamic memory corruptions, which most commonly lead to (or are a prerequisite for) CRAs. We are motivated to take this path due to the intrinsic limitations of static analysis techniques used in the first part of this thesis. More precisely, we develop a compiler-based sanitizer tool, called CASTSAN, which detects object type confusions during runtime and which is completely integrated into a well-established compiler framework (*i.e.,* Clang/LLVM). CASTSAN is a fully functional object type confusion detection tool that is based on a novel and efficient technique for detection of only polymorphic `C++` object type confusions. Thus, if consistently used it can considerably reduce the likelihood of CRAs.

Next, we design a static compiler based tool, named LLVM-CFI, which can assess state-of-the-art static CFI defenses. The intuition behind this decision is twofold: First, due to the fact that currently memory corruptions cannot fully be eradicated from programs, we would like to provide a runtime defense to harden a program. Second, we would like to design and

implement a novel CFI-based technique, which will be introduced in the next parts of this thesis, for protecting indirect program control flow transfers. In order to effectively address this task, we first need to learn how effective the existing state-of-the-art CFI defenses are and which level of security they offer. Thus, we develop LLVM-CFI, a novel framework for assessing static CFI policies w.r.t. calltarget set reduction after a certain CFI defense was applied. Further, by using LLVM-CFI, we gain important knowledge which helps us to prepare the next design decisions for the tools presented later in this thesis. Further, we design and implement a compiler-based tool, called ρFEM, which is based on the results from the second part of this thesis. ρFEM protects program CFG backward-edges stemming from indirect and direct forward-edge program control flow transfers. ρFEM is based on a novel technique for protecting program CFG backward-edges relying on a fine-grained CFI policy, which provides an optimal set of return targets for each protected callee. In this way, the likelihood of successfully performing CRAs exploiting backward edges is greatly reduced. At the same time, a solution serving as a competitive alternative for shadow stacks is provided. In contrast to shadow stack techniques, except Intel Control Flow Enforcement (CET) which is based on hardware support or Return Address Defender (RAD) which uses page permissions, ρFEM does not rely on entropy and information hiding. Thus, the corresponding protection disclosing attack vectors which are relevant for shadow-stack techniques do not apply for ρFEM.

Finally, in the last part of this thesis, we develop a framework called τCFI for protecting legacy program binaries with novel CFI policies designed by taking into account the lessons we learned and summarized in previous chapters. These lessons have helped us to design and implement a tool which can effectively protect forward and backward program CFG edges in stripped program binaries as this type of information is usually not required in production-ready binaries. Note that most of the semantic information has vanished through the compilation process. In this way, CRAs which rely on corrupting forward and/or backward CFG edges due to indirect control flow transfer violations are mitigated by greatly reducing the likelihood of successfully performing such an attack when hardening the program binary with τCFI.

Zusammenfassung

Menschen auf der ganzen Welt nutzen zunehmend mehr Software in ihrem alltäglichen Leben, was dazu führt, dass die Grenze zwischen der digitalen und realen Welt zunehmend verschwindet. Während Software stetig populärer wird, steigt auch exponentiell das Interesse von verschiedenen Akteuren (*z.B.* staatliche Angreifer, staatlich gesponserte Angreifer, *etc.*). Digitale Daten werden heutzutage oft als das neue Gold betitelt. Aus diesem Grund unternehmen Angreifer zum Teil große Anstrengungen, um private Geräte ohne das Wissen der Benutzer zu kapern, um an wertvolle Daten zu gelangen. Deshalb ist es wichtig, ein tiefgreifendes Verstandnis von Code-Wiederverwendungs-Angriffen (CWA) zu haben. Denn diese stellen einen Hauptangriff Vektor dar, der von diesen Akteuren verwendet wird, um ihre Ziele zu erreichen. Diese Dissertation bietet daher Lösungen und Ansätze zur Minderung von Problemen dieser Art, indem CWAs mit zwei verschiedenen Forschungsansätze adressiert werden.

Im ersten Teil dieser Dissertation präsentieren wir ein statisches symbolisches Ausführungsframework, in welchem wir zuerst ein Ganzzahl Überlauferkennungswerkzeug namens INTDETECT entwerfen und implementieren. Anschließend wird dieses in der Eclipse Entwicklungsumgebung integriert. INTDETECT kann Ganzzahlüberläufe in `C` Quellcode-Programmen erkennen, die oftmals zu einer Speicher-Korruption oder sogar zu einer CWA führen können. Des Weiteren kann INTDETECT Ganzzahl-Fehler zuverlässig erkennen ohne dabei Falschmeldungen zu melden. Wir integrieren dieses Werkzeug in die etablierte und weit verbreitete Eclipse Entwicklungsumgebung. Als nächstes erweitern wir unser statisches `C` Quellcode-Analyse-Framework, das wir bereits oben erwähnt haben, durch die Integration eines Ganzzahl-Überlauferkennungs- und Reparatur-Werkzeugs namens INTREPAIR. INTREPAIR kann Quellcode-Reparaturen automatisch durchführen, die einem Programmierer helfen den Ganzzahl-Überlauf zu korrigieren. INTREPAIR kann einen Fehler effizient entfernen, überprüft automatisch generierte Reparaturvorschläge und führt kein unerwünschtes Programmverhalten ein. Außerdem erweitern wir unser statisches `C` Quellcode-Analyse-Framework, sodass dieses Pufferüberläufe, welche in vielen Situationen eine Hauptvoraussetzung fur CWAs sind, erkennen und reparieren kann. Hierfür stellen wir ein Werkzeug namens BUFFREPAIR zur Verfügung. BUFFREPAIR erzeugt automatisch Pufferüberlauf Reparaturen, und kann einen vorher generierten Quellcode-Reparatur-Vorschlag validieren.

Im zweiten Teil dieser Dissertation präsentieren wir einen dynamischen Ansatz zur Erkennung von Speicher Verfälschungen, die oft zu CWAs führen können oder sogar eine Voraussetzung für diese sind. Unsere Motivation hierfür ergibt sich aus den Einschränkungen der statischen Analysetechniken wie im ersten Teil dieser Dissertation herausgearbeitet. Wir entwickeln ein Compiler basiertes Erkennungswerkzeug namens CASTSAN, das während der Programmlaufzeit polymorphe `C++` Objekttyp-Konflikte erkennt. CASTSAN wird anschließend in einem etablierten Compiler-Framework (*d.h.* Clang/LLVM) integriert. CASTSAN ist ein voll funktionsfähiges Objekttyp-Verwirrung Detektion-Werkzeug, das auf einer neuen und effizienten Technik zu Erkennung von polymorphen Objekttyp-Verwirrungen basiert. Bei konsequenter Verwendung von CASTSAN kann die Wahrscheinlichkeit von CWAs erheblich reduziert werden.

Als Nächstes entwickeln wir ein statisches Compiler-basiertes Werkzeug namens LLVM-CFI, das statische Kontrollfluss Integritäts (CFI) Regeln bewerten und untereinander vergleichen kann. Die Wahl dieses Ansatzes ist zweifach motiviert. Erstens Speicherkorruptionen in Programmen können nicht vollständig verhindert werden, deshalb um Programme zu härten wollen wir eine Laufzeitverteidigung bereitstellen. Zweitens wollen wir eine neuartige CFI basierte Technik entwerfen und implementieren, die in den nächsten Teilen dieser Dissertation vorgestellt wird. Diese neue CFI Technik soll Schutz gegen nicht erlaubten indirekten Programmsteuerung-Fluss-Übertragungen bieten. Um diese Aufgabe effektiv zu bewältigen, finden wir zunächst heraus, wie effektiv die bestehenden CFI-Abwehrmechanismen sind und welches Sicherheitsniveau diese bieten. Hieraus motiviert entwickeln wir LLVM-CFI, ein neuartiges Framework zur Bewertung statischer CFI-Schutzregeln. Durch die Verwendung von LLVM-CFI erhalten wir wichtige Erkenntnisse, die uns bei der Vorbereitung der nächsten Werkzeug-Entwurfsentscheidungen helfen sollen. Außerdem entwickeln und implementieren wir ein Compiler-basiertes Werkzeug namens ρFEM, das auf den Erkenntnissen basiert, die wir im zweiten Teil dieser Dissertation gewonnen haben. ρFEM schützt Programm-Kontrolfluss-Graphen (PKG)-Rückwärtskanten, die von indirekten und direkten Vorwärts-Kanten-Programmsteuerungsfluss-Übertragungen stammen. ρFEM basiert auf einer neuartigen Technik zum Schutz von Programm- PKG-Rückwärtskanten die auf präzisen CFI-Regeln basieren. Die CFI Technik ermöglicht für jede geschützte und aufgerufene Programm-Funktion eine optimale Anzahl von Rückkehrzielen. So wird die Wahrscheinlichkeit der erfolgreichen Durchführung von CWAs, die auf Rückwärts-Transferkanten basieren, stark reduziert. Gleichzeitig bieten wir eine Lösung, die eine wettbewerbsfähige Alternative für Shadow-Stack-basierte Ansätze darstellt. Im Gegensatz zu Shadow-Stack-Ansätzen, die nicht auf Intel Control Flow Enforcement (CET) und Return Address Defender (RAD) basieren, ist ρFEM nicht auf die Adressraum-Entropie und das Verstecken von Information angewiesen. Somit sind die entsprechenden Angriffsvektoren nicht relevant.

Des Weiteren entwickeln wir im letzten Teil dieser Dissertation ein Framework namens τCFI, um Legacy-Programme, in denen *z.B.* der Quellcode nicht verfügbar ist, zu schützen. τCFI basiert auf den neuen CFI-Regeln die wir aus den Erkenntnissen des zweiten Teils dieser Dissertation entwickeln. Diese helfen uns, ein Software-Werkzeug zu entwerfen und zu implementieren, das Vorwärts- und Rückwärts-Programm-Kontrollfluss-Kanten im Programm-Binärdateien wirksam schützen kann. Auf diese Weise werden CWAs, die auf der Korruption von rückwärtigen Kontrollfluss-Kanten basieren stark eingeschränkt, was die Wahrscheinlichkeit von erfolgreichen Angriffen deutlich reduziert, wenn τCFI zum Einsatz kommt.

Contents

List of Figures

List of Tables

List of Acronyms

- ABI - Application Binary Interface
- ACICS - Argument Corruptible Indirect Call Site
- API - Application Programming Interface
- ASLR - Address Space Layout Randomization
- AST - Abstract Syntax Tree
- AT - Address Taken
- AUFNIRA - Arrays, Uninterpreted Functions, Non-linear Integer and Real Arithmetic
- CCDF - Complementary Cumulative Distribution Function
- CDF - Cumulative Distribution Function
- CFB - Control Flow Bending
- CFG - Control Flow Graph
- CFI - Control Flow Integrity
- CFV - Control Flow Variant
- COOP - Counterfeit Object Oriented Programming
- COP - Call Oriented Programming
- CRA - Code Reuse Attack
- CWE - Common Weakness Enumeration
- CoFI - Change of Flow Instruction
- DEP - Data Execution Prevention
- DFS - Depth First Search

- DoS - Denial of Service
- EC - Equivalence Classes
- ESC - Extended Static Checking
- FP - False Positive
- GCC - GNU Compiler Collection
- HW - Hardware
- ID - Identifier
- IDE - Integrated Development Environment
- IL - Intermediate Language
- IR - Intermediate Representation
- ISA - Instruction Set Architecture
- IVT - Interleaved Virtual Table
- JIT - Just in Time
- JOP - Jump Oriented Programming
- JS - JavaScript
- LBR - Last Branch Register
- LLVM - Low Level Virtual Machine
- LOC - Lines of Code
- LTO - Link Time Optimization
- MCFI - Modular Control Flow Integrity
- OS - Operating System
- PC - Personal Computer
- PT - Processor Trace
- ROP - Return Oriented Programming
- RQ - Research Question

- RTTI - Runtime Type Information
- SD - Standard Deviation
- SEH - Structured Exception Handler
- SLoC - Source Lines of Code
- SMT - Satisfiability Modulo Theories
- SQL - Structured Query Language

List of Publications

TOPS'21 **Paul Muntean**, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. Analyzing CFI Defenses: The Bad, The Good & The Potential. In *in submission*, ACM, Apr. 2021.

NDSS'20 **Paul Muntean**, Richard Viehoever, and Claudia Eckert. iTOP: Indirect Transfer Oriented Programming: Automating Control-Flow Hijacking Attacks In *in submission*, IEEE, Oct. 2020.

ACSAC'20 **Paul Muntean**, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. ρFEM: Efficient Callee Protection Using Reversed Caller Mappings. In *Annual Computer Security Applications Conference (ACSAC)*, ACM, Dec. 2020.

ACSAC'19 **Paul Muntean**, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. Analyzing Control Flow Integrity with LLVM-CFI. In *Annual Computer Security Applications Conference (ACSAC)*, ACM, Dec. 2019.

TSE'19 **Paul Muntean**, Martin Monperrus, Hao Sun, Jens Grossklags, and Claudia Eckert. IntRepair: Informed Repairing of Integer Overflows. In *Transactions on Software Engineering (TSE)*, IEEE, Aug. 2019.

Usenix Sec.'19 Felix Fischer, Huang Xiao, Ching-yu Kao, Yannick Stachelscheid, Benjamin Johnson, Danial Razar, Paul Furley, Nat Buckley, Konstantin Böttinger, **Paul Muntean** and Jens Grossklags. Stack Overflow Considered Helpful! Deep Learning Security Nudges Towards Stronger Cryptography, In *Proceedings of the USENIX Security Symposium (USENIX Security)*, Santa Clara, CA, USA, Aug. 2019.

ESORICS'18 **Paul Muntean**, Sebastian Wuerl, Jens Grossklags, and Claudia Eckert. CastSan: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Barcelona, Spain, LNCS, Sept. 2018.

RAID'18 **Paul Muntean**, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. τCFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*, Heraklion, Greece, LNCS, Sept. 2018.

Arxiv'18	**Paul Muntean**. Automated CFI Policy Assessment with Reckon. In Arxiv CoRR abs/1812.08496, 2018.
Arxiv'17	**Paul Muntean**, Jens Grossklags, and Claudia Eckert. Practical Integer Overflow Prevention. In ArXiv CoRR abs/1710.03720, 2017.
Tech. Rep.'16	**Paul Muntean**, and Alexander Malkis. Information Exposure Checker. *Technical report, TUM-I1646, Technical University of Munich*, ˙`https://mediatum.ub.tum.de/doc/1326283/1326283.pdf`, 2016.
Tech. Rep.'16	**Paul Muntean**, and Alexander Malkis. Automatic Security Checks on the Model Level. *Technical report, TUM-I1657, Technical University of Munich*, ˙`https://mediatum.ub.tum.de/doc/1341307/1341307.pdf`, 2016.
Arxiv'16	**Paul Muntean**. Mobile Robot Navigation on Partially Known Maps using a Fast A Star Algorithm Version. In *Arxiv CoRR abs/1604.08708*, 2016.
SAFECOMP'15	**Paul Muntean**, Vasantha Kommanapalli, Andreas Ibing, and Claudia Eckert. Automated Generation of Buffer Overflow Quick Fixes Using Symbolic Execution and SMT. In *Proceedings of the International Conference Computer Safety, Reliability, and Security (SAFECOMP)*, LNCS, Sept. 2015.
QRS'15	**Paul Muntean**, Adnan Rabbi, Andreas Ibing, and Claudia Eckert: Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code. In *Proceedings of the Quality, Reliability and Security Conference (QRS) Companion*, Vancouver, Canada, IEEE Aug. 2015.
ISSA'15	**Paul Muntean**, Mustafizur Rahman, Andreas Ibing, and Claudia Eckert: SMT-constrained symbolic execution engine for integer overflow detection in C code. In *Proceedings of the Information Security for South Africa (ISSA)*, Johannesburg, South Africa, IEEE, Aug. 2015.
InnoSWDev'14	**Paul Muntean**, Claudia Eckert, and Andreas Ibing. Context-sensitive detection of information exposure bugs with symbolic execution. In *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices, (InnoSWDev)*, co-located with ACM's FSE conference, Hong Kong, China, ACM, Nov. 16, 2014.
Arxiv'14	**Paul Muntean**. Modeling of information-flow restrictions. *Technical report TUM-I1416, Technical University of Munich*, ˙`https://mediatum.ub.tum.de/doc/1244626/1244626.pdf`, 2014.

Chapter 1

Introduction

What is the overall view? Protecting software programs against control-flow hijacking attacks (*e.g.,* ROP, DOP, COOP) is a basic requirement needed in order to have a secure system since these types of attacks are more and more prevalent. This situation is confirmed by the increasing number of CRAs reported by Mitre's CWE [177]. Nowadays users use many programs from multiple vendors for different purposes. Some of these vendors take security seriously. But all are driven by cost and time to market criteria.

A CRA represents an exploit which is performed by reusing existing code from a vulnerable application or other system components such as shared libraries. Typical attack vectors are represented by memory safety issues due to: buffer overflows, integer overflows, NULL pointer dereferences, use after free, use of uninitialized memory, double frees, object type confusions, see Payer [220], for more details. In this thesis we differentiate between control flow graph (CFG) violating attacks and non CFG violating attacks (*e.g.,* data-only-attacks). Note that, in most cases, the attacker needs to trigger undefined behavior in order to perform such an attack. Thus, this thesis focuses on providing tools for protecting user-space applications against control-flow hijacking attacks which are based on a memory corruptions. Further for data-only CRAs, we advise the reader to consult, for example, Ispoglou *et al.* [121].

However, there are multiple ways to protect programs against control-flow hijacking attacks. Important to note is that such an attack represents a user capability which can be performed: (1) completely manually, (2) partially manually, or (3) fully automatically (if a tool support is available). The user usually triggers undefined behavior to build his attacks. As such, in most cases these types of attacks are manually built and thus represent a user capability since these highly depend on the skills and knowledge of the attacker in order to craft such an attack.

Modern operating systems, compilers and binary hardening tools provide many techniques for protecting software against control-flow hijacking attacks. Two main OS based approaches (*e.g.,* DEP [164] and ASLR [219] are used to protect programs. Data execution prevention (DEP) forbids jumping into and executing data, and address space layout randomization (ASLR) helps to randomize the code and data sections inside a program during loading. Further, these two approaches do not solve the problem of protecting against these attacks as these were shown to be easily by-passable by attackers. DEP can be bypassed by changing the bit of the page table which indicates if that particular page is executable or not. ASLR is bypassable through information leaks which help the attacker to learn about the layout of the program which he wants to attack. Thus, their security primitives are weak in the face of modern and advanced attacks.

Further, as these OS based defenses were shown to be of questionable utility researchers are continuously proposing new OS based defenses which use hardware support and comprise of stronger primitives which should lower the likelihood of a successful attack. As a result, the OS based applications try to focus on imposing stricter runtime policies. Also, live monitoring of applications is another line of research which relies on heuristics and a previously computed CFG in order to detect malicious program execution streams but with less success as recent research has demonstrated, see Schuster *et al.* [239]. Finally, there is no clear separation between techniques which can be used only for the OS kernel space and, as such, these are applied to user space as well.

What is the problem? User space based protection. As the problem of protecting against control-flow hijacking attacks is a multi-layer problem, one has to rely on multiple components in order to tackle this problem. By multi-layer we mean, in this context, that the user space protection against control-flow violating attacks can be addressed in multiple ways. The main approaches are as follows: (1) preventing memory corruptions through runtime checking and software testing, (2) hardening binaries against control-flow violations when source code is unavailable, (3) providing hard to bypass fine-grained runtime primitives, and (4) hardware based monitoring of malicious program execution streams. Note that this list is not exhaustive and other attack mitigation lines of defense exist having their advantages and disadvantages. One of the most notable approaches is (4) live monitoring of execution streams which is based on the interplay between user space and kernel space, found recently application in the Windows OS 10 (*i.e.,* CFGuard [167]).

What are the existing solutions? Live monitoring of program execution approaches, such as kBouncer [216], ROPecker [44], and ROPGuard [83] validate the execution of a program by monitoring branch transfers based on keeping track of so called *from* and *to* address pair and in certain time intervals these jumps are compared against a precomputed program CFG. While some simple CRAs can be detected by using these primitives this type of approach can easily be fooled and is of questionable usage when more advanced CRAs are performed, see Schuster *et al.* [239].

What are the limitations of the solution? Unfortunately, the heuristics on which these types of systems are based can only keep track of limited amount of from-to address pairs in hardware which was not specifically designed for this purpose but rather for program debugging. Further, this type of defenses assume that the previously computed CFG is sufficient for assessing if an execution trace is malicious or not.

This assumption does not hold since these CFG are not as precise as these should be in order to catch malicious behavior. Further, it is well known that alias analysis of pointers is undecidable as demonstrated by Ramalingam [229]. As such, computing a precise CFG which would allow for precise analysis is unfeasible. Further, advanced CRAs (*e.g.,* COOP) exploit exactly pointer based indirect transfers for which construction a CFG is very difficult, and in practice rather program primitives are used to reconstruct metadata, which can be used to check these indirect transfers such as the program class hierarchy.

Further, recent CRAs explicitly target this main intrinsic limitation which makes the program monitoring approaches by filtering benign from malicious program executions very difficult. As such, we can see many lines of research to address this problem.

No one shot solution. In general, the main problem is that the machine instruction set architectures (ISA) are designed without security in mind but rather are mostly optimized for low runtime overhead. Further, the main research in the last decades went into runtime overhead reduction and thus we are confronted nowadays with intrinsic ISA limitations which could potentially be eradicated by coming up with secure by design ISAs. Until then we address CRAs from multiple dimensions and pursue the cat and mouse game which have not seen its peeks yet.

What is our insight? In this thesis, we focus on how to reduce the likelihood of a successful CRA in cases: (1) source code is under development, (2) is fully available, or (3) the source code is no longer available (*e.g.,* legacy applications). As such, we first look at how can tools help programmers spot bugs early on in the development cycle and how to reliable repair them. For this purpose, in a first step, we come up with tools which can help a programmer to find and repair software bugs which can lead to CRAs in an automated fashion. Equipped with this knowledge, we propose runtime based tools which can use program data that is only available during runtime in order to detect other types of memory corruptions that can lead to CRAs. As a consequence of the intrinsic limitations of these types of tools we propose new control flow integrity (CFI) policies which can be enforced during program compile time and to legacy binaries. With the knowledge gained, we answer the question on how the likelihood of successful exploitation can be reduced and build a framework which can be used to assess the protection level of different static state-of-the-art CFI defenses during compile time.

Improving user space based protection. As techniques for protecting software against attacks become more and more prevalent, attackers also adapt and provide new methods to perform CRAs. A relatively new approach is based on advanced attack (*e.g.,* COOP [238]), which uses only available instructions and do not violate the caller-callee function calling convention. To accomplish this, advanced attacks define a new set of gadgets and rely on hard to determine (before runtime of) forward-edge transfer violations. A gadget is a set of machine code instructions which usually terminate in an indirect transfer instruction (*e.g.,* `ret`, `jump`, `call`, *etc.*). This set of gadgets allow powerful attacks which despite what the literature proposes are not Turing complete (see Dullien *et al.* [76] for more details) and can infect a system, for example, with malware. Despite this major threat posed by this type of attacks there are only a few countermeasures which can combat advanced attacks [59, 268] at the source code and binary level in user space.

Thus, in this thesis, we see the mitigation of advanced attacks as a multi-layer problem which we believe it needs to be tackled with a corresponding multi-step approach. As mentioned before, reasoning about alias analysis is undecidable, we focus on recuperating program metadata during compile time (virtual tale hierarchy) and from program binary (function parameter types) to: (1) use the program virtual table hierarchy for checking of runtime object type confusions (CastSan), (2) use the program virtual table hierarchy for imposing return target sets for callees (ρFEM), (3) assess different static state-of-the-art CFI defenses (LLVM-CFI), and (4) impose function signature for constraining forward and backward edge transfers (τCFI). In order to recuperate the whole program virtual hierarchy we use the compiler to collect all relevant virtual table metadata and transport it to make it available during program link time. Further, in order to build function signatures for forward and backward edge dispatches we use the x86-64 bit function calling convention and the width of the used registers. Next, we perform a liveness and reaching analysis in order to find matching pairs.

Further, we employ static symbolic execution and build a fully integrated framework into an IDE for detection and repair of integer overflow and buffer overflows. Our framework constructs for each analyzed program a CFG and then for a previously extracted path the path constraints are translated to satisfiability modulo theories (SMT) constraints and solved with the help of a SMT solver. In this way integer overflows and buffer overflows can be detected and automated repairs are suggested for the user.

Finally, we provide a multi-step approach of tackling advanced CRAs by helping to remove memory corruptions which are likely to lead to CRAs during program development or during runtime and provide tools for hardening the source code and binary code against advanced CRAs as well a framework for assessing state of the art static CFI defenses.

Kernel space based protection. Even though we did not provide kernel space mitigation against advanced CRAs, we note that our techniques developed for protecting user space application can also be adapted to protect the kernel space. Note that similar approaches as ours have been adapted with success in the past to protect kernel space applications as well.

1.1 Research Questions

This thesis presents several approaches to combating advanced CRAs in the user space. This consists in a three steps approach to address this problem, namely: (1) detecting and repairing memory corruptions which can lead to advanced attacks as well as runtime detection of memory corruptions which can be exploited in order to perform attacks, (2) hardening of program's source code and machine code with novel fine-grained CFI policies, and (3) assessing static CFI defenses with the goal to determine which is more effective w.r.t. protecting against these types of attacks. To address and mitigate this problems, we address several research questions (**RQ**s) and formulate them as following.

- **Static source code analysis for automated memory corruption localization and repair generation.**
  - **RQ1:** How to efficiently locate integer overflows which potentially lead to CRAs in `C` source code based programs without false negatives and potential false positives? Within **RQ1**, we need to research on how to design a novel technique which can be used to effectively and reliably detect integer overflows in `C` source code without

false positives. Previous research did not address the problem of static integer overflow detection with the help of static symbolic execution from a programmer usable tool which can be integrated in a ready to use, well integrated and widely used IDE.

- **RQ2:** How to automatically generate and validate integer overflows repairs in `C` source code based programs which potentially lead to CRAs? After we researched in **RQ1** how to effectively detect integer overflows now within **RQ2**, we need to devise a new methodology to automatically generate integer overflow repairs which are validated automatically, can remove the previously detected fault, and does not introduce unwanted program behavior. Previous research did not address the integer overflow repair generation from the angles presented above, and thus we think that in this way we can provide more useful repairs.
- **RQ3:** How to automatically generate and validate buffer overflows, which can potentially lead to CRAs with the help of `C` source code based program repairs? After presenting a technique for generating integer overflows repairs **RQ2** the goal of **RQ3** is to come up with a new technique which can be used to detect and repair buffer overflows in `C` source code programs without false negatives but potentially false positives and which can be easily integrated in the previous mentioned **RQ1** framework. Previous research did not address repair correctness, and generation of buffer overflow repairs at *non-in-place* locations, and thus we think that by focusing on these repair characteristics we can improve the state-of-the-art.

• **Dynamic source and machine code based analysis and instrumentation for runtime memory corruption detection, CRAs mitigation, and static CFI policies assessment.**

- **RQ4:** How to more effectively detect object type confusions in source code programs which can potentially be used to perform CRAs? The goal of **RQ4** is to devise a new technique for detecting runtime based object type confusions which can be used to perform CRAs. Previous research has looked at runtime library based support for detection of object type confusions. In this research we focus on a purely static technique with no need of object type tracking during runtime, and thus we improve w.r.t. runtime efficiency the state-of-the-art.
- **RQ5:** How to assess state-of-the-art static CFI defenses and provide the set of legitimate calltargets for each callsite which are still accessible even after a static state-of-the-art CFI policy was applied? Next, in order to design an effective CFI-based technique for protecting program CFG backward-edges against CRA based exploits in **RQ6** and **RQ7**, we focus within **RQ5** to come up with a framework which can be used to assess such kind of static CFI based defenses that are widely used and for which there is currently no reliable way to assess them w.r.t. how effective this can protect against certain types of CRAs. Previous work w.r.t. providing a tool for automated static CFI policy assessment is not publicly available at the time of writing this thesis, as such we would like to provide the first tool of this kind in order to better assess static CFI policies against each other.

- **RQ6:** How to protect CFG backward edges from program control flow bending which can be used for performing CRAs in the presence of a deployed defense? In **RQ6** we focus on securing program control flow graph edges during runtime under the assumption that an exploitable memory corruption exists and which was not previously detected and removed from the vulnerable program. Previous work has mostly relied on the presence of a shadow stack technique for protecting backward edges. Since these techniques can be bypassed we want in this research to provide a competitive alternative which does not have the limitations of shadow stacks and thus improve state-of-the-art w.r.t. protection of runtime backward edge dispatches.
- **RQ7:** How to protect CFG forward and backward edges in program binaries which are usually corrupted during a CRAs? In **RQ7** we focus on protecting CFG forward and backward in binary programs based on the lessons learned in **RQ5**. Further, we assume that source code is not available (*i.e.,* legacy code, closed source libraries, *etc.*). Previous work has assumed that shadow stack techniques are efficient w.r.t. backward edge protection and function signatures can be used to protect forward edges, and thus in this research we want to improve state-of-the-art by proposing a novel technique for protecting forward edges. This technique is more precise then similar state of the art solutions. Yet another another technique will be introduced for protecting backward edges, this technique does, for example, not suffer from the intrinsic limitations of shadow stack techniques.

1.2 Contributions

What are our contributions? In this thesis, we provide answers to the above mentioned research questions. In the following, we briefly list the contributions made while researching for the answers to the previously mentioned research questions. Note that the following contributions are listed for completeness reasons at the beginning of the appropriate chapters too.

- **Static source code analysis for automated memory corruption localization and repair generation.**
  - We design a novel source code repair generation technique for integer overflows in `C` programs.
  - We implement INTREPAIR, a prototype of our novel integer overflow repairing technique, for `C` source code programs. INTREPAIR can automatically repair integer overflows across multiple integer precisions.
  - We show the effectiveness of INTREPAIR's w.r.t. code repairing and that the repairs induce a low runtime overhead by running INTREPAIR on 2,052 `C` programs contained in the currently largest open-source test suite for `C/C++` source code (NSA's Juliet) and with 50 synthesized programs which range up to 20 KLOC.

- We demonstrate that IntRepair is superior to manually generated repairs and also more time-effective than repairing the same programs manually by using IntRepair within a user-based controlled experiment.
- We provide precise symbolic modeling of `C` related semantics needed for integer overflow detection.
- We design and implement an integer overflow checker, called IntDetect, as an Eclipse IDE plug-in based on our static execution engine, and automated testing based on automatically generated jUnit test cases and Eclipse projects.
- We present an experimental evaluation of IntDetect on the currently largest open source `C/C++` test case CWE_190_Integer_Overflow contained in the Juliet test suite.
- We provide an algorithm for generation of *in-place* and *non-in-place* buffer overflow quick fixes.
- We present a novel approach, called BuffRepair, for automated buffer overflow fault repair generation based on program input saturation.
- We implement a semi-automated patch insertion based tool for visualizing source code files based on differential views (old code and patched source code).
- We implement within BuffRepair an approach for automated checking of program behavior preserving after a patch was applied to a source code based program.

• **Dynamic source and machine code based analysis and instrumentation for runtime memory corruption detection, CRAs mitigation and static CFI policies assessment.**
  - We design a novel fine-grained backward-edge protection technique without relying on information hiding.
  - We implement our technique based on the Clang/LLVM compiler framework inside a prototype called ρFEM.
  - We show that ρFEM has a low runtime overhead of 2.72% in geomean for the Google's Chrome Web browser and of less than 1% in geomean for the SPEC CPU2017 benchmarks.
  - We develop a novel technique for detecting `C++` object type confusions during runtime, which is based on the linear projection of virtual table hierarchies.
  - We implement our technique in a prototype, called CastSan, which is based on the Clang/LLVM compiler framework and the Gold plug-in.
  - We demonstrate that CastSan is more efficient than other state-of-the-art tools by running CastSan on the same benchmarks.
  - We show that static CFI attacker models are powerful and drastically lower the bar for performing CRAs against state-of-the-art defenses.

- We implement LLVM-CFI, a novel framework usable for generating low-effort CRAs and for empirically assessing CFI defenses against each other, using constraint-driven static analysis of aggregated program meta-data obtained during program compilation.
- We compare existing static state-of-the-art CFI defenses against each other, and show their strengths and weaknesses by employing LLVM-CFI's CFI defense constraints. We compute the legitimate target sets and demonstrate that these allow further reasoning about possible attacks.
- We present a NodeJS-based case study with the goal of highlighting how LLVM-CFI can be used to craft CRAs against a state-of-the-art defense.
- We introduce four new CFI defense assessing metrics that are more effective and precise than existing metrics. We also show the significance of one of our metrics by putting it to work in our evaluation.
- We present τCFI, a new CFI system that improves the state-of-the-art CFI with more precise forward-edge identification by using type information reverse-engineered from stripped x86-64 binaries.
- We implement τCFI as a binary instrumentation framework to enforce a fine-grained forward-edge and backward-edge protection.
- We show that τCFI is more precise and effective than other state-of-the-art techniques.

1.3 Thesis Outline

The rest of this thesis is organised as follows:

In Chapter 2, we present the required background knowledge needed in order to understand the rest of this thesis. More specifically, we introduce background information about CRAs and how this can be mitigated. Further, we introduce the control flow integrity technique for protecting indirect control flow transfers. We present information about object type confusions and needed data in order to understand how program backward edges can be protected. Also, we introduce knowledge about integer overflows and buffer overflows and present general information about these topics. Finally, we give an overview about forward and backward CFG edge transfers in program binaries and present advantages and disadvantages of shadow stack based techniques.

In Chapter 3, we discuss related work concerning the detection of object type confusions and protection of program backward edges using compiler based instrumentation. Then we discuss about automated CRA gadget discovery and existing metrics used to assess the efficiency of CRA protection techniques. Further, we present related work about integer overflow and buffer overflow detection and automated repair generation as well as how advanced CRAs can be mitigated.

In Chapter 4, we present IntDetect, a tool usable for detection of integer overflows in `C` source code based programs across multiple integer precisions.

In Chapter 5, we present IntRepair which can detect and generate automated repairs which can completely remove the previously detected integer overflow.

In Chapter 6, we present, BuffRepair, a tool usable for automated detection and repair generation of buffer overflows in `C` source code based programs.

In Chapter 7, we present CastSan, a compiler based tool used for detection of object type confusions during runtime. This tool can detect object type confusions which potentially can lead to CRAs.

In Chapter 8, we present LLVM-CFI, a compiler based tool which can be used to model and assess state-of-the-art static CFI defenses and provides the possibility to pinpoint legitimate calltagerts still available after a CFI policy was applied and which through control flow bending for example can still be reached and used within a CRA.

In Chapter 9, we present ρFEM, a compiler based tool which can be used to protect backward edges by inferring a minimal set of legitimate callee return targets.

In Chapter 10, we present τCFI, a tool used for program binary analysis and instrumentation with the goal to protect forward and backward edges of the program CFG which are the result of program indirect control flow transfers.

Finally, in Chapter 11, we conclude this thesis, present potential future work avenues, and at the end of this Chapter we provide final remarks.

Chapter 2

Background

In this Chapter, we present required knowledge in order to better understand the rest of this thesis. More precisely, in Section 2.1, we present background knowledge about code reuse attacks (CRAs), and in Section 2.2, we introduce the concept of integer overflows detection, while in Section 2.3, we highlight some essential information about integer overflow repairing. In Section 2.4, we introduce information on how buffer overflows are repaired in general, and in Section 2.5, we present important concepts about object type confusions, while in Section 2.6, we introduce basic knowledge about CFI and how these protection techniques can be best assessed. In Section 2.7, we give an overview of program CFG backward-edges, and in Section 2.8, we describe notions about type information recovery from program binaries. Finally, note that parts of this chapter have already been published by Muntean *et al.* [195, 196, 197, 198, 199, 200, 201].

2.1 Code Reuse Attacks

In this Section, we present the difference between code reuse attacks and advanced code reuse attacks and introduce some mitigation techniques.

2.1.1 Simple Code Reuse Attacks

Code-reuse attacks (CRAs) are of two main types: (1) program control-flow violating attacks (*i.e.,* ROP, DOP, COOP, JOP, *etc.*) and (2) program control flow not violating attacks (*i.e.,* data-only-attacks, see Ispoglou *et al.* [121]). While historically (1) were encountered before (2) both types of attacks use machine code instructions in order to craft malicious behavior. These machine code instructions are called gadgets and by chaining them together in a so called gadget chain an attack can be performed. These gadgets are used to perform computations which are not Turing complete (see Dullien [76] for more details) despite the fact that the current existing research community propagates the knowledge as being Turing complete. Note that neither *real* Turing completeness nor the shorthand for *write and compute anywhere in the address space* is required for exploitation.

Further, a gadget chain can have from a few gadgets (*i.e.,* 2-3) to hundreds or even thousands of gadgets. The number of gadgets depends on the attack success definition (*i.e.,* consider the attack success determined by the successful execution of a gadget chain containing *N* gadgets). Note that each existing attack presented in the academic community defines its own set of gadgets. Further, each gadget set is essentially w.r.t. the number of used instructions a superset or a subset of another attack gadget set. Most commonly, these attacks use program indirection due to the fact that program indirection is hard to be statically determined and difficult to enforce dynamically. Yet another reason is that alias analysis in binary programs is undecidable, see Ramaligam [229] for more details.

In this thesis, we focus on (1) program control flow violating attacks (*i.e.,* non-data-only attacks). Non-data-only attacks are violating the control flow of the program and stem most of the time from a stack or a heap-based memory corruption (*i.e.,* integer or buffer overflow, dangling pointer, type confusion, *etc.*). These types of attacks are used to mount disastrous attacks which remain stealth most of the time to program-monitoring tools (*i.e.,* anti virus tools). This is due to the fact that the monitoring tools cannot differentiate between benign or malicious program execution which is based on the attacker's intended logic which is performed through chaining of gadgets together.

2.1.2 Advanced Code Reuse Attacks

These (1) attacks can be further classified into non-advanced code reuse attacks (*i.e.,* historically before advanced CRAs) and advanced code reuse attacks. These following characterizations are based on what these attacks particularly violate and how these attacks can be effectively mitigated.

Non-advanced code reuse attacks can be detected by enforcing the caller/callee calling convention while advanced code reuse attacks do not violate this calling convention (see COOP attack [238]). Also, some non-advanced code reuse attacks can be mitigated my using coarse-grained CFI-based policies while advanced code reuse attacks can only be mitigated by using fine-grained CFI-based policies. Some of the non-advanced code reuse attacks could be mitigated by ASLR and/or DEP while advanced code reuse attacks bypass these protection mechanisms. Both types of attacks can be mitigated by employing a CFG approximation-based CFI policy while the `C++` program-based attacks can also be mitigated by employing object-oriented program concepts based on CFI-based policies. All non-advanced code reuse attacks and some advanced code reuse attacks (*i.e.,* in the COOP attack the stack-base pointer (BP) moves up and down) which corrupt the stack and can be mitigated by monitoring the stack behavior during runtime and comparing it with the normal (expected) program behavior, particularly when parameters are passed to functions indirectly.

More precisely, advanced CRAs are more difficult to defend against than CRAs and have appeared from a timeline perspective after CRAs. The exact time when advanced CRAs were introduced is not clear so for this reason, we used a simple characterization in this thesis in order to separate them.

2.1.3 Code Reuse Attacks Prerequisites

In order to perform a code reuse attack, usually a memory corruption is needed (*i.e.,* buffer overflow, object type confusion, integer overflow, *etc.*). Note that there are CRAs which rely on a memory corruption which is not necessarily caused by a bug.

Next, the attacker needs a binary layout leakage in order to know where the data and the code sections are located in the vulnerable program. Further, he may want to know where the system Libc is in order to be able to call exploitable functions. The attacker may also want to deactivate Data Execution Prevention (DEP) and/or Address Space Layout Randomization (ASLR). Next, the attacker needs to put his payload on the system where the vulnerable application runs. Further, the attacker will exploit the memory corruption and run the program/logic contained in the payload. This way, the attacker calls code reuse gadgets one by one in order to perform his malicious Turing-complete computations such as installing malware or preparing the stage for an advanced persistent threat APT.

2.1.4 Mitigation of Code Reuse Attacks

Code Reuse Attacks such as ROP and its manifestations (*i.e.,* RILC, JIT-ROP, COP, COOP, JOP, Stiching Numbers [189]) can be mitigated with binary rewriting (user space application and OS kernel), source code recompilation or runtime monitoring approaches such as (1) fine-grained CFI with hardware support PathArmor [266], (2) by using coarse-grained CFI such as CCFIR [289], (3) coarse-grained CFI based on binary loader CFCI [291], (4) fine-grained code randomization STIR [275] and O-CFI [187], (5) cryptography with hardware support-based CCFI [160], (6) based on the ROP stack pivoting, PBlocker [227], (7) canary based as Dyna-Guard [223], (8) checking vTable integrity for protecting against COOP based on CFI for source code auch as SafeDispatch [124], VTV [261] LLVM and GCC compiler based vor vTable protection and binary rewriting such as vfGuard [226], vTint [287] and [59] (9) with runtime hardware support-based on a combination of LBR, PMU and BTS registers CFIGuard [284], and (10) with code recompilation (e.g., reassemble disassembly [269], superset disassembly [17], or probabilistic disassembly [170]) with CFI and/or randomization enforcement against JIT-ROP, MCFI [210], RockJIT [211] and PiCFI [212] and any combination of the above.

2.2 Integer Overflows

In this Section, we present required background knowledge in order to better understand the rest of this thesis.

2.2.1 Integer Overflows

Integer overflow is a known cause of memory corruption and a widely known type of vulnerability [272]. It often leads to stack or heap overflow and thus is usually exploited indirectly (as opposed to buffer overflows which are exploited directly). More specifically, integer overflows occur at runtime, when the result of an integer expression exceeds the maximum allowed value (*e.g.*, $2^{32} - 1$).

```
1 int64_t data = 0LL;
2 //Potential flaw: use random value
3 data = (int64_t)RAND64();
4  if(data > 0){
5  /* Potential flaw: if (data*2)>
6  LLONG_MAX, this will overflow */
7  int64_t result = data * 2;
8  printLongLongLine(result);}
```

Figure 2.1: Integer overflow shaded gray at line seven.

Figure 2.1 depicts an integer overflow memory corruption at line number 7, which could manifest because there is no proper check in place for verifying the range of admissible values for `data`. This integer overflow (and potential underflow) error can be avoided by checking the value of `data` to see if it is less than or equal to $\frac{LLONG\_MAX\_VAL}{2} = 4.611.686.018.427.387.903$ and greater than or equal to $\frac{-LLONG\_MAX\_VAL}{2}$.

2.2.1.1 Characteristics of Integer Overflows

Integer overflows can be classified as malicious or benign. Essentially, an integer overflow manifests itself when the program receives a user-supplied input and subsequently the input value is used in an arithmetic operation to trigger an integer overflow. Thus, a smaller-than-expected value is supplied to the memory allocation function and as a result a smaller-than-expected memory will be allocated. Note that a smaller-than-expected value can be supplied to the memory allocation function and this might not always be an issue as it is possible that the allocation is properly sized, but a subsequent integer overflow results in a buffer underflow. Deciding between the types of integer overflow related problems is rather difficult and a lot of research has been devoted in the last years to this type of classification [255]. The general desire in the research community is to categorize different hard-to-find integer overflows w.r.t. how exploitable these are in contrast to just finding and repairing them.

2.2.1.2 Integer Overflow Related Problems

There are several integer overflow related problems that we will next list and briefly describe. CWE-191, integer underflow (wrap or wrap-around) [175], is the result of multiplying two values with each other and the result is less than the minimum admissible integer value due to the fact that the product subtracts one value from another. CWE-192: integer coercion error [174], manifests during bad type casting and the extension or truncation of primitive data types. CWE-193: off-by-one error [179], manifests during product calculation/usage; an incorrect maximum/minimum value is used which is one more, or one less than the correct value. CWE-194: unexpected sign extension [181], appears when an operation performed on a number can cause it to be sign-extended when it is transformed into a larger data type. CWE-195: signed to unsigned conversion error [180], manifests when a signed primitive that is used inside a cast to an unsigned primitive can produce an unexpected result if the value of the signed primitive cannot be represented using an unsigned primitive. CWE-196: unsigned to signed conversion error [182], manifests when an unsigned is used inside a cast to a signed primitive, which can produce an unexpected value if the result of the unsigned primitive cannot be represented using a signed primitive. CWE-19:, numeric truncation error [178], manifests when a primitive is casted to a primitive of a smaller size and data is lost in the conversion. CWE-680: integer overflow to buffer overflow [176], appears when an integer overflow occurs that causes less memory to be allocated than expected, which can lead to a buffer overflow.

2.2.2 Detecting Integer Overflows

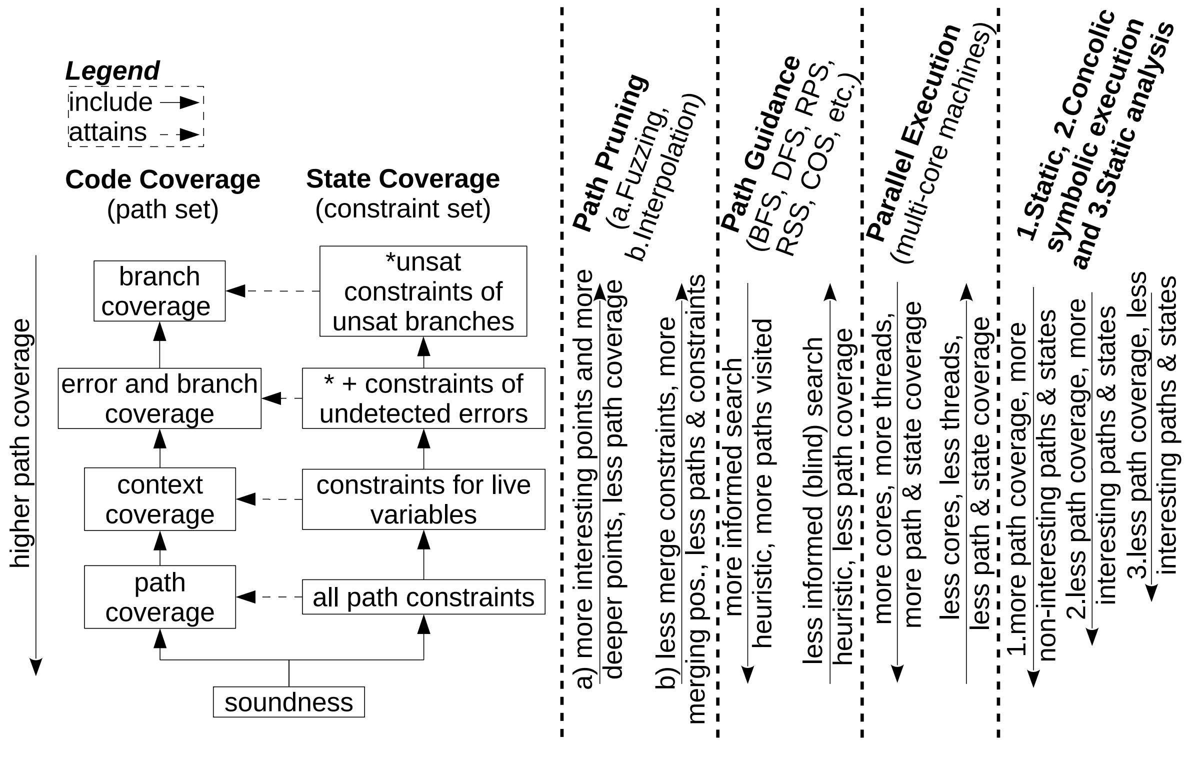

Legend
include
attains
Code Coverage
(path set)
State Coverage
(constraint set)
higher path coverage
branch coverage
*unsat constraints of unsat branches
error and branch coverage
* + constraints of undetected errors
context coverage
constraints for live variables
path coverage
all path constraints
soundness
Path Pruning
(a.Fuzzing, b.Interpolation)
a) more interesting points and more deeper points, less path coverage
b) less merge constraints, more merging pos., less paths & constraints
Path Guidance
(BFS, DFS, RPS, RSS, COS, etc.)
more informed search heuristic, more paths visited
less informed (blind) search heuristic, less path coverage
Parallel Execution
(multi-core machines)
more cores, more threads, more path & state coverage
less cores, less threads, less path & state coverage
1.Static, 2.Concolic symbolic execution and 3.Static analysis
1.more path coverage, more non-interesting paths & states
2.less path coverage, more interesting paths & states
3.less path coverage, less interesting paths & states

Figure 2.2: Program path and state coverage vs. static and dynamic analysis techniques.

Figure 2.2 depicts the code coverage (*i.e.,* program path coverage) and state coverage (*i.e.,* symbolic variable coverage) w.r.t. the most used analysis techniques to address the detection of integer overflow bugs. As far as we know, there is no technique which can be used for solving the problem of integer overflow detection. Several techniques have emerged over time with more or less applicability depending on the concrete scenario in which they are applied. We do not intend to review the advantages and disadvantages of these techniques w.r.t. to each other, but rather briefly stress why we decided to use static symbolic execution for the generation of our code repairs and briefly highlight its advantages. Consequently, there are several techniques which can be used to detect and repair integer overflow with more or less success. These techniques will be briefly compared against each other w.r.t. to program path coverage and state coverage. We decided to use these dimensions, depicted in Figure 2.2, as they make the most sense for our goals mentioned in Section 5.1. In order to reach these goals, we want to achieve high path coverage and state coverage. The generated source code repairs should be sound w.r.t. the fact that these should not change program behavior and the fault should be correctly removed after the repair was applied. For this reason, we opted for static symbolic analysis which achieves higher path coverage than concolic or purely static analysis techniques by visiting program paths in a depth-first search (DFS) fashion. Our static analysis technique benefits from the possibility of parallel execution which can be effectively used to speed up the analysis. Not only does this allow more paths to be visited, but also more states can be analyzed simultaneously as opposite to single-threaded scenarios which are more limited. For this purpose, we currently use a DFS strategy of path traversal. This strategy helps us perform a more informed search space traversal than the one performed without this technique. We plan to implement other techniques in the future. We currently perform path pruning by merging paths based on dead variables and checking satisfiability of paths at branch nodes. This helps drastically reduce uninteresting paths as well as reducing search locations. Additionally, we only check interesting program locations (*e.g.,* assignments) in the source code for integer overflow bugs, thus further reducing the possible search space. Finally, techniques such as fuzzing and interpolation have high priority targets on our future work agenda and we think that these can be implemented into our tool as well.

2.2.2.1 Symbolic Execution Based Input Validation

Symbolic execution-based techniques can be used to achieve all of the guarantees mentioned above. Furthermore, the repairs are cheap to construct and to insert. On one hand, symbolic execution-based techniques can achieve more guarantees than other repair generation techniques. On the other hand, these techniques are based on computationally intensive analysis strategies which if not applied in a suited manner, may not scale well (or at all) with large programs. We believe that repair tools should be used early by programmers during development, since the level of software complexity is low and gets higher as the number of code lines increases. Finally, we believe the following. First, manually-written source code repairs should be avoided and only used in *easy-to-address* situations. Second, compilers should not be used for repairing integer overflows since the number of guarantees which they can offer is low. Finally, specialized tools which provide more guarantees should be used for repair generation.

2.2.3 Avoiding Integer Overflows

It is important to avoid integer-overflow-based memory corruptions since these are insidious, costly and exploitable [73]. The *exploitability* of integer-overflow-based memory corruptions is a well understood topic and, for this reason, *not very difficult to be performed* by a skilled attacker, for programs written in `C/C++` since these programing languages are notoriously prone to integer overflow bugs. Code containing such a memory corruption induces a *large attack surface* which can even be *exploited through the network* by attackers. Thus, the attackers may perform CRAs which may result in serious consequences for all systems running that particular source code version. Open source code can be studied by attackers and new *integer overflow bugs can be detected with relatively low effort* and even without tool support. *Zero-day integer overflow bugs* in open source software have *disastrous consequences* since these are easily exploitable and huge gains can be achieved by the attackers. Finally, *integer overflow based vulnerabilities are traded online*, and *attackers can buy integer-overflow-based vulnerabilities* for a fraction of the potential damage or the achievable attacker benefit. As a matter of fact, we believe that for these reasons and others not mentioned here for brevity, software should be kept as *clean* as possible from integer overflow bugs.

2.3 Mitigating Integer Overflows

Before presenting the technical details of our approach, we highlight the necessary background information required to better understand the rest of this thesis.

2.3.1 Symbolic Execution Engine

IntRepair uses a symbolic execution approach for fault detection and repair generation. A symbolic execution engine [196, 200] constructs a control flow graph (CFG) for each analyzed program and extracts execution paths. Constraints along the execution path are encoded into SMT equations, using the SMT-LIB format SMT-LIB [15]. The translation of CFG nodes into SMT is performed by a translator algorithm, which extends the program's abstract syntax tree (AST) visitor class, according to the visitor pattern [84]. This is usually based on a *bottom-up* traversal of each program statement located on the currently analyzed program execution path.

In our work, we use the Codan static symbolic execution engine [196, 113, 114]. In Codan, single static assignment (SSA) variables are created for `C` expressions, which are associated with no variables in the analyzed program. Before creating a new variable, the interpreter checks whether there is already a symbolic variable. Further, for all SMT formulas created for a particular program statement, one symbolic variable is created for each of the variables contained in the original program statement. Next, a single path is extracted from the previously computed CFG and traversed. In Codan: (1) loops can be traversed a configurable number of times, (2) the analysis can be customized to look, for example, for the first *N* faults located on the currently analyzed program path, and (3) Codan performs path-caching and backtracking traversal, which avoids traversing the whole program path from the beginning and collecting all constraints again. Codan uses the Z3 [68] solver as its backend in order to solve SMT constraints.

2.3.2 Program Input Validation

First, there are several techniques which can be used to repair integer overflow based memory corruptions. Most of these techniques are based on program input validation. Of all these techniques, manual fixing is the most tedious and provides fewer guarantees than the other techniques. Next, we present some possible research paths and compare them against each other in order to answer the question stated above.

2.3.2.1 Manual Input Validation

Manually written input validation checks for repairing integer overflows have the following properties. First, they are prone to error and take much time to be inserted in large code bases. Second, they cannot guarantee that the integer overflow bug was really removed for tricky code locations (*i.e.,* multi-dependent control-flow-based source code locations). Finally, they are inapplicable across multiple integer precisions, and cannot guarantee that the intended behavior of the program is preserved.

However, there are multiple real-life documented use cases where an integer overflow bug was repaired after a cascading array of repairs was applied one after the other. We dub this type of trail of repairs as *try-and-error repairs*. The main disadvantage of such repairs is that they give the impression to the tool user that the bug was removed, when in reality the bug was not fixed.

2.3.2.2 Compiler-based Input Validation

Compiler-based input validation checks are cheap, fast to insert but can be optimized away by some compilers. When using the GCC compiler, some input validation checks are useless because they might be optimized/removed during compilation since the C++ standard N4296 [120] specifies that integer, arithmetic and signed overflows are considered undefined behavior, thus implementation specific. Furthermore, the GCC developers believe that the programmers should detect the overflow before it happens rather than using the overflowed result to check the existence of the overflow during runtime (see detailed discussion [139, 87]). First, this is impossible in some situations since the search space for program inputs that trigger an integer overflow is infinite. Second, as a consequence of compiler implementation specifics, some checking conditions may be removed totally when the program is compiled with GCC in combination with specific optimization options. On one hand, compiler-based repairs do not guarantee that the repair really removed the bug, are not applicable across multiple integer precisions and do not guarantee that unwanted behavior is introduced. On the other hand, compiler-based runtime checks have access to more specific information than static tools. Thus, in some scenarios, they can provide considerable benefits w.r.t. bug prevention. Finally, we believe that stand-alone compilers should not be used for repairing integer overflows during compile time. In contrast, specialized tools which can provide more guarantees should be used.

2.3.2.3 Symbolic-execution-based Input Validation

Symbolic-execution-based techniques can be used to achieve all of the desired repair guarantees mentioned above. Furthermore, the repairs are cheap to construct and to insert. On one hand, symbolic execution-based techniques can achieve more guarantees than other repair generation techniques. On the other hand, these techniques are based on computationally intensive analysis strategies which, if not applied in a suited manner, may not scale well (or at all) with large programs. We believe that repair tools should be used early on by programmers during development, since the level of software complexity is low and it gets higher as the number of code lines increases. Thus, we believe the following. First, manually written source code repairs should be avoided and only used in *easy-to-address* situations, Second, compilers should not be used for repairing integer overflows since the number of guarantees which these can offer is low. Finally, specialized tools which provide more guarantees should be used for repair generation.

2.4 Buffer Overflows

2.4.1 History

Buffer overflows were first partially understood and documented by Anderson [4] which describes that by exploiting a buffer overflow, new code can be injected by an attacker and this way, the attacker could gain control over the machine. By 1988, the first documented buffer overflow was used by the Morris worm in order to propagate itself over the Internet [192]. In 1995, this buffer overflow was rediscovered and the findings were published by providing more details. Later in 1996, Elias Levy (know as Aleph One) published a step-by-step introduction to exploiting buffer overflow vulnerabilities [214]. Further, in 2001, the Code Red worm exploited a buffer overflow in Microsoft's Internet Information Services [157] and in 2003, the SQL Slammer worm compromised millions of Microsoft's SQL Server 2000 machines. Lately, in 2003 buffer overflows in Xbox video games were exploited to bypass the license-based software protection. Similar exploits were published for PlayStation 2 as well. Finally, the cat an mouse game between attackers and defenders continues as `C/C++` programming languages are some of the most prominent choices for developing software systems and are nowadays the dominating programming languages for developing OSs.

2.4.2 Description

A buffer overflow occurs when data written to a buffer also corrupts data values in memory adjacent to the destination of the buffer due to insufficient bounds checking. More precisely, this appears when copying data from a larger buffer to smaller one without previously checking data size and available space. During an exploit, this data that is being copied is most likely a program which can then be executed by the attacker to perform malicious computations.

2.4.3 Exploitation

Stack-based buffer overflows are used by the attacker to manipulate the program in several ways as follows. First, by overwriting a local variable that is located near the vulnerable buffer on the stack in order to change the behavior of the program. Second, by overwriting the return address in a stack frame. Third, by overwriting a function pointer or exception handler which is afterwards executed. Finally, by overwriting a local variable or pointer of a different stack frame which will be used by the function which owns that frame later.

Heap-based buffer overflows appear on the program heap. These vulnerabilities are exploitable in a different way than stack-based buffer overflows. During a heap-based buffer overflow, program data is corrupted in order to cause the application to overwrite internal structures such as linked list pointers. For example, the dynamic memory allocation linkage of `malloc` is overwritten and the resulting pointers are used to overwrite a program function pointer.

2.4.4 Protection

Static program analysis (source code, machine code or intermediate representation) can be used to check for the presence of buffer overflows. These techniques have their limitations which are dependent on the complexity of the analyzed program and the runtime which has to be simulated. Next, dynamic and hybrid approaches alleviate some of the limitations of the static techniques but still can not guarantee that all buffer overflow are detected.

The buffer overflows are most common in `C/C++` programming languages and as such, other programming languages which are strongly typed such as Java and Python are recommended, which can help prevent buffer overflows [215]. Other programming languages (*e.g.,* Ada, Eiffel, Lisp, Smalltalk, OCaml, *etc.*) provide runtime checking or even compile-time checking. These approaches can be used to detect or prevent buffer overflows. Often times security is traded for performance when deciding which programming language and compiler flags to use.

Safe libraries are yet another way to mitigate against buffer overflows. Safe libraries provide alternatives for string manipulation functions such as `gets`, `fscanf`, `fscanf` which should be avoided. The available safe libraries perform bounds checking on the data that is being manipulated.

Systems which check if the stack was altered after the called function returns are Libsafe [136], StackGuard [58], and ProPolice [116]. For example, Microsoft's Data Execution Prevention (DEP) explicitly protects the pointer to the Structured Exception Handler (SEH) from being overwritten. Other protection techniques such as the shadow stack implementations available in the Clang and GCC compiler can also protect against buffer overflows, since the data and pointers are split between the two stacks.

Pointer-based protection such as PointGuard [57] is a compiler-based protection which can be used to protect pointers and addresses which are used by an attacker. Pointers are XOR-encoded before and after usage. The idea behind this approach is that the attacker theoretically does not know what the value will be used to encode and decode the pointer and in this way he cannot predict where it will point to in case he overwrites it with a new value.

Data Execution Prevention (DEP) is used to prevent the execution of code on the stack or on the heap. This way, injected code cannot randomly be executed. An attacker may need first to deactivate DEP. Further, DEP does not protect against return-to-libc attacks or any other attack which does not rely on the execution of the attacker code.

Address Space Layout Randomization (ASLR) is a security feature which involves arranging randomly program data areas in a process address space. This technique makes buffer-overflow-based exploits more difficult but not impossible since during an exploit the attacker needs to know the layout of the vulnerable program. The layout can eventually be disclosed by a data layout leakage, which may help the attacker disclose the binary layout.

Fuzzing [230] is another promising technique for detecting buffer overflows. This technique is based on providing a considerable amount of program inputs with the goal to trigger buffer overflows during runtime. Further, software testing based on programmer-provided test cases can also be successfully used to detect buffer overflows.

2.5 C++ Object Type Confusion

Before presenting the technical details of our approach, we review necessary background information.

2.5.1 C++ Type Casting

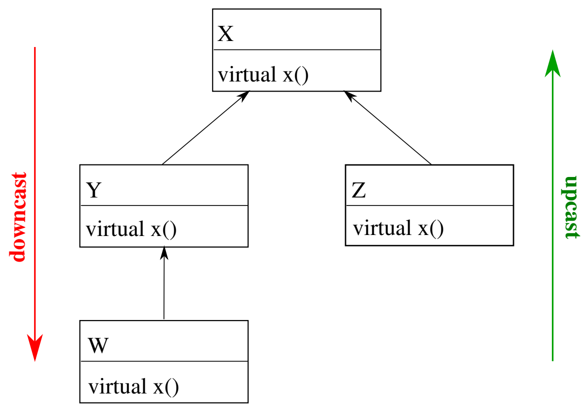
Object type casting in `C++` allows an object to be cast to another object, such that the program can use different features of the class hierarchy. Seen from a different angle, object typecasting is a `C++` language feature, which augments object-oriented concepts such as inheritance and polymorphism. Inheritance facilitates that one class contained inside the program class hierarchy inherits (gets access) to the functionality of another class that is located above in the class hierarchy. Object casting is different, as it allows for objects to be used in a more general way (*i.e.,* using objects and their siblings, as if they were located higher in the class hierarchy). `C++` provides `static`, `dynamic`, `reinterpret` and `const` casts. Note that `reinterpret_cast` can lead to bad casting, when misused and is unchecked *by design*, as it allows the programmer to freely handle memory. In this thesis, we focus on `static_cast` and `dynamic_cast` (see N4618 [119] working draft), because the misuse of these can result in bad object casting, which can further lead to undefined behavior. This can potentially be exploited to perform, for example, local or remote code reuse attacks on the software.

The terminology used in this thesis is aligned to the one used by colleagues [107], in order to provide terminology traceability as follows. First, *runtime type* refers to the type of the constructor used to create the object. Second, *source type* is the type of the pointer that is converted. Finally, *target type* is the type of the pointer after the type conversion.

An *upcast* is always permitted if the target type is an ancestor of the source type. These types of casts can be statically verified as safe, as the object source type is always known. Thus, if the source type is a descendant of the target type, the runtime type also has to be a descendant and the cast is legal. On the other hand, a *downcast* cannot be verified during compile time. This verification is hard to achieve, since the compiler cannot know the runtime type of an object, due to intricate data flows (for example, inter-procedural data flows). While it can be assumed that the runtime type is a descendant of the source type, the order of descent is unknown. As only casts from a lower to a higher (or same) order are allowed, a runtime check is required to check this.

2.5.2 C/C++ Legal and Illegal Object Type Casts

A type cast in C/C++ is legal only when the destination type is an ancestor of the runtime type of the cast object. This is always true if the destination type is an ancestor of the source type (upcast). In contrast, if the destination type is a descendant of the source type (downcast), the cast could only be legal if the object has been upcast beforehand.



(a) Class hierarchy with four classes.

```
/* downcast */
X *x = new W();
Y *y = static_cast<Y>(x);

/* first upcast*/
Z *z = new Z();
X *x = static_cast<X>(z);

/* second upcast */
Y *y = new W();
X *x = static_cast<X>(y);
```

(b) Downcast and upcast cast in C++.

Figure 2.3: C++ based object type down-casting and up-casting examples.

Figure 2.3 depicts object upcast and downcast operations with respect to a given class hierarchy. The graph of Figure 2.3(a) is a simple class hierarchy. The boxes are classes, and the arrows depict inheritance. The code of Figure 2.3(b) shows how upcast and downcast look in C++. The upcast and downcast arrows besides the graph visualize the same casts that are coded in C++ in Figure 2.3(a). To verify the cast, the runtime type of the object is needed. Unfortunately, the exact runtime type of an object is not necessarily known to the compiler for each cast, as explained in Section 2.5.1. While the source type is known to the compiler for each cast, it can only be used to detect very specific cases of illegal casts (*e.g.*, casts between types that are not related in any way, which means they are not in a descendant-ancestor relationship). All upcasts can be statically verified as safe because the destination type is an ancestor of the runtime type. If the destination type is not an ancestor of the runtime type, then the compiler should throw an error.

2.5.3 Virtual Table Inheritance Trees

In order to encode object subtype checks with range checks over the virtual pointers values, we need to be able to validate that the runtime type of an object inherits from the destination type of the casted object. In order to achieve this, we need the class hierarchy of the protected program during runtime. Next, the virtual pointer (vptr) of an object can be used as identifier, and its value as an ordering number inside the vtable inheritance tree. By ordering we mean the exact location of the object type in the pre-order traversal of the primitive virtual table tree derived from the class hierarchy. The value of the vptr represents the vtable address of its runtime type. This therefore represents the position of the vtable of the runtime type of the object in memory. The position is decided by the compiler during compilation, which places all vtables in memory. Therefore, in order to manipulate the value of the vptr, the virtual tables (vtables) of all object types have to be placed in a certain order in memory. Since we need to control the values of virtual pointers in the produced program binary, we need explicit control over the locations and layout of the virtual tables in memory.

By default the Clang compiler does not impose a specific order on the virtual tables in memory. This means that there could exist memory gaps between the start addresses of the vtables, which can lead to an unwanted situation, namely that the vtable ranges are not tight. First, these gaps could introduce imprecision (too many addresses could be allowed) for the forward edge target sets, and thus the layout of the vtables has to be modified. Second, due to these gaps and the fact that the table layout is not ordered w.r.t. the class hierarchy, object cast checks cannot use these ranges since the resulting ranges (intervals) contain memory gaps.

In order to overcome these limitations, we use an algorithm for interleaving the vtables of a program in memory without gaps. The vtable interleaving algorithm repositions the vtables in memory with the goal to provide a minimal set of target addresses for each object dispatch on virtual functions (*i.e.,* forward edge). More specifically, the vtables of polymorphic types are reordered with the help of a pre-order traversal of the class hierarchy graph with no memory gaps between them. This way if the vptr of an object is corrupted by an attacker with the purpose for example to call a function of a class that does not respect the static type at the virtual call location, then it can be detected during runtime. This ensures that only legitimate virtual functions (the ones which respect the per object class hierarchy) are allowed at any particular object cast location.

Figure 2.4 depicts a `C++` class hierarchy containing only virtual functions (a) and the corresponding primitive vtables inheritance trees in (b) and (c). Figure 2.4, the class hierarchy and virtual inheritance trees are represented as UML class hierarchies containing classes and their inherited functions. The arrows depict inheritance, similar to UML class hierarchy inheritance. Figure 2.4(a) contains the following elements: (1) a box (node) represents a polymorphic class, a box contains its class name (first row, *i.e., X*) and its inherited virtual functions (following rows, *i.e.,* `virtual x()`), and (2) the inheritance relationship between two classes is depicted by arrows. An arrow pointing from one class to another signifies that the lower located class inherits from the class located in the upper part. Figure 2.4(b) and Figure 2.4(c) depict two virtual table inheritance trees, derived from the class hierarchy depicted in Figure 2.4(a), which

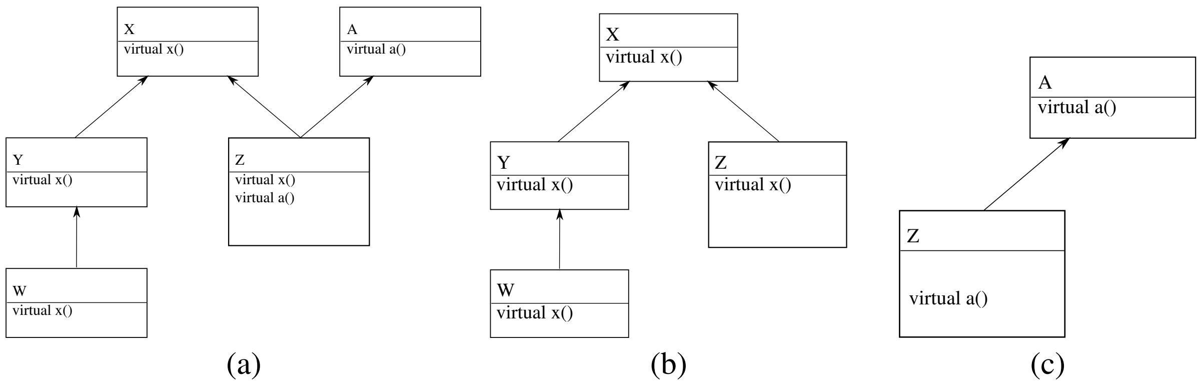


Figure 2.4: Class hierarchy containing virtual functions (a) and the corresponding vtable inheritance trees (b) and (c).

were obtained by performing a pre-order traversal starting at the two roots, X and A. These two trees result, because a polymorphic class (*i.e., Z*) has two ancestors (*i.e., X* and *A*). A box in a tree is a virtual class having its type depicted in the first row. Further, if a polymorphic class inherits from multiple polymorphic classes, then this relationship will be represented by two or more boxes in multiple trees (*i.e., Z* has a box in the tree of Figure 2.4(b) for its inheritance from *X* and a box in the tree of Figure 2.4(c) for its inheritance from *A*). Each box contains virtual functions that are inherited within the tree (*i.e.,* the box of *Z* in Figure 2.4(b) only contains the function `virtual x()`, while the box of *Z* in Figure 2.4(c) only contains the function `virtual a()`). Similar to Figure 2.4(a), an arrow depicts the inheritance between vtables.

Finally, note that a base class hierarchy can contain more primitive virtual inheritance trees. Thus, we need to be able to split complex class hierarchies into simple trees because under the hood, the C++ Itanium ABI splits objects into primitive objects, and vtables into primitive vtables in such a way that the primitive objects/virtual tables form trees. This helps decouple a complex directed acyclic graph (DAG) hierarchy into multiple simple trees.

2.5.4 Type Casting in Practice

The C++ language provides `dynamic_cast`, which can guarantee the correctness of type castings, under certain cases. Specifically, `dynamic_cast` can only be applied to polymorphic types, and only when RTTI information was emitted. Unfortunately, it is seldom used due to its inefficiency. For example, [142] showed that `dynamic_cast` is 90 times slower than `static_cast` on average. This slowdown can be traced to the recursive traversal and linear time checks performed by `dynamic_cast()`. Further, this renders this type of check ineffective mostly for large programs like for example Mozilla Firefox. Therefore, its confirmed security advantage is avoided or even forbidden in for example the Google Chrome project. Because of this, all currently legal and illegal object casts available bad cast detection tools replace the RTTI information with their own metadata, in order to optimize the type comparison which is needed to check legitimate casts.

For this reason, large software project such as Chrome use a workaround based on a function such as `isType()` which returns the true allocated type of the object. As a result, this technique successfully decouples an `dynamic_cast` into an explicit type check followed by a `static_cast` which drastically reduces the runtime check cost. Note that `dynamic_cast` is based on a very general algorithm that can account for various complex corner cases of multiple and virtual inheritance, while `isType()` is a very simple exact type comparison, that only works in the specific cases the programmer used it. Further, this approach is still error-prone since it falls under the error-prone manual modifications paradigm which is affected by (1) incorrect marked object type identity flag or (2) the absence due to human factors of the `isType()` check. Finally, even with custom RTTI information bad-casting errors still are prevalent.

2.5.5 Object Type Confusion Example in Google's V8

```
1  type* type::cast(Object* object) { SLOW_ASSERT(object->Is##type());
2      return reinterpret_cast<type*>(object);}
3  bool IsCompatibleReceiver(LookupIterator* lookup, Handle<Map> receiver_map) {
4   DCHECK(lookup->state()==LookupIterator::ACCESSOR);
5   Isolate* isolate = lookup->isolate();
6   Handle<Object> accessors=lookup->GetAccessors();
7    if (accessors->IsExecutableAccessorInfo()) {
8        Handle<ExecutableAccessorInfo> info =
9        Handle<ExecutableAccessorInfo>::cast(accessors);
10       if (info->getter() != NULL &&
11 !ExecutableAccessorInfo::IsCompatibleReceiverMap(isolate, info, receiver_map)){
12            return false;
13  }
14 } else if (accessors->IsAccessorPair()) {
15   return true;}}
```

Figure 2.5: The `IsCompatibleReceiver()` function contained in Google V8 engine. This source code is used in Google Chrome before v. 48.0.2564.82. It does not ensure receiver compatibility before performing a cast of an unspecified variable, which allows remote attackers to cause a denial of service or possibly have another unknown impact via crafted JavaScript code (see source code differential view [95]). The ## symbol represents a macro.

Figure 2.5 depicts the bug fix of CVE-2016-1612 [47] which manifests due to a bad cast at line number 11. The gray shaded lines represent all code lines that are relevant for this type confusion. This object type confusion can be circumvented by first performing a receiver compatibility check before performing a cast of an unspecified variable as depicted at line numbers 8-9. The goal of this check is to check if the state of the `lookup` object is a lookup iterator `ACCESSOR`. Without this check, the `reinterpret_cast` at line number 2 would allow remote attackers to cause a denial of service or possibly have another unknown impact via crafted JavaScript code.

A `reinterpret_cast` is particularly dangerous because it forces the reinterpretation of memory area into a different type which most commonly breaks underlying type assumptions. As such, it is recommended that it is used with care and guarded by runtime checks (*e.g.,* asserts). Finally, it is recommended (as in [107]) that `reinterpret_cast` should be avoided and either `dynamic_cast` should be used, or a pair of an `isType()` runtime check and `static_cast`.

2.5.6 Security Implications of Object Type Confusion

The application behavior becomes undefined after an incorrect bad `static_cast` is performed. In order to understand what the consequences of bad-casting are we need to understand the internals of the compilers which implement this feature. Bad typecasts allow an attacker to corrupt memory in a targeted manner. If a bad-casting occurs near to a virtual pointer the attacker can control the input to member variables or it can overwrite the virtual pointer and hijack the program execution.

Figure 2.4 depicts a C++ class hierarchy containing only virtual functions (a) and the corresponding primitive vtables inheritance trees in (b) and (c). In Figure 2.4, the class hierarchy and virtual inheritance trees are represented as UML class hierarchies containing classes and their inherited functions. The arrows depict inheritance, similar to UML class hierarchy inheritance. Figure 2.4(a) contains the following elements: (1) a box (node) represents a polymorphic class, a box contains its class name (first row, *i.e., X*) and its inherited virtual functions (following rows, *i.e.,* `virtual x()`), and (2) the inheritance relationship between two classes is depicted by arrows. An arrow pointing from one class to another signifies that the lower located class inherits from the class located in the upper part. Figure 2.4(b) and Figure 2.4(c) depict two virtual table inheritance trees, derived from the class hierarchy depicted in Figure 2.4(a), which were obtained by performing a pre-order traversal starting at the two roots, X and A. These two trees result, because a polymorphic class (*i.e., Z*) has two ancestors (*i.e., X* and *A*). A box in a tree is a virtual class having its type depicted in the first row. Further, if a polymorphic class inherits from multiple polymorphic classes, then this relationship will be represented by two or more boxes in multiple trees (*i.e., Z* has a box in the tree of Figure 2.4(b) for its inheritance from *X* and a box in the tree of Figure 2.4(c) for its inheritance from *A*). Each box contains virtual functions that are inherited within the tree (*i.e.,* the box of *Z* in Figure 2.4(b) only contains the function `virtual x()`, while the box of *Z* in Figure 2.4(c) only contains the function `virtual a()`). Similar to Figure 2.4 the control flow of the program such as in the advanced COOP [238] code reuse attack. This shows that COOP attacks do not necessarily require a buffer overflow memory corruption upfront. Furthermore, non-control-data attacks are possible too [142].

The C++ standard mandates that the program behavior is undefined after a bad typecast. In practice, bad typecasts allow an attacker to corrupt memory in a targeted manner. These memory corruptions can allow an attacker to modify control-flow relevant state such as function pointers or vtable pointers. This leads to a wide array of attacks such as ROP, JOP and some more advanced code-reuse attacks such as COOP [238], that defeat many state-of-the-art CFI defenses.

The exploitability of bad-casting depends on the possibility of performing out-of-bound memory access or manipulate memory semantics. Furthermore, this is dependent on the object layout as specified by the used C++ application binary interface (ABI). Depending on the used ABI the security implications for the same bad casting bug can be different. Finally, as the treat posed by bad-casting errors is alarming, the awareness of this error should be raised to avoid future security breaches. For example, CVE-2016-1612 [47] where a bad-cast in Google V8 engine used by Chrome running on Mac, Windows, Linux and Android could be exploited. This vulnerability allows remote attackers to cause a denial-of-service or possibly have another unknown impact via crafted JavaScript code. The fix of this vulnerability was rewarded with 3k in cash by Google [47].

2.5.7 Object Type Confusion Defenses

There are several recent lines of research aiming to protect against bad typecasting. Haller *et al.* [107] point out that we can broadly split object type confusion defenses into two categories: (1) techniques based on an embedded vtable pointer in the casted object, and (2) techniques leveraging disjoint metadata. Embedded vtable pointers which can be used only for polymorphic casts have the advantage that the runtime cast checks based on these can be performed relatively fast. Compared to that, metadata approaches where the type of the object has to be tracked during runtime can reach higher coverage while being fundamentally slower.

Checker	**Poly**	**Non-poly**	**No blacklist**	**Tracking**	**Threads**
UBSan [99]	✓				✓
Clang CFI [149]	✓		✓		✓
CaVer [142]	✓	✓	✓	✓	limited
TypeSan [107]	✓	✓	✓	✓	✓
HexType [125]	✓	✓	✓	✓	✓
CastSan [201]	✓		✓		✓

Table 2.1: High-level features overview of object type confusion checking tools.

Table 2.1 depicts the high level feature overview of CastSan and other comparable tools. CastSan covers all casts between classes that are covered by the IVT tool. This includes all polymorphic classes which do not inherit from classes of the `std` namespace. Classes that do not have a vtable can not be checked w.r.t. object casts. As CastSan does not rely on runtime object tracking but does solely need the virtual function pointer of the object to perform its check, CastSan can check any casts that are performed within threads that are not the main thread of the program and does not need a blacklist. Also, other sanitizers like HexType depend on objects to be allocated by a trackable allocator to be able to insert metadata of objects on construction. Because CastSan does not need additional metadata, it can check object casts regardless of allocation method. Therefore, no further limitation has to be noted for checking casts between polymorphic classes. The coverage was tested using the benchmark suites of ShrinkWrap, TypeSan and IVT. Using these benchmarks, there was no case that CastSan could not correctly handle. When compiling the Chrome 33 source code using CastSan, 30757 (it contains in total 50994 object casts) typecasts are instrumented in total. Non-polymorphic classes on the other hand can not be checked due to the absence of a vtables.

UBSan [99] and Clang-CFI (cast checker) [149] are both based on vtables. UBSan instruments static casts in order to effectively turn them into dynamic casts and it requires manual blacklisting in order to avoid false negatives due to non-polymorphic classes. The type checking mechanism of UBSan is inherently slow and for this reason, UBSan is intended as a testing tool rather than always on solution due to its high overhead. UBSan can not support non-polymorphic classes.

CaVer [142], HexType [125] and TypeSan [107] are based on disjoint metadata. CaVer supports non-polymorphic classes by relying on disjoint metadata and without blacklisting. CaVer has high performance overhead due to inefficient metadata tracking and slow checks even doubling the runtime overhead. CaVer cannot rely on vtables and for this reason, it has to track all live objects during runtime. Due to its inefficient underlying implementation, it can not handle stack objects shared between threads even with perfectly implemented synchronization. CaVer has a reduced object allocation coverage in practice which leads to limited type-confusion detection coverage compared to TypeSan. TypeSan is yet another tool which is based on disjoint metadata that reduces the runtime overhead of checking object typecasts and which can detect errors which are hard to reproduce during testing. TypeSan is based on object tracking which supports all types of classes with no blacklisting required.

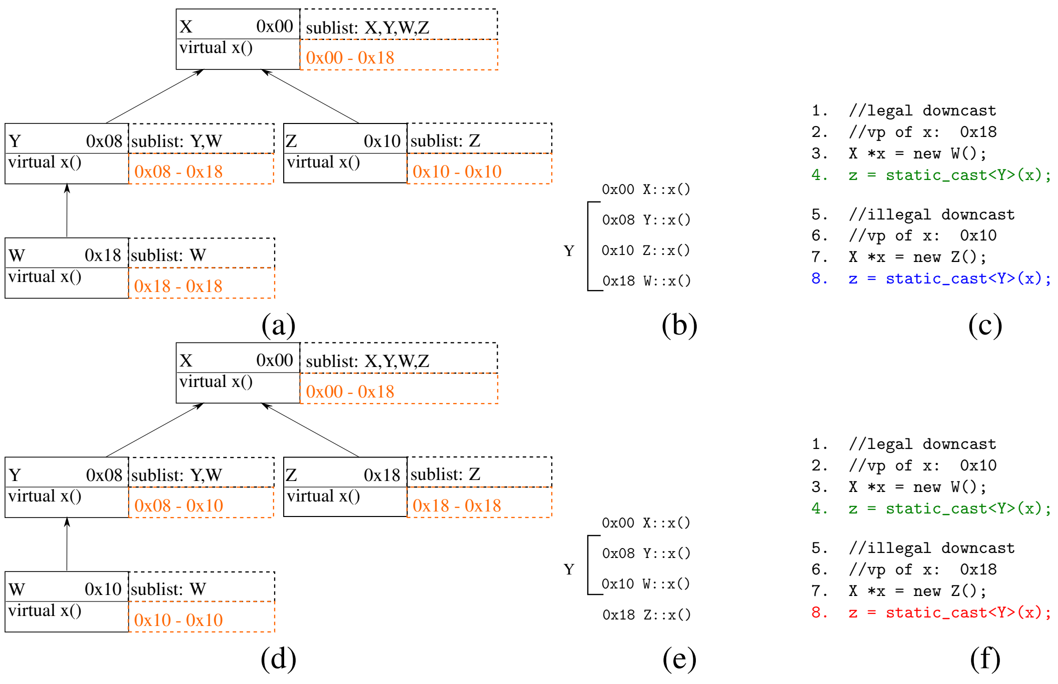
Clang-CFI cast checker achieves a lower performance overhead compared to other state-of-the-art tools by optimizing its use of type metadata [52] to encode relationships between class object types based on class hierarchy relationships. Specifically, Clang-CFI embeds a bitmap for each class A, encoding the valid starting offsets for all vtables for objects of type A (or any of its subtypes). To reduce the size of these bitmaps, Clang-CFI relies on several optimizations: (1) all vtables for class A start at some base offset Base, and (2) all vtables for class A are aligned by some power of 2 (*e.g.,* 1,024.0). Runtime checks in Clang-CFI consist of a range check and a bit lookups in the appropriate bitmap. In some cases Clang-CFI can optimize away the bit lookup, or even the range check. Clang-CFI can not support non-polymorphic object casts since it uses vtable pointers to perform lookups in the bitmaps.

In this thesis, we present CastSan, a generic solution for object type confusion detection during runtime based on efficient representation of class hierarchies and fast range-based checks. CastSan does not require blacklisting and supports all polymorphic object types. Furthermore, it does provides better typecasting results as the above presented tools w.r.t. performance.

2.5.8 Ordered vs. Unordered Virtual Tables

In this Section, we briefly describe the differences between in-memory ordered and unordered vtables and how these can be used to detect object type confusions during runtime.

Figure 2.6(a), Figure 2.6(b), and Figure 2.6(c) highlight the case in which an illegal object cast would not be detected if the vtables are not ordered (see blue shaded code in line number eight), while Figure 2.6(d), Figure 2.6(e), and Figure 2.6(f) show how a legal (see green shaded



```
1.  //legal downcast
2.  //vp of x:  0x18
3.  X *x = new W();
4.  z = static_cast<Y>(x);

5.  //illegal downcast
6.  //vp of x:  0x10
7.  X *x = new Z();
8.  z = static_cast<Y>(x);
```

```
1.  //legal downcast
2.  //vp of x:  0x10
3.  X *x = new W();
4.  z = static_cast<Y>(x);

5.  //illegal downcast
6.  //vp of x:  0x18
7.  X *x = new Z();
8.  z = static_cast<Y>(x);
```

Figure 2.6: Illegal and legal object casts vs. ordered and unordered virtual tables.

code in line number four) and an illegal (see red shaded code in line number eight) object cast can be correctly identified by using the object vptr in case the vtables are ordered in memory.

Note that the main difference between the vtables' layouts depicted in Figure 2.6(b) and Figure 2.6(d) is that the vtable layout entries depicted in Figure 2.6(d) are not arranged in pre-order traversal, while the vtable entries shown in Figure 2.6(d) are arranged in pre-order traversal. Further, note that, (1) green shaded code indicates that the legal downcast was correctly detected (program execution continues), (2) blue highlighted code means that the illegal downcast was not detected (program execution continues), (3) red shaded code means that the illegal downcast was correctly detected (program execution stops or an error log is printed).

Next, we will present the contents of the six sub-figures depicted in Figure 2.6. Figure 2.6(b) depicts the unordered vtable, while Figure 2.6(e) depicts the ordered vtable. Figure 2.6(b) and Figure 2.6(e) represent how the vtables' addresses of the hierarchies are appearing in memory. Both vtables are ordered by their vtable addresses; Figure 2.6(e) shows the vtable addresses are ordered and used by CastSan. Figure 2.6(b) depicts the vtable addresses unordered as they are present in vanilla LLVM. The vtable address ordering depicted in Figure 2.6(e) corresponds to the hierarchy depicted in Figure 2.6(d).

The vtable hierarchy trees depicted in Figure 2.6(a) and Figure 2.6(d) contain in the attached black dotted frames the sub-lists of allowed types, while in Figure 2.6(b) and in Figure 2.6(e) we can observe in the orange dotted frames the allowed types ranges for legal object casting before and after the pre-order traversal, respectively. Note that the differences between Figure 2.6(a) and Figure 2.6(d) are represented by the ranges, which are depicted in the orange shaded boxes attached on the right side of each black shaded box. Further, Figure 2.6(a) and Figure 2.6(d) depict the same virtual inheritance tree with different vtable address values. In Figure 2.6(d), the vtable addresses are ordered by the interleaving algorithm as used by CastSan, while in Figure 2.6(a) they are not. Both trees contain: (1) the classes and their functions, (2) their vtable address (top right of the box), and (3) in addition to that, their sub-list (black dotted frame). These contain all inheriting types and the vtable range from the first element of the sub-list to the last element of the sub-list (orange dotted frame).

The code listings depicted in Figure 2.6(c) and Figure 2.6(f) depict the same `C++` object type casts, with the vptr of the constructed object depicted in the comments for each case. Both these code listings first depict a legal cast, followed by an illegal object cast.

On one hand, Figure 2.6(c) shows the vptr value as it would be present in the unordered case of Figure 2.6(b) and Figure 2.6(a). The object x, that is constructed at line number seven with the constructor of Z (runtime type) has a vptr of value `0x18` in the unordered case. x is referenced by a pointer of type X (source type) and at line number eight it is cast to Y (destination type). This is an illegal object cast, as Z does not inherit from Y. The vptr of x is in the range of Y built from the unordered vtable layout of Figure 2.6(b). A range check would, therefore, falsely conclude that the cast is legal.

On the other hand, Figure 2.6(f) depicts the same objects as constructed after ordering according to Figure 2.6(e) and Figure 2.6(d). At line number three, the object x is instantiated having (runtime) type W. The object, therefore, has a vptr with value `0x10` according to Figure 2.6(d). The object is referenced by a pointer of type X (source type) and at line number four, the object x is cast to Y (destination type). This cast is a legal object cast, as the vptr `0x10` has a value between the vtable address of Y `0x08` and the address value of the last member of the sub-list of Y `0x10`. Note that this memory range is depicted in Figure 2.6(e). Further, in line number seven, the object x is newly allocated with the constructor of Z. Next, the object is cast to Y at line number eight. As x's vptr is `0x18`, which is the vtable address of Z, it can be observed that the cast is illegal. The reason is that the vptr value `0x18` is larger than the largest value of the sub-list of Y, which is the vtable address of W, `0x10`. Thus, in this way the object type confusion located at line number eight can be correctly detected.

Finally, note that the range checks, which we will use in our implementation, are precise, when the vtables of all program hierarchies are ordered with no gaps in memory according to, for example, their pre-order traversal. In case this is not guaranteed, then the range checks could generate false positives as well as false negatives (see the blue shaded code in Figure 2.6(c)).

2.6 Code Reuse Attack Primitives and Mitigation

In this section, we highlight the necessary background information in order to better understand the rest of this thesis.

2.6.1 Code Reuse Attack Primitives

In order to better understand code reuse attacks (CRAs), we describe what happens during a CRA at the source code and binary level.

Forward-edge-based code reuse attacks exploit the forward-edges of CFGs. First, at the source code level by performing calls through either function pointers (*e.g.,* single level of indirection) or virtual calls (*e.g.,* double level of indirection). For example, these calls may use an array of function pointers that is accessed by a pointer to a virtual table (vtable) plus an index. Second, at the binary level, `jump`, and `call` instructions are used to redirect the program control flow to a different address than the one intended in the original program CFG.

Backward-edge-based code reuse attacks violate CFG backward edges. First, at the source code level, the function will not return to the next source code line from where the function was first called. Second, in the program binary, the address located on the stack is modified such that the function's `ret` instruction will return to a different address than the one next to the instruction from where the function was initially called (mostly through a `call` instruction).

Finally, these two types of primitives (forward and backward edges) are used to link gadgets, in order to form a gadget chain with the goal of performing Turing-complete malicious computations.

Backward/forward-edge based code reuse attacks violate both forward and backward edges inside a program, in order to perform the intended malicious behavior. Note that most CRAs either use forward or backward edges when performing the malicious computations.

Gadget Types. Each code reuse attack has its own set of gadgets. Interestingly, the gadgets have a runtime behavior similar to the original program instructions and are mainly subsets or supersets of existing gadget sets. Currently, the smallest gadget w.r.t. the number of machine instructions has two instructions (*i.e.,* two consecutive `push` and `pop` instructions, equivalent to a return). In addition, the largest gadget comprises a whole virtual function (*i.e.,* Counterfeit Object Oriented Programming (COOP) [238]). Note that while for a certain type of attack a whole virtual function could represent one gadget, for other CRAs this function may contain many gadgets. From here, we observe the afore mentioned subset relationships. These relationships are dependent on the notion of gadget length which varies and is constrained by the gadget set used and the goals of the attacker.

Gadget Availability means that a certain gadget is present in a certain program or library and is not constrained by any protection technique. Note that a gadget can be available for a certain attack and not for other types of attacks. This is due to the fact that certain protection techniques are too coarse grained w.r.t. certain attacks.

Gadget Usability means that a gadget is useful within certain CRA types. Note that each CRA has its own set of gadgets. In this case, we assume that while a gadget is available (*i.e.,* protected or not) it can be usable or not usable by a certain type of CRA. This property indicates if a gadget belongs to the set of gadgets of the current CRA type.

Gadget Linkability is based on the concept of availability and usability of a certain gadget. In case that a gadget is available and usable, linkability describes how easy or difficult it is for an attacker to use this gadget at a certain position inside a gadget chain. In this case, the difficulty is directly proportional to: (1) number of operations which have to be performed in order to link a gadget to another and (2) for example, in case a function primitive (*i.e.,* `int mprotect(void *addr, size_t len, int prot);`) is not accessible, linkability describes how difficult it is to get to this primitive and make it usable during the attack.

2.6.2 Control Flow Integrity

Control-Flow Integrity (CFI) is a state-of-the-art technique used successfully along other techniques to protect forward and backward edges against program control flow violations. CFI is used to mitigate CRAs by, for example, pre-pending an indirect callsite with runtime checks that make sure only legal calltargets are allowed by an as precisely as possible computed control flow graph (CFG) [1].

Protection Schemes. Alias analysis in binary programs is undecidable [229]. For this reason, when protecting CFG forward-edges, defenders focus on using other program primitives to enforce a precise CFG during runtime. These primitives are most commonly represented by the program's: class hierarchy [106], virtual table layouts [226], quasi-class hierarchies [218], binary function types [268] (callsite/calltarget parameter count matching), *etc.* They are used to enforce a CFG which is as close as possible to the original CFG being best described by the program control-flow execution. Note that state-of-the-art CFI solutions use either static or dynamic information for determining legal calltargets.

Static Information. CFI solutions that use static information allow callsites to target: (1) all function entry points *e.g.,* [290], map callsite types to target function types by creating a mask which enforces that the number of provided parameters (up to six) has to be higher than the number of consumed parameters *e.g.,* [268], (2) a quasi-class hierarchy (no root node(s) and the edges are not oriented) can be recuperated from the binary and enforced *e.g.,* [218], (3) all virtual tables that can be recuperated and enforced *e.g.,* [226], only certain virtual table entries are allowed *e.g.,* [286] based on a precise function type mapping, and (4) sub-class hierarchies are enforced *e.g.,* [261, 106, 24].

Dynamic Information. The goal of CFI solutions which use dynamic information is to refine their runtime analysis by leveraging program information which is only available during program execution. In particular, PiCFI [212] restricts the set of calltargets to functions which have their address computed during runtime. Context-sensitive solutions with different levels of context precision rely on hardware features such as the Last Branch Register (LBR) [266] to track a limited range (*i.e.,* 16 up to 32 address pairs) of so-called *from* and *to* addresses pairs during runtime. They then compare them against a precomputed program CFG. Finally, note that Intel Processor Trace (PT) [89] can be used to build a longer history of address pairs compared to other approaches.

2.7 Program Callee Primitives and Mitigation

Before presenting the technical details of our approach, we highlight the necessary background information required to better under- stand the rest of this thesis. In Section 2.7.1, we present program indirection and relate it to C/C++ programs, and in Section 2.7.2, we describe several types of program callsites, while in Section 2.7.3 we discuss program control flow graph edges. Finally, in Section 2.7.4, we present several basic concepts related to virtual tables hierarchies.

2.7.1 Indirect Control Flow Transfers

Program indirection is represented by any program control flow transfer which is performed for example with the help of a pointer, other types of level of indirection (trampolines) or trough registers. Basically speaking, program indirection is represented by the control flow graph (CFG) edges which can not statically determined due to input dependency, complex control flow, *etc.* While static edges are fixed edges inside the CFG, dynamic edges are not present or hard to be determined due to program indirection. These type of edges exist due to object dispatches (forward edges), pointers accessing different fields of data structures, function returns (backward edges), *etc.* In the context of control flow integrity (CFI), researchers are categorizing program indirection into forward and backward CFG edges as these type of edges are usually used to link gadgets together. Further, code reuse attacks violate forward and/or backward edges since it is currently very difficult to protect these control flow transfers due to the fact that alias analysis in program binaries is undecidable [229]. Finally, other useful program metadata (*i.e.,* class hierarchies, virtual tale hierarchy) is not available, not usable (for C only programs which have not class hierarchies), hard to be statically or dynamically determined with high precision.

2.7.2 Program Callsite Types

We divide callsites into three distinct type: direct, virtual or indirect callsites. A direct callsite has a single fixed callee, a virtual callsite uses the C++ virtual dispatch mechanism, and lastly an indirect callsite, which most commonly uses a function pointer for an indirect call. These three types are all handled slightly differently:

- *D:* **Direct callsites.** This is the easiest case, which is handled as explained in Section 10.3.
- *V:* **Virtual callsites.** In this case, we use class hierarchy information to infer a range of IDs, which includes the IDs of all function implementations, which can be called by this callsite. The callsite then gets assigned two `NOPs` carrying the range data. This case is explained in detail in Section 10.3.
- *I:* **Indirect callsites.** For indirect callsites, we use a function signature matching technique, which is explain in more detail in Section 10.3.

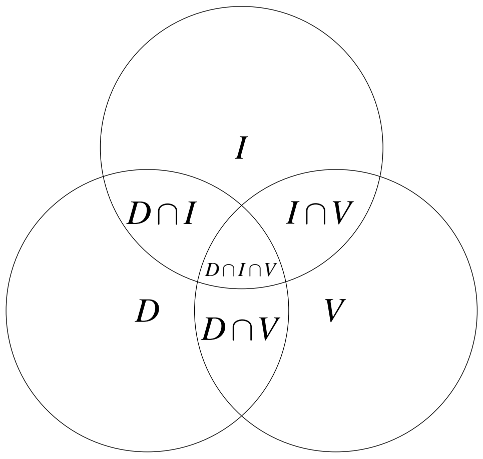


Figure 2.7: Types of callsites for a particular callee.

Figure 2.7 depicts the fact that even though a particular callsite has only a single type at a time, a particular callee might have multiple callsites and therefore can be called using more than one of these types. Nevertheless, we want to briefly highlight the other callsite types which can be used to call a particular callee.

- *$D \cap I$:* **Direct and Indirect Callsites Intersected.** There are callees which can be called by both direct and indirect callsites.
- *$D \cap V$:* **Direct and Virtual Callsites Intersected.** There are callees which can be called by both direct and virtual callsites.
- *$I \cap V$:* **Indirect and Virtual Callsites Intersected.** There are callees which can be called by both indirect and virtual callsites.
- *$D \cap I \cap V$:* **Direct, Indirect and Virtual Callsites Intersected.** There are callees which can be called by direct, indirect and virtual callsites.

2.7.3 Control Flow Backward Edges

Return edges in assembly code are represented by function return instructions (`ret` instruction) and are used to return the control flow of the program to the address after the callsite which originally called the function. Depending on the instruction set architecture (ISA) (*i.e.,* x86, x86-64, ARM, SPARC, *etc.*) the format of the return instruction can vary (*i.e.,* `ret`), with no parameter on the right hand side. Note that for the herein mentioned return edge details the format of the corresponding instruction is irrelevant. However, the pre-conditions and post-conditions needed for normal program execution are of importance. These are determined by the used calling convention and can slightly vary depending on the used architecture binary interface (ABI) (*e.g.,* Itanium ABI [122], Microsoft ABI [101], ARM ABI [9], *etc.*).

On one hand, the number of used and not used registers varies for each calling conventions, the stack usage and whether the stack is cleaned after or before a function returns. On the other hand, they usually do not vary w.r.t to the fact that each callee needs to return program control flow execution to the next address after the callsite.

Code reuse attacks inject different return addresses to violate the normal program control flow. This is achieved by manipulating the return addresses stored on the stack. These addresses will be used later by a return instruction to change the control flow of the program. Code reuse attacks use sequences of instructions located before a return instruction for their malicious intents. The instructions are used to prepare information for the next steps (gadgets) present in the code reuse gadget chain. Since an unprotected backward edge can target any address in the program the sequence can start at an arbitrary point (address) in a function. This results in a violation of the original CFG. By detecting such violation, it allows a backward edge protection tool to mitigate a CRA.

2.7.4 Virtual Table Hierarchy

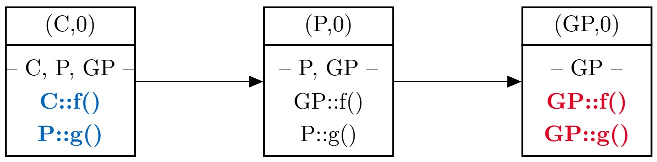


Figure 2.8: Virtual tables of a single C++ class hierarchy.

Figure 2.8 depicts three class-relevant terms required to better understand the remainder of the thesis. More specifically, Figure 2.8 depicts a single most derived type *C* with a parent *P* and grand-parent *GP*. An arrow indicates a class inheritance relation. With regard to a specific virtual callsite (which is used to call a virtual function contained in this class hierarchy), we introduce the following definitions.

Precise class. The precise class of a callsite is the least-derived type of which the object used at the callsite can be. Usually, the precise class is the static type of the variable used for the virtual call. However, depending on the compiler implementation, a stricter type can be inferred (see the Clang/LLVM [50] compiler framework for more implementation details). In our example, we assume that we have a virtual callsite which uses a variable of static type *C*. This static type is called the precise class of the callsite.

Base class. We define the base class as the class which provides the function implementation which is called when an object of precise type is used, *i.e.*, the object has a dynamic type of precise class. Therefore, the vtable entry used for the object dispatch is located in the vtable of precise class and per definition points to a function of base class. It follows that if the precise class itself implements (or overrides) the function used at the callsite, then the precise class and the base class of the callsite are the same.

Figure 2.8 depicts in blue shaded color the vtable entries used, in case the object at the callsite is of precise type (*C*). If the callsite dispatches function *f()*, then the base class for this callsite is *C*, since *C* overrides function *f()*. If instead the callsite dispatches function *g()*, then the base class for this callsite is *P*, because class *C* does not override function *g()* and instead uses the implementation provided by class *P*.

Root class. The root class is defined as the class first introducing the function (*i.e.,* the least derived class declaring the function). Note that this class might declare this function as `abstract` and not provide any implementation for it. In Figure 2.8, *GP*, depicted with red font color, is the root class of both functions *f()* and *g()*.

2.7.5 Shadow Stack Techniques

Shadow stack techniques can be used similarly to stack canaries, in order to protect against backward-edge program control flow violations, see Dang's [62] PhD thesis for more details. These techniques consist of complementing the program with additional code, which is able to check if the caller/callee calling convention is respected during runtime. This technique relies on building a second stack for each function stack located in the program. Runtime checks ensure that each function return address –that was put in the shadow stack before entering the called function– is popped from the stack before leaving the called function or before the stack frame was cleaned up by the program. Essentially, a shadow stack technique keeps track of all addresses that are pushed and popped on the stack and checks that the push-pop address pairs match. This way, the caller/callee calling convention is enforced. In addition, the program stack is not corrupted (polluted) by the attacker with fake addresses that are usually used to chain code reuse gadgets, as for example in return oriented programming (ROP) attacks.

While these techniques are effective in theory, they have received less acceptance. Only SafeStack [49] is in production, but was recently bypassed [94]. Further, their practicability and efficiency is questionable due to the fact that they rely on information hiding. We next list some of their limitations.

Hiding the Shadow Stack. The shadow stacks inside a program are typically hidden from the usual program execution through one level (ideally more) of indirection (*i.e.,* trampolines, segment register). These levels of indirection should guarantee that the attacker is not able to find the shadow stacks. This ensures that the attacker cannot write into them (these reside in writable memory). However, it is not yet demonstrated, that one level of indirection is sufficient to ensure that the shadow stack cannot be found (through information leaks) by a motivated and resourceful attacker. Since shadow stack implementations put the shadow stacks in writable memory, if they are found, they could be overwritten by the addresses that the attacker wants to use in their attack.

Binary Size Blow-up. Shadow stack-based techniques provide a separate shadow stack for each function, that is either instrumented inside the protected program or inside a library loaded along with the protected program. Due to the fact that the number of these additional shadow stacks can be high, the size of the program binary can grow rapidly. Some techniques considerably increase the size of the binary. Well-implemented (lazily allocated) shadow stacks will at most double the amount of memory used for stacks (this 2x factor is the same regardless of multi-threading, see [63] for more details). In addition, stacks tend to be very small compared to the heap. Therefore, these techniques might not be suitable for all types of program memory restrictive applications, such as certain embedded devices. Shadow stack techniques, which are, for example, based on hardware features such as Intel's CET (no known implementation at the time of writing this thesis) and compiler support, have negligible program binary size increase after hardening the executable.

Not returning calls. The C++ programing language, for which most of the shadow stack techniques were designed, provides some function calls, that do not return or respect the caller-callee function calling convention. These functions are: `longjump`, `tail calls`, *etc.* For these types of calls, the shadow stack techniques cannot be used, since these types of function calls do not return to the address next to the calling function. Finally, all tools enforcing shadow stack policies: (1) do not handle these types of calls due to complexity reasons, and (2) because these types of calls do not violate the caller-callee function calling convention, as these do not return at all.

Runtime overhead. As each function's return address has to be pushed, compared with the stack top value and popped from the shadow stack, the runtime overhead varies drastically from one shadow stack technique to another. Depending on the count of operations (instructions) which need to be performed (1-3), the shadow techniques have high performance overheads (around 10% see [63]), making them infeasible for deployment in production software. For these reasons, researchers are looking for approaches to do these operations with a minimal number of steps, such that the overhead is as low as possible and no memory leak is favored by these operations.

Limited Support for External Calls. Most C/C++ programs rely on third-party libraries and thus functions residing in these external libraries can be performed. For this reason, this type of external call needs to be protected as well. Most of the binary and compiler-based tools do not protect these shared libraries for a number of reasons: (1) binary-based tools usually cannot deal with functions having their address not taken, (2) binary-based tools often fail to analyze large binaries due to their complexity, and (3) the compiler-based tools opt to not recompile shared libraries due to increased analysis complexity, thus backward edges (also forward edges) remain unprotected. Note that BinCFI [290] (binary tool) could have easily added a shadow stack to these libraries, but omitted it, due to the resulting overhead. In other words, the backward edges contained in shared libraries are not protected and thus, the attack surface remains considerably high and the protection added to the program does not help much when all the needed gadgets reside in a shared library.

Emulated Shadow Stacks. Tools which approximate perfect shadow stack (*i.e.,* do not contain all caller-callee address pairs) techniques (*e.g.,* [266]) achieve only a coarse-grained precision w.r.t. the return addresses, which can be checked. On one hand, these techniques are optimized for performance. On the other hand, some of the return edges remain unprotected due to their imprecision. Furthermore, the checks of harvested addresses are slow due to: (1) the high volume of data flowing through the CPU, (2) the need to collect and analyze this data, and (3) the relative low speed of the continuous reads. As such, these techniques are mostly blind to attacks which use backward edges [239]. Therefore, other techniques which emulate a shadow stack more fine-grained or have the same precision as a perfect shadow stack implementation (shared library support is not supposed) are needed.

2.8 Type Inference in Program Binaries

In this Section, we provide the needed technical background to set the stage for the remainder of this thesis.

2.8.1 Exploiting C++ Object Dispatches

```
 1 class nsMultiplexInputStream final
 2   :public nsIMultiplexInputStream //A0
 3   ,public nsISeekableStream //A1
 4   ,public nsIIPCSerializableInputStream //A2
 5   ,public nsICloneableInputStream{ //A3
 6 nsTArray<nsCOMPtr<nsIInputStream>> mStreams;
 7 NS_IMETHODIMP nsMultiplexInputStream::Close(){
 8   MutexAutoLock lock(mLock);
 9   mStatus = NS_BASE_STREAM_CLOSED;
10   //set NS_OK flag
11   nsresult rv = NS_OK;
12   //get array length
13   uint32_t len = mStreams.Length();
14  //array-based main loop gadget (ML-G)
15  for (uint32_t i = 0; i<len; ++i){
16   //(0)hijacked object dispatch
17   nsresult rv2=mStreams[i]->Close();
18   if (NS_FAILED(rv2)) {
19       rv = rv2;
20   }
21  }
22   return rv;
23 }
```

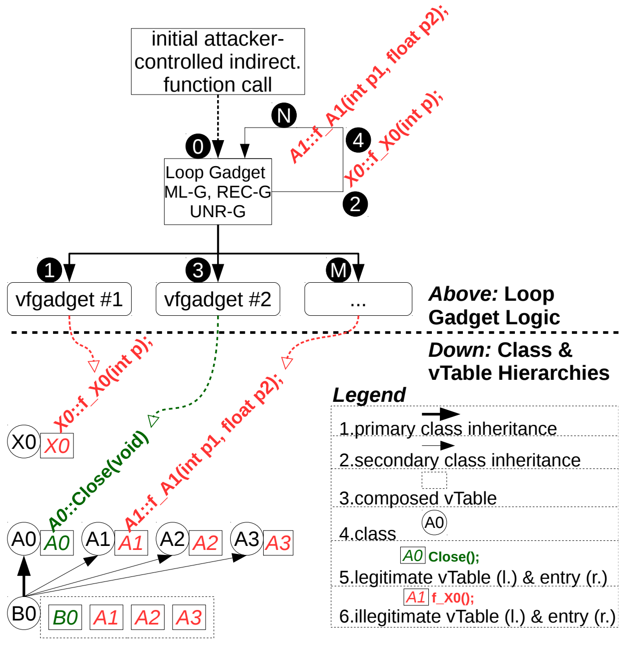

initial attacker-controlled indirect. function call
Loop Gadget ML-G, REC-G UNR-G
0
N
4
2
A1::f_A1(int p1, float p2);
X0::f_X0(int p);
1
3
M
vfgadget #1
vfgadget #2
...
Above: Loop Gadget Logic
Down: Class & vTable Hierarchies
X0::f_X0(int p);
A0::Close(void);
A1::f_A1(int p1, float p2);
X0
A0
A1
A2
A3
B0
Legend
1.primary class inheritance
2.secondary class inheritance
3.composed vTable
4.class
5.legitimate vTable (l.) & entry (r.)
Close();
6.illegitimate vTable (l.) & entry (r.)
f_X0();

Figure 2.9: COOP main loop gadget (ML-G) operation with the associated C++ code.

Figure 2.9 depicts a C++ code example (left) and how a COOP main-loop gadget (right) (*i.e.*, based either on ML-G (main-loop), REC-G (recursive) or UNR-G (unrolled) COOP gadgets, see [59] for more details) is used to sequentially call COOP gadgets by iterating through a loop (REC-G excluded) controlled by the attacker.

First, the object dispatch (see line 17 depicted in Figure 2.9) is exploited by the attacker in order to call different functions in the whole program by iterating on an array of fake objects previously inserted in the array through, for example, a buffer overflow. Second, in order to achieve this, the attacker previously exploits an existing program memory corruption (*e.g.*, buffer overflow), which is further used to corrupt an object dispatch, ➊, by inserting fake objects into the array and by changing the number of initial loop iterations. Next, she invokes gadgets ➊ and ❸ up to Ⓜ, through the calls, ❷ and ❹ up to Ⓝ, contained in the loop. As it can be observed in Figure 2.9, the attacker can invoke from the same callsite legitimate functions (in total Ⓝ) residing in the virtual table (vTable) inheritance path (*i.e.,* at the time of writing this thesis this type of information is particularly hard to recuperate from program binaries) for this particular callsite, indicated with green color vTable entries. However, a real COOP attack invokes illegitimate vTable entries residing in the entire initial program class hierarchy (or the extended one) with little or no relationship to the initial callsite, indicated with red-color vTable entries. Third, this way different addresses contained in the program (1) (vTable) hierarchy (contains only virtual members), (2) class hierarchy (contains both virtual and non-virtual members) and (or) the whole program address space can be called. For example, the attacker can call any entry in the: (1) class hierarchy of the whole program, (2) class hierarchy containing only legitimate targets for this callsite, (3) virtual table hierarchy of the whole program, (4) virtual table hierarchy containing only legitimate targets for this callsite, (5) virtual table hierarchy and class hierarchy containing only legitimate targets for this callsite, and (6) virtual table hierarchy and class hierarchy of the whole program. Finally, because there are no intrinsic language semantics—such as object cast checks—in the C++ programming language for object dispatches, the loop gadget indicated in Figure 2.9 can be used without constraint to call any possible entry in the whole program. Thus, making any program address the start of a potential usable gadget.

2.8.2 Type-Inference on Executables

Recovering variable types from executable programs is generally considered difficult for two main reasons. First, the quality of the disassembly can vary considerably from one used underlying binary analysis framework to another and w.r.t. the compiler flags which were used to compile the binary. Note that production binaries can be more or less stripped (*i.e.,* RTTI or other debugging symbols may or may not be available *etc.*) from useful information, which can be used during a type-recovering analysis. τCFI is based on DynInst and the quality of the executable disassembly is sufficient for our needs. In contrast to other approaches, the register width-based type recuperation of τCFI is based on a relatively simple analysis compared to other tools and provides similar results. For a more comprehensive review on the capabilities of DynInst and other tools, we advise the reader to review Andriesse *et al.* [5]. Second, if the type inference analysis requires alias analysis, it is well known that alias analysis in binaries is undecidable [229] in theory and intractable in practice [202]. Further, there are several highly promising tools such as: Rewards [147], BAP [28], SmartDec [81], and Divine [14]. These tools try more or less successfully to recover (or infer) type information from binary programs with different goals. Typical goals are: (1) full program reconstruction (*i.e.,* binary to code conversion, reversing, *etc.*), (2) checking for buffer overflows, and (3) checking for integer overflows and other types of memory corruptions. For a comprehensive review of type inference recovering tools in the context of binaries, we suggest consulting Caballero *et al.* [33]. Finally, it is interesting to note that the code from only a few of the tools mentioned in the previous review are actually available as open source.

2.8.3 Security Implications of Program Indirect Transfers

Indirect Forward-Edge Transfer. Illegal forward-edge indirect calls may result from a virtual pointer (vPointer) corruption. A vPointer corruption is not a vulnerability but rather a capability, which can be the result of a spatial or temporal memory corruption triggered by: (1) bad-casting [142] of `C++` objects, (2) buffer overflow in a buffer adjacent to a `C++` object, or (3) a use-after-free condition [238]. A vPointer corruption can be exploited in several ways. A manipulated vPointer can be exploited to make it point to any existing or added program virtual table entry or to a fake virtual table added by the attacker. For example, an attacker can use the corruption to hijack the control flow of the program and start a COOP attack [238]. vPointer corruptions are a real security threat that can be exploited in many ways as for example if there is a memory corruption (*e.g.,* buffer overflow, use-after-free condition), which is adjacent in memory to the `C++` object. Consequently, each memory corruption, which can be used to reach the memory layout of an object (*e.g.,* object type confusion) can be potentially used to change the program control flow.

Indirect Backward-Edge Transfers. Program backward edges (*i.e.,* `jump`, `ret`, *etc.*) can be corrupted to assemble gadget chains such as follows. (1) No CFI protection technique was applied: in this case, the binary is not protected by any CFI policy. Obviously, the attacker can then hijack backward edges to *jump* virtually anywhere in the binary in order to chain gadgets together. (2) Coarse-grained CFI protected scenarios: In this scenario, if the attacker is aware of what addresses are protected, the attacker may deviate the application flow to legitimate locations in order to link gadgets together. (3) Fine-grained CFI protection scenarios: in this case, the legitimate target set is stricter than in (2). But, assuming that the attacker knows which addresses are protected and which are not, she may be able to call legitimate targets through control flow bending. (4) Fully precise CFI protected scenarios (*i.e.,* SafeStack [135] based): in this scenario, the legitimate target set is stricter than in (3). Even though we have a one-to-one mapping between calltargets and legitimate return sites, the attacker could use this one-to-one mapping to assemble gadget chains if at the legitimate calltarget return site there is a useful gadget [36].

Interestingly to notice in this context is that through: (1) memory layout analysis (through highly configurable compiler tool chains) of source-code-based locations which are highly prone to memory corruptions such as declarations and uses of buffers, integers or pointer deallocations one can obtain the internal machine code layout representation; (2) analysis of a code corruption which is adjacent (based on (1)) to a `C++` object based on application class hierarchy, the virtual table hierarchy and each location in source code where an object is declared and used (*e.g.,* modern compiler tool chains can spill out this information for free), one can derive an analysis which can determine—up to a certain extent—if a memory corruption can influence (*e.g.,* is adjacent to) a `C++` object.

Finally, tools based on these two concepts (*i.e.,* (1) and (2)) can be used by attackers, *e.g.,* to find new vulnerabilities, and by defenders to harden the source code only at the places which are most exposed to such vulnerabilities (*i.e.,* targeted security hardening).

2.8.4 Shadow Stack Techniques

In this Section, we present details on how shadow stack techniques can by bypassed in general and why we need an alternative solution for protecting backward edges.

Shadow stack techniques are based on the assumption that these are hard to be found by an attacker and that these have to remain writable during runtime as addresses are pushed and popped during runtime. According to Goktas *et al.* [94] there are at least four attack vectors (namely; neglected pointers, thread spraying, allocation oracles, and guessing oracles) which can be used independently to bypass a shadow stack implementation. More broadly, these attack vectors show that all entropy based hiding techniques can be bypassed by a motivated attacker.

Simulated shadow stacks. The original CFI implementation of Abadi *et al.* [1] protects backward edges by simulating a shadow stack *i.e.,* without building new shadow stacks. Abadi *et al.* present two approaches: (1) based on pre-fetching (*i.e.,* using `prefetchnta` instruction) a label into a register before entering/calling a function and comparing with this label before leaving the function, and (2) based on hardware registers (segment register `gs:[0h]`) in which the return address of a function is loaded before entering/calling a function and comparing it before leaving the function with the address where the function returns. More specifically, each program function can return to only an address which was previously loaded into an register by using hardware segments.

The segment register `gs` always points to the memory segment that holds the shadow call stack and which has been created to be isolated and disjoint from other accessible memory segments. On Windows, `gs` is unused in application code; therefore, without limitation, CFI verification can statically preclude its use outside this instrumentation code. As shown in the figure, the instrumentation code maintains (in memory location `gs:[0h]`) an offset into this segment that always points to the top of the stack. The use of the protected shadow call stack implies that each return goes to the correct destination, so no ID-checks are required on returns in this instrumentation code. Approach (1) is in our opinion not affected by these attack vectors but in case the attacker knows the register in which the label was stored he can rewrite the a different label. Approach (2) is affected by the attack vectors presented above since the segments on which it relies, where the return address is stored (see bottom part of Fig. 7 in [1]) can be found and overwritten by an attacker and also the return address in the stack can be overwritten by the attacker and as such he may return were ever he wants. To this end, this proprietary implementation is not open source and (1) relies on a special instruction (*e.g.,* `prefetchnta` only for 32-bit available) and (2) relies on hardware segment registers (i.e., `gs:[0h]`) which are not available on all architectures. Further, the overhead of these techniques is on average more than 15% when protecting both backward and forward edges.

As response compiler based solutions (*i.e.,* Clang's SafeStack [49] and in GCC's shadow stack implementation [86]) where proposed in the last years. These techniques do not rely on exotic instructions, program segments, and hardware registers. These approaches provide a second writable shadow stack for each program stack. In essence these techniques work by pushing the return address of each function into a secondary writable shadow stack before entering/calling a function and popping this address before leaving the function. In case these addresses match, then program execution is continued else program execution is terminated. The accesses to the shadow stacks are made trough pointers and the main assumption is here that the shadow stack are hard to find.

Real shadow stacks. The Clang's SafeStack and GCC's shadow stack implementation relies on splitting each program stack in two stacks on which addresses are pushed and popped before entering a function and before leaving it, respectively. It turns out that neglected pointers, thread spraying and allocation oracles techniques can be all independently used to locate the shadow stacks. These work under the assumption that there is a memory corruption, the attacker has read/write primitives and that heap and module data is disclosed. As such, the shadow stack can be located and overwritten by the attacker which may return to any address in the program he desires.

Shadow stacks downsides. In terms of security, shadow stack protection is stronger than the approach of reversing forward edges when assuming that the shadow stack's integrity is protected. However, shadow stack's integrity has to be protected by a separate mechanism. Randomization is not a good idea here, as shown by Goktas *et al.* [94]. Software-based fault isolation would guarantee its integrity, but that comes with additional overhead. The shadow stack's implementation has a higher overhead than the approach of reversing forward edges. This would be a different situation with hardware supported shadow stacks however. The shadow stack has trouble of being compatible with common software practices such as `longjumps` or `setjumps`, exceptions, continuations, *etc.* that perform unconventional control flows. It takes extra effort to make it work at least.

However, our motivation is to provide a new technique for protecting backward edges in program binaries which does not rely on information hiding or any special instructions and is not affected by arbitrary writes since the labels are in write protected code located. Thus, our approach is not affected by the four attack vectors [94]. Our approach enforces N addresses per return site while a shadow stack enforces one address per return site.

2.8.5 Polymorphism in C++ Programs

Polymorphism along with inheritance and encapsulation are the most used modern object-oriented concepts in `C++`. In `C++`, polymorphism allows accessing different types of objects through a common base class. A pointer of the type of the base object can be used to point to object(s) which are derived from the base class. In `C++`, there are several types of polymorphism: (a) compile-time (or static, usually is implemented with templates), (b) runtime (dynamic, is implemented with inheritance and virtual functions), (c) ad-hoc (*e.g.,* if the range of actual types that can be used is finite and the combinations must be individually specified prior to use), and (d) parametric (*e.g.,* if code is written without mention of any specific type and thus can be used transparently with any number of new types). The first two are implemented through early and late binding, respectively. In `C++`, overloading concepts fall under the category of *c)* and virtual functions, templates or parametric classes fall under the category of pure polymorphism. However, `C++` provides polymorphism through: (i) virtual functions, (ii) function name overloading, and (iii) operator overloading. In this thesis, we are concerned with dynamic polymorphism, based on virtual functions (see ISO/IEC N3690 [129]), because it can be exploited to call: (x) illegitimate virtual table entries (not) contained in the class hierarchy by potentially varying the number of parameters and types, (y) legitimate virtual table entries (not) contained in the class hierarchy by potentially varying the number of parameters and types, and (z) fake virtual tables entries not contained in the class hierarchy by potentially varying the number of parameters and types. By legitimate and illegitimate virtual table entries we mean those virtual table entries which for a single indirect callsite lie in the virtual table hierarchy. More precisely, a virtual table entry is legitimate for a callsite if from the callsite to the virtual table containing the entry there is an inheritance path (see [106]). Virtual functions have several uses and issues associated, but for the scope of this thesis we will look at the indirect callsites which are exploited by calling illegitimate virtual table entries (*i.e.,* functions) with varying number and type of parameters, see (x) above). More precisely, (1) load-time enforcement: as calling each indirect callsite (*i.e.,* callee) requires a fixed number of parameters which are passed each time the caller is calling, we enforce a fine-grained CFI policy by statically determining the number and types of all function parameter that belong to an indirect callsite, and (2) runtime verification: as differentiating during runtime legitimate from illegitimate indirect caller/callee pairs requires parameter type and parameter number, we insert before each indirect callsite a check used for determining during runtime if the caller matches with the callee based on certain CFI policies.

2.8.6 Real COOP Attack Example

The bug CVE-2014-3176 was exploited by Crane *et al.* [59] in order to perform a COOP attack, on the Google Chromium Web browser. The details of this attack are highly complex involving not properly handled interaction of browser extensions between the IPC, the sync API, and Google V8 engine and for this reason we briefly present a better documented COOP exploit which is in principle similar with this attack.

Figure 2.10 depicts[1] a Turing complete COOP attack [238] which was used to attack the Mozilla Firefox web browser. Essentially, by exploiting an existing buffer overflow bug the attacker was able to call into existing virtual table entries by having a main loop gadget at his disposal. Next we present the steps needed in order to perform this attack.

First, the attacker uses the `C++` class `nsMultiplexInputStream` (see Figure 2.10) which contains a main loop gadget (ML-G) inside the `nsMultiplexInputStream::Close(void)` function in order to perform indirect calls by dispatching calls on the fake objects contained in the array. The objects contained in the array during normal execution are of `nsInputStream` type and each of the objects will call the `Close(void)` function in order to close each of the previously opened streams.

[1]The class inheritance hierarchy of the classes involved in the COOP attack against the Firefox browser. Red letters indicate forbidden virtual table entries and green letters indicate allowed virtual table entries for the given indirect callsite contained in the main loop gadget.

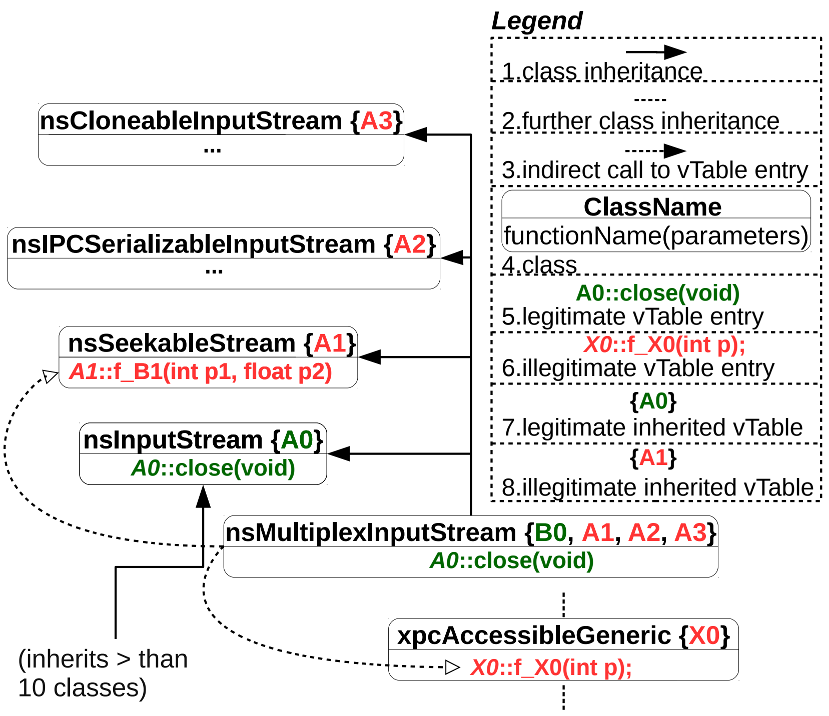

Legend
1.class inheritance
2.further class inheritance
3.indirect call to vTable entry
ClassName
functionName(parameters)
4.class
A0::close(void)
5.legitimate vTable entry
X0::f_X0(int p);
6.illegitimate vTable entry
{A0}
7.legitimate inherited vTable
{A1}
8.illegitimate inherited vTable
nsCloneableInputStream {A3}
...
nsIPCSerializableInputStream {A2}
...
nsSeekableStream {A1}
A1::f_B1(int p1, float p2)
nsInputStream {A0}
A0::close(void)
nsMultiplexInputStream {B0, A1, A2, A3}
A0::close(void)
xpcAccessibleGeneric {X0}
X0::f_X0(int p);
(inherits > than
10 classes)

Figure 2.10: Class hierarchy of classes used in the COOP attack.

Second, for performing the COOP attack, the attacker crafts a C++ program containing an array buffer holding six fake objects. These fake objects can call inside (and outside) the initial class and virtual table hierarchies with no constraints. During the attack a buffer is created in order to hold the fake objects. The crafted buffer will be used instead of the real code in order to call different functions available in the program code. For example, the attacker calls a function contained in the class `xpcAccessibleGeneric` which is not in the class hierarchy or virtual table hierarchy of the initially intended type of objects used inside the array. Moreover, the header file of this class (`xpcAccessibleGeneric`) is not included in the class `nsMultiplexInputStream`.

Third, in total, six fake objects are used to call into functions residing in unrelated class hierarchies with varying number of parameters and return types. The final goal of this attack is to prepare the program memory such that a Unix shell can be opened at the end of this attack.

Lastly, this example illustrates why detecting vPointer corruptions is not trivial for real-world applications. As depicted in Figure 2.10, the class `nsInputStream` has 11 classes which inherit directly or indirectly from this class. The classes `nsSeekableStream`, `nsIPCSerializable-InputStream` and `nsCloneableInputStream` provide additional inherited virtual tables which represent illegitimate calltargets for the initial `nsInputStream` objects and legitimate calltargets for the six fake objects which were added during the attack. Furthermore, declaration and usage of the objects can be widely spread out in the source code. This makes detection of the object types (*i.e.,* base class), range of virtual tables (*i.e.,* longest virtual table inheritance path for a particular callsite) and parameter types of the virtual table entries (*i.e.,* functions) in which it is allowed to call a trivial task for source code applications, but a hard task when one wants to apply similar security policies (*e.g.,* which rely on parameter types of virtual table entries) to binary executables.

2.8.7 Mitigation of Forward-Edge Based Attacks

Binary based tools. TypeArmor [268] has around 3% runtime overhead and enforces a CFI-policy based on runtime checking of caller-callee pairs which relies on runtime function parameter count matching. Compared to TYPESHIELD, this tool does not use function parameter types and assumes that a backward-edge protection is in place. VCI [78] and Marx [218] are both based on approximated program class hierarchies: (1) do not recover the root class of the hierarchy, and (2) the edges between the classes are not oriented. Thus, both tools enforce for each callsite the same virtual table entry (*i.e.,* index based) contained in one recovered class hierarchy denoted by father-child relationships between the recovered vtables. Finally, both tools use up to six heuristics and simplifying assumptions in order to make the problem of program class hierarchy reconstruction tractable.

Source-code-based tools. To the best of our knowledge, only the Clang-CFI [149] and IFC-C/VTV [261] (up to 8.7% performance overhead) compiler based tools are deployed in practice and can be used to check legitimate from illegitimate indirect forward-edge calls during runtime by checking if virtual object pointers comply with the program class hierarchy inheritance relationships. Furthermore, ShrinkWrap [106] is a GCC compiler based tool which further reduces the legitimate virtual table ranges for a given indirect callsite (*i.e.,* object dispatch) through precise analysis of the program class hierarchy and virtual table hierarchy. Evaluation results show similar performance overhead results as the previous mentioned tools but with more precision with respect to legitimate virtual table entries (calltargets) per callsite. We noticed by analyzing the previous research results that the overhead incurred by these security checks can be very high due to the fact that for each callsite many range checks have to be performed during runtime. Therefore, in our opinion, despite its security benefit these types of checks are most likely not applicable to software where performance is key.

Other types of tools. Other highly promising source-code-based tools (albeit also not deployed in practice) which can overcome some of the drawbacks of the previously described tools are as follows. Bounov *et al.* [24] presented a tool ($\approx$1% runtime overhead) for indirect forward-edge callsite checking based on virtual table layout interleaving. The tool has better performance than VTV and better precision with respect to allowed virtual tables per indirect callsite. Its precision (selecting legitimate virtual tables for each callsite) compared to ShrinkWrap is lower since it does not consider virtual table inheritance paths. vTrust [286] (average runtime overhead 2.2%) enforces two layers of defense (virtual function type enforcement and virtual table pointer sanitization) against virtual table corruption, injection and reuse.

2.8.8 Mitigation of Backward-Edge Based Attacks

According to one of the currently most comprehensive surveys by Burow *et al.* [32] assessing backward edge protection techniques and runtime overhead comparisons, tools can be distinguished into providing low, medium, and high levels of protection w.r.t. backward-edges. Further, this survey provides runtime overhead comparisons. classifies the backward-edge protection techniques in binary-based, source-code-based, and other types (*i.e.,* with HW support, *etc.*).

Binary based tools. Burow *et al.* provide the following insights w.r.t. binary-based tools. The original CFI implementation from Abadi *et al.* [1] as well as MoCFI [156], kBouncer [216], CCFIR [289], bin-CFI [290], Reins [276], O-CFI [187], PathArmor [266], LockDown [221] mostly suffer from imprecision (high number of reused labels); have low runtime efficiency; and some of the tools do not support shared libraries.

Source-code-based tools. SafeStack [49], Hypersafe [274], CF-Locking [19], MIP [209], CF-Restrictor [224], KCoFi [61], RockJIT [211], CCFI [160], Kernel CFI [90], MCFI [210], piCFI [212] relatively high precision w.r.t. enforced address return set, and all do not support shared libraries (except MCFI), have high coverage (almost all backward edges are protected).

Other types of tools. ROPecker [44], HW-asst. CFI [66], CFIGuard [284] are mostly relying on HW features which were not specifically built for protecting backward edges and for this reason the tools are not runtime effective and their precision is rather low and mostly not effective against state-of-the-art code reuse attacks.

Chapter 3

Related Work

In this Chapter, we present related work with respect to detection of integer overflows in Section 3.1, and in Section 3.2, we present tools usable for integer overflow classification and repair generation, while in Section 3.3, we present related work to buffer overflow repair generation. In Section 3.4, we introduce some techniques for detection of object type confusions and in Section 3.5, we talk about static and dynamic gadget discovery as well as existing metrics for assessing CFI defenses. In Section 3.6, we highlight some of the tools usable for program CFG backward-edge runtime protection. Further, in Section 3.7, we describe how backward and forward edges based attacks can be mitigated as well as mitigation of advanced and not advance CRAs. Finally, note that parts of this Chapter have already been published by Muntean *et al.* [195, 196, 197, 198, 199, 200, 201].

3.1 Detecting Integer Overflows

Integer-related problems detecting research focuses primarily on integer overflow vulnerabilities, either employing dynamic or static code approaches [54] which run on binary or source code. The static approaches are mostly based on an analysis framework and added run-time checks at certain interesting points in code (*e.g.,* assignments, $x := expr.$) located on satisfiable program paths.

3.1.1 Static Analysis Tools

The static analysis tool UQBTng [280] decompiles the binary files and then uses model checking based on CBMC [53] to detect integer overflows. IntScope [272] first transforms the analyzed binaries into an intermediary representation (IR) and based on symbolic execution and taint analysis it checks for integer overflows. These two approaches operate on binary files and cannot figure out the original variables data types since they get lost during the compilation process, whereas our approach can use the original variables data types from source code. ARCHERR [45] can examine million of lines of source code but can not deal with string operations and has on average 35% of false positives, whereas our approach has a lower false positives rate. Microsoft's PREfast [166] is used during source code compile time and relies on source code annotations which is integrated in the Microsoft VS IDE and are provided in advance. Our approach does not require costly annotations. On the other hand, we think that the annotations represent an useful source of information which is typically not available during static analysis. Microsoft's PreFix [193] is based on the Z3 [166] solver, runs on large legacy C/C++ source code repositories in order to find a wide range of problems related to integers and uses a ranking mechanism to filter out false positives. It is based on a ranking mechanism used to filter out false positives and suffers from imprecision because it does not use the already available annotations contained in the source code as the authors acknowledge. INTDETECT compared to PreFix focuses integer related problems. It is based on a ranking mechanism used to filter out false positives and suffers from imprecision because of not using the already available annotations contained in the source code as the authors acknowledge. INTDETECT compared to PreFix focuses on only one type of integer related problems (integer overflows) by using `C` function models whereas the PreFix tool does not use function models. Ceesay and colleagues [38] added type qualifiers to detect integer overflow problems. Their approach relies on expensive system extensions which are linked into the compiler and are used to check the code for integer errors. Their approach requires user annotation whereas our approach does not require annotations. Ashcraft *et al.* [10] and Sarkar *et al.* [235] used bounds checking and taint analysis to see if untrusted values are used in trusted sinks. These two approaches are based on insensitive information flow whereas our approach is context sensitive.

3.1.2 Dynamic Analysis Tools

The dynamic analysis tools are used to detect integer related problems: RICH [29], BRICK [42], SmartFuzz [188], SAGE [93] and IOC [73]. RICH [29] instruments programs to detect safe and unsafe operations based on well-known sub-typing theory. BRICK [42] detects integer overflows in compiled executables using a modified Valgrind [203] version. Its accuracy and efficiency depend on the test input used to exercise the instrumentation. It is either slow ($50\times$ slowdown) or has many false positives. SmartFuzz [188] is also based on Valgrind, but it uses dynamic test generation techniques to generate inputs, leading to good test coverage. SAGE [93] uses dynamic test generation, but it targets fewer integer problems than SmartFuzz. IOC [73] is an integer overflow and underflow problems detection tool integrated with the Clang compiler. Compared with INTDETECT, these tools can neither exercise all bugs due to dynamically generated test cases nor full path coverage can be achieved.

3.2 Repairing Integer Overflows

There is a large body of research work focusing on integer overflow *detection*: ARCHER [45], UQBTng [280], PREfast [56], Rich [27], SAGE [93], CBMC [53], IntScope[272], Brick [42], IntFinder [41], SmartFuzz [188], PREfix [193], IntPatch [288], IOC [73], IntFlow [225], SoupInt [270], SIFT [154], TAP [249], Diode [248], Indio [292], Zhang *et al.* [285], and IntEQ [255]. In contrast, only few approaches focus explicitly on integer overflow *repairs*: CIntFix [43], SoupInt [154], CodePhage [247], TAP [249], and SIFT [154]. Out of these approaches, only TAP [249] (technical report) and SIFT [154] first explicitly detect the integer overflow and then repair it. The other tools—which do not first confirm the bug existence—mostly blindly change the code in all error-prone locations in the hope to avoid integer related problems. For example: (1) CIntFix [43] utilizes integers of infinite size with two's complement encoding in place of original bounded integers, and (2) AIC/CIT/RAO [54] relies on three code transformations: add integer cast (AIC), replace arithmetic operator (RAO), and change integer type (CIT) to change the program in order to avoid integer overflows. However, in most cases these tools are triggering high runtime overhead and have a considerable likelihood of changing program behavior.

TAP [249] first detect the integer overflow and than propose a repair for it. TAP is similar to IntRepair w.r.t. the fact that both tools first detect an integer overflow and next they generate a code repair which removes the integer overflow. TAP utilizes the integer overflow discovery algorithm from DIODE [248]. Another improvement of IntRepair over TAP is that a validation mechanism is provided by IntRepair to check if the integer overflow is indeed removed from the program.

SIFT [154] first detects the integer overflow (bug detection is based on Diode [248]) and then generates an input filter to eliminate the bug at the binary level. Sift is a static input filter generation tool, which inserts input filters in the program binary for which the source code was previously manually annotated with source code annotations. While currently the only sound integer overflow repair tool, Sift relies on tedious user source code annotations that are not always available for a source code program or require a considerable and tedious annotation overhead. Upper bound source code annotations for loops are needed when the analyzed expression depends on a number of values that is not finite. Further, not all types of integer overflow relevant sites are supported (*i.e.,* only memory allocations and block memory copy sites). For some types of applications (*i.e.,* web servers *etc.*) with no available input format specification (*i.e.,* no image or video files) Sift cannot be applied. Sift relies on tedious user source code annotations and cannot guarantee that no unwanted program behavior is introduced since the filters may remove only the integer overflows which these cover. Sift does not support multi-precision integer overflow repairs, and annotated stub standard `C` library functions need to be provided upfront for functions that influence the computed symbolic condition (if not provided, the filter will not be generated).

In the above-mentioned approaches, fault localization is performed before patch generation. In contrast, we combine fault localization and patch generation, and as a result, we obtain the capability of generating precise patches which remove the bug for various program inputs (bug detection is not program input independent) and at the exact location where the bug was detected upfront. For this purpose, we use SMT solving for bug localization. However, unlike IntRepair, SMT solving is not used for repair synthesis by TAP or Sift. As such, their bug removal process is bound to a limited number of test inputs to confirm that the bug was removed after the repair was inserted, which does not guarantee that the bug was really removed for all possible program inputs.

Tool	Venue	Top Tier	s. s. d.	s. ¬ s. d.	d. s. d.	d. ¬ s. d.	SMT solv.	¬ SMT solv.	source code	binary code	intermediate	C support	C++ support	overflow	underflow	signedness	truncation	sound	complete	benign	exploitable	annotations	fuzzing	mutating	detect (d)	repair (r)	(d&r)
ARCHERR [45]	ESORICS'04	✓				✓		✓	✓			✓		✓	✓							✓			✓		
UQBTng [280]	22C3'05					✓		✓		✓				✓							✓	✓			✓		
PREfast [166]	MSR-TR'10			✓				✓	✓			✓	✓	✓								✓			✓		
Rich [29]	NDSS'07	✓				✓		✓	✓			✓		✓	✓	✓	✓			✓	✓				✓		
SAGE [93]	NDSS'08	✓			✓		✓			✓				✓	✓	✓	✓						✓		✓		
IntScope [272]	NDSS'09	✓	✓				✓				✓			✓										✓	✓		
Brick [42]	ARES'09	✓				✓		✓			✓			✓	✓	✓	✓								✓		
IntFinder [41]	ICICS'09					✓		✓		✓				✓	✓	✓	✓								✓		
SmartFuzz [188]	Usenix Sec.'09	✓			✓		✓	✓			✓			✓	✓	✓	✓						✓		✓		
PREfix [193]	MSR-TR'09		✓				✓	✓				✓	✓	✓				✓							✓		
IntPatch [288]	ESORICS'10	✓		✓					✓			✓	✓	✓				✓							✓		
IOC [73]	ICSE'12	✓				✓		✓	✓			✓	✓	✓	✓	✓	✓								✓		
IntFlow [225]	ACSAC'14	✓				✓		✓	✓			✓	✓	✓	✓	✓	✓								✓		
SoupInt [270]	DIMVA'14				✓		✓			✓				✓						✓	✓					✓	
SIFT [154]	ACM SN'14			✓				✓	✓			✓		✓				✓				✓				✓	
TAP [249]	MIT-TR'14		✓				✓		✓	✓		✓	✓	✓							✓				✓	✓	✓
Diode [248]	ACM SN'15		✓				✓			✓				✓											✓		
Indio [292]	RAID'14	✓	✓				✓			✓				✓							✓				✓		
Zhang *et al.* [285]	SNPD'16				✓		✓			✓				✓											✓		
IntEQ [255]	ICSE'16		✓				✓		✓			✓	✓	✓	✓	✓	✓			✓	✓				✓		
CIntFix [43]	COMPSAC'16					✓		✓	✓			✓		✓	✓	✓	✓					✓				✓	
INTREPAIR [197]	-		✓						✓			✓		✓	✓			✓							✓	✓	✓

Table 3.1: Integer overflow detection and repair features. Entries are arranged by year.

Table 3.1 summarizes a string of tools used for integer overflow detection and repair, and the underlying techniques on which these tools are based. In Table 3.1 the ✓symbol means addressed and the abbreviations have the following meaning: static symbolic detection (s. s. d.), dynamic symbolic detection (d. s. d), and ¬ logical not. We deliberately excluded from Table 3.1 commercial static symbolic execution tools (*e.g.,* CodeSonar; see [245] for more details), which scale to programs having millions LOC and which can efficiently detect integer overflows, since their internals are mostly unknown. Further, we have added a single tool based on bounded model checking (BMC), namely CBMC [53], which can be used to check pointer safety, array bounds, and user-provided assertions. While BMC can be used in principle to detect integer overflows, we consider this line of research orthogonal to our research.

Further, it is interesting to note that most of the tools presented in Table 3.1 are either only used for integer overflow detection or if they are used for code repair then they do not consider detecting first the integer overflow, and none of the integer overflow repair tools are complete for analysis trade-off reasons. Finally, integer overflow has threatened software programs for decades, and many efforts have been made to address this problem. Next we will present the most representative work.

3.2.1 Detecting Integer Overflows

Library Support. Safe integer libraries such as SafeInt [169] and IntegerLib [240] are widely used during runtime. These approaches impose on the programmers that they rewrite existing code to use safe integer operations. IntRepair in contrast detects and repairs integer overflows without any assistance from developers to rewrite code.

Runtime Checks. RICH [27], IOC [73], IntPatch [288], RA [222] and IntTracker [254] utilize program instrumentation to insert security checks around integer arithmetic operations of interests in order to detect integer overflows during runtime. One major drawback of this approach is that these tools insert runtime checks conservatively such that they often incur a high runtime overhead. In contrast, IntRepair inserts checks only when it previously found an integer overflow. It therefore imposes a lower performance overhead.

Static Integer Overflow Detection. KINT [273] utilizes taint analysis and integer range analysis to generate test cases which can trigger integer overflows in Linux kernel. It requires optionally procedure specifications from the programmer to characterize parameter value ranges and reports many false positives. SIFT [154] uses a sound static program analysis to collect constraints from program entries to integer arithmetic operations which can flow into memory allocation operations and block copy operations, to generate input filters which can decline inputs that may trigger overflow errors. Similarly, DIODE [248] relies on targeted site identification and goal-directed conditional branch enforcement to discover integer overflows. Note that KINT [273], SIFT [154] and DIODE [248] can only handle integer overflows which can influence memory allocations or block copy operations, *i.e.,* IO2BO, whereas IntRepair can deal with all types of integer overflows.

Dynamic Integer Overflow Detection Taking a different approach, fuzzing-based software testing and concolic testing are two widely used dynamic techniques to detection integer overflows. Random fuzzing [92] is widely used by researchers to expose program errors, not only integer overflows. Due to the fact that the new inputs it generates often fail certain sanity checks, random fuzzing is relatively ineffective. Directed fuzzing such as BuzzFuzz [85] and TaintScope [271] first uses dynamic taint tracking to identify input bytes that can influence values at critical program sites such as memory allocation sites and system calls, and then mutates these particular bytes so as to expose errors which reside deeply inside programs. Since the size of fuzzed input is reduced, directed fuzzing is much more effective than random fuzzing.

Concolic Testing. [46, 158, 241] is a newer alternative than random and directed fuzzing. These tools execute programs both concretely and symbolically on a seed input until an interesting program expression is reached. Although successful in many scenarios [34], concolic testing faces several challenges [228, 35]. Specifically, the resulted constraint systems for deeper program paths get very complex and thus beyond the capabilities of current SMT solvers. SmartFuzz [188] is a concolic testing tool which can detect integer overflows, non-value-preserving width conversions, and potentially dangerous signed/unsigned conversions. However, it is limited by deep program paths and blocking checks. Dowser [108] is a fuzzer that combines taint tracking, program analysis, and symbolic execution to find buffer overflows. Note that compared to IntRepair, these dynamic techniques need a set of seed inputs and may suffer from a heavy runtime overhead.

3.2.2 Classifying Integer Overflow

As discussed in RICH [27], SIFT [154] and IntEQ [255], integer overflow might be introduced into the programs intendedly by programmers in order to accomplish benign purposes, such as hashing computation and random number generation. These benign integer overflows are not harmful and it is not needed to fix them.

IntFlow [225] first utilizes static data flow analysis to find safe integer arithmetic operations which originates from trusted sources such as constants and configuration files, or which finally propagates into security-unrelated program sites such as debug information printing, and then inserts security checks for those integer arithmetic operations besides these safe ones to gain runtime protection. IntFlow could exclude the reports for benign integer overflows to some extent. However, there still exists a lot of benign integer overflows which are along the paths from untrusted sources to security-related program sites.

A further improvement is made by IntEQ [255], which uses equivalence checking across multiple precisions to recognize benign integer overflows from harmful ones. IntEQ determines whether an integer overflow is benign by comparing the effects of an overflowed integer arithmetic operation in the actual world with limited precision and the same operation in the ideal world with sufficient precision to evade the integer overflow. However, IntEQ still fails to recognize 20% benign integer overflows according to its experiments.

Hence, it is not easy to identify all the benign integer overflows automatically since benign integer overflows are essentially the programmers' intentions and it is difficult to formalize and infer what programmers intend to do. INTREPAIR tries to mitigate this problem by letting programmers make the final decisions. When an integer overflow is detected by INTREPAIR, whether to further repair it or not is determined by programmers. If needed, a proper overflow repair pattern is selected and patched automatically.

3.2.3 Repairing Integer Overflows

SoupInt [270] is an off-line system that can diagnose whether an exploit is caused by an integer overflow and generate emergency patches at proper locations to fix it. Note that SoupInt works for binary code whereas INTREPAIR works for source code.

CIntFix [43] utilizes integers of infinite size with two's complement encoding in place of original bounded integers to provide runtime protection for `C` source code functions. The analysis performed by CIntFix is syntax-directed and rule-based, which avoids sophisticated and imprecise analysis. However, its analysis cannot scale to large programs and CIntFix can miss to repair some integer overflows as well. From the total of 2052 programs in the same test suite, CIntFix could repair only 1938, whereas INTREPAIR could fix all of them. Also, the source code expansion is above 25%, which is almost 25 times more than that of INTREPAIR. The runtime on the repaired programs is 16%, whereas INTREPAIR's slowdown on the same programs is about 1%.

CodePhage [247] is an automatic patching tool, transferring the patch for an integer overflow from donor applications to recipient applications based on the assumption that applications of the same type may have the similar behavior under the same input file. The following sentence is completely messed up: Given the same input file, if there is no integer overflow for donor applications whereas it happens for recipient applications, there should have a patch in donor applications which can fix the integer overflow that occurs in recipient applications. CodePhage does such transferring. INTREPAIR differs in that its code repair technique enable it to automatically generate repairs in the absence of donor applications.

AIC/CIT/RAO [54] is a static source code analysis tool built as an Eclipse plug-in which provides several code transformations to avoid integer errors. Three types of code transformations are provided: add integer cast (AIC), replace arithmetic operator (RAO), and change integer type (CIT). Based on these three safe code transformations, this tool can protect against integer overflow without the need to detect the integer overflow first. However, these three transformations cannot be applied to all unsafe situations and a fraction of the variable declarations are also modified since in some situations, the preconditions which have to be checked are far more complex than what AIC/CIT/RAO can cover. In this situation, no transformations are applied. The tool has a runtime overhead of the hardened programs for each of the three types of transformations which is over 30% (see Figure 3 in [54]) compared with the original program. This is not acceptable in real software. In contrast, INTREPAIR first detects integer overflows using static symbolic analysis and then it generates a repair which can protect against the integer overflow with a much lower runtime overhead than the three above presented transformations.

DirectFix [161] is a static test-case-based analysis tool used for generating simple repairs at the expression level (*i.e.,* other tools operate at the statement level) which does not target specifically integer overflow repairs. This tool relies on test cases in order to locate the bug. The intuition behind this tool is to propose simplistic repairs which generate less regressions compared to other tools such as SemFix [205]. The repairs of DirectFix are up to 56% identical to the ground truth repairs contained in the SIR testsuite [72] and target operators in existing expressions. Compared to IntRepair which first locates the bug and then proposes complex repairs which can have multiple lines of code, DirectFix does not validate the repair. Further, the authors claim that as the repairs are more minimalistic, they should generate less regressions as other tools. We do not agree with the hypothesis from DirectFix that *...simple repairs are likely to be less hazardous.* general simplistic repairs always lead to less regressions. Further, in case the bug cannot be removed by introducing the repair at the expression level or there is no expression which can be mutated by DirectFix, then this tool cannot fix the bug. In contrast, INTREPAIR detects the bug at the statement level and encloses the statement with a repair which can eliminate the bug.

3.3 Repairing Buffer Overflows

3.3.1 Generating Buffer Overflow Repairs

Source code patches for repairing buffer overflow bug can be generated in different ways [109], from free form bug reports [148], [277], [3], from statically defined patch patterns [132], [105], from test suite using SMT solver [69], [205], from test suite and genetic programming [141], [278], by replacing the unsafe *libc* [250], [236], functions with safe functions, [58].

Hafiz *et al.* [244] addressed *buffer overflows quick fixing* by replacing unsafe library functions with safe alternatives. Cowan *et al.* [60] have used *static analysis for generating code patches* based on four approaches in which the buffer overflow vulnerabilities can be *defended*. Jacobs [123] has proposed to use *buffer overflow refactoring patterns* as an extension for the `C` language called SMART `C`. In recent years, many *quick fix generation tools for buffer overflows* have been proposed: AutoPaG [146], SafeStack [40], DYBOC [246], TIED [12], LibsafePlus [12], LibsafeXP [145], HeapShield [79].

To the best of our knowledge, the AutoPAG [146] tool is most similar to our approach from the backward visiting of program statements perspective. BUFFREPAIR can not be compared with AutoPAG from the point of view of computation time and quick fix quality at this stage of development since AutoPAG has several limitations which we will briefly list. Our algorithm stops the search after encountering the first *non-in-place* bug fix location, whereas AutoPAG tries to detect all possible *non-in-place* bug fix location by running a repeated inefficient data flow analysis (no program execution paths used). AutoPAG is unaware of program execution paths and uses a rudimentary backward information flow propagation approach based on the sequential ordering of program statements. The analysis (no SMT solver used) is repeated until there are no visited variables in the previously constructed set of tainted variables. This set can contain all program variables and can generate a significant overhead as already mentioned by the authors of AutoPAG.

3.4 Detecting C++ Object Type Confusions

3.4.1 Virtual Table Pointer-based Tools

Clang-CFI [51, 52] (C++ object type cast checking tool) is similar to CASTSAN in that it uses no runtime library and all cast check detection metadata is computed during compile time. However, there are no publicly available evaluation results of Clang-CFI, and therefore we evaluated Clang-CFI in Section 10.4 independently. Clang-CFI relies on bitsets in order to model the class hierarchy of a program. Clang-CFI uses these bitsets to encode the valid virtual table start addresses for each class. Compared to CASTSAN, Clang-CFI has a higher runtime overhead, as the bit-set checking technique on which it relies apparently is less efficient than our virtual table based technique.

3.4.2 C++ Object Type Runtime Tracking

All currently available polymorphic and non-polymorphic object type confusion detection tools (except Clang-CFI) rely on dynamic checks (*i.e.,* LLVM's Compiler-RT is mostly used) for several key reasons, as follows. First, the object type has to be tracked during runtime. Second, this is due to the limited precision of static analysis techniques, which cannot recuperate the object type or a set of possible types before program runtime, Third, the object type confusions manifest only during runtime. Finally, object type confusions are hard to replicate statically (*i.e.,* compile time or through symbolic execution, without running the program).

However, the most significant reason is the fact that the types of casted objects, referenced by pointers, may be program input dependent and thus only precisely obtainable during runtime. On one hand, in the best case, the allocation of the object being cast can be tracked during compile time (*e.g.,* if the runtime path from allocation to cast is linear). On the other hand, in the worst case, the object type cannot be approximated (*e.g.,* the object was given via a void-pointer from an external function previously).

3.4.3 Compiler-based Tools

UBsan [99], CaVer [142], TypeSan [107], and HexType [125] are compiler-based tools that perform object type confusion detection at runtime for C++ based programs. Since HexType is the successor of TypeSan, the tools are very similar to each other from a technical perspective. These two tools and CaVer rely on a runtime metadata service and can reach a high coverage while imposing a considerable performance overhead. CASTSAN on the other hand uses metadata that is statically created at compile time and can therefore apply very performant checks at runtime. CASTSAN can protect against polymorphic casts by using vtable hierarchy based ranges and without using a black list. Thus, the assumption made by [107] w.r.t. the fact that polymorphic casts can be checked without relying on RTTI information can be confirmed. Further, CASTSAN does not rely on slow runtime metadata structures as HexType does. Furthermore, the evaluation Chapter shows that CASTSAN has less overhead than HexType, which in turn is more efficient than CaVer, UBSan and TypeSan. Compared to TypeSan, CASTSAN partially shares the instrumentation layer, which is unavoidable, but it uses completely different metadata without storing data at runtime. More precisely, CASTSAN uses the vtables of polymorphic classes. These tables that need to be in memory at runtime anyways already provide a view on the class hierarchy. That is enough for CASTSAN to perform runtime checks without relying on further metadata as maintained by HexType. HexType on the other hand reaches a higher coverage, as it can check non-polymorphic objects as well. CASTSAN is more runtime-efficient than CaVer and HexType, which both require a red-black tree to be traversed (only for the slow path) during each check.

3.4.4 Binary-based Tools

Most distant to our work is the work of Dewey *et al.* [71] were able to recuperate vtables from program binaries and detect object type confusions indirectly by checking the bounds of a virtual function call. This was achieved by enforcing a policy to check if the vptr lies inside some legitimate bounds. As suggested by the authors, their analysis is imprecise because for example—as also demonstrated by Prakash *et al.* [226]—determining the end of a vtable in binaries without RTTI information is not trivial. Thus, false positives and false negatives are raised and as such, this type of tool is in the best case usable before system deployment.

3.4.5 IVT vs. TypeSan

In contrast to what TypeSan's authors claim w.r.t. to the fact that IVT [24] (an extension of SafeDispatch [124]) can achieve similar goals as TypeSan by protecting virtual function calls, we strongly disagree with this claim, since first, IVT does not check specifically for illegal casts at the right location (*i.e.,* object dispatch checks for ensuring the forward-edge integrity are inserted at object dispatches to virtual functions only), and second, the standard IVT implementation does not detect specific type cast violations, but only a potential consequence of it. Thus, it can be confirmed after analyzing the IVT code that no particular up-cast or down-cast relation is checked during an object dispatch on a virtual function. Further, IVT cannot detect any particular type of illegal cast and it was not intended as such. On the other hand, it can be confirmed, as mentioned by TypeSan's authors, that IVT can not protect against non-polymorphic illegal object casts at all, as confirmed by the authors of IVT. First, as illegal casted `C++` objects do not necessarily require to be later on used for object dispatches, in this thesis we strongly disagree with the claim stated by TypeSan's authors that IVTs solution can only detect type confusion if the object is subjected to a virtual call, since the illegal cast is not detected, but a consequence of this which does not necessarily need to manifest for any given `C++` program. Second, the undefined behavior—can lead to an exploitable vulnerability—which results after an illegal object cast, can manifest *freely* until the casted object is used and checked by IVT in a virtual function object dispatch. Finally, we agree with TypeSan's authors which correctly observe that virtual dispatch protection schemes [286, 106] do not prevent the misuse of type confusion in general.

3.4.6 CastSan vs. IVT

IVT [24] is a compiler-based (Clang/LLVM) tool which protects forward edges (object dispatches) during runtime by checking that these point into a legitimate class sub-hierarchy. IVT is not intended to detect object type confusions but it can combat a consequence of object type confusions (*i.e.,* hijacking virtual dispatch to execute arbitrary code). IVT can detect sibling and non-sibling (located on the same branch) polymorphic object type confusions in theory only if there is an object dispatch in the code which uses this casted object. Thus, the cast can be detected but not at the right location in the code and only if on the casted object an object dispatch is performed later after the cast was performed. Note that IVT cannot detect object type confusions at all if no object dispatch happened after the object was cast since IVT adds checks just before each object dispatch. The vtable ranges computed by IVT are similar to the ranges used by CASTSAN, but these are built for a different purpose and used in a different place in code (*i.e.,* object dispatch) and thus with a lower precision than CASTSAN. CASTSAN uses more precise class hierarchy sub-trees when determining the minimal allowed range for each checked object dispatch than IVT. Finally, the vtable range concept of IVT is reused by CASTSAN to detect object type confusions with higher precision and more effectiveness than IVT could theoretically do.

3.5 Assessing Control Flow Integrity Defenses

3.5.1 Defense Assessment Metrics

AIR [290], fAIR [261], and AIA [91] metrics have limitations (see Carlini *et al.* [36]) and are currently the available CFI defense assessment metrics which can be used to compare the protection level offered by state-of-the-art CFI defenses w.r.t. only forward-edge transfers. These metrics provide average values which shed limited insight into the real offered protection level and thus cannot be reliably used to compare CFI-based defenses. Most recently, ConFIRM [283] also attempted to evaluate CFI, especially the compatibility, applicability, and relevance of CFI protections with a set of microbenchmarking suites. In contrast, is not a benchmark suite but rather a framework for modeling CFI defenses and comparing them against each other w.r.t. protection level these offer. Burow *et al.* [32] propose two metrics: (1) a qualitative metric based on the underlying analysis provided by each of the assessed techniques, and (2) a quantitative metric that is the product of the number of equivalence classes (EC) and the inverse of the size of the largest class (LC). In contrast, we propose , a CFI defense assessment framework and CTR, a new CFI defense assessment metric based on absolute forward-edge reduction set analysis, without averaging the results. CTR provides precise measurements and facilitates comprehensive CFI defense comparison.

3.5.2 Static Gadget Discovery

Wollgast *et al.* [281] present a static multi-architecture gadget detection tool based on the analysis of the intermediate language (IL) of VEX, which is part of the Valgrind [265] programing debugging framework. The tool can find a series of CFI-resistant gadgets. Compared to LLVM-CFI, both tools leave the gadget chain building as a manual effort. In contrast, when using LLVM-CFI, it is possible to define a specific CFI-defense policy and search for available gadgets while the tool of Wollgast *et al.* specifies CFI resistant gadgets by defining their boundaries (start and end instructions). These have to conform to some constraints and respect the normal program control flow of the program in order to be considered CFI resistant. These types of gadgets are more thoroughly described by Goktas *et al.* [104], and Schuster *et al.* [239].

RopDefender [67], ROPgadget [233], and Ropper [237] are non-academic gadget detection tools for program binaries. These tools are used to search inside program binaries with the goal to find consecutive machine code instructions, which are similar to a previously specified set of rules that define a valid gadget. While performing a fast search, these tools however cannot detect defense-aware gadgets, since these tools do not model the defense applied to the program binary. As such, these tools cannot determine which gadgets are usable after a certain defense was applied.

3.5.3 Dynamic Gadget Discovery

Newton [267] is a runtime binary analysis tool which relies on taint analysis to help significantly simplify the detection of code reuse gadgets defined as callsite and legal calltarget pairs. At first, quite effectively as demonstrated by the authors of Newton, this tool can model part of the byte memory dependencies in a given program. Newton is able to model a series of code reuse defenses by not focusing on a specific attack at a time. Finally, Newton is able to craft attacks in the face of several arbitrary memory write constraints.

Conti *et al.* [55] introduce StackDefiler, a set of stack corruption attacks that leverage runtime object allocation information in order to bypass fine-grained CFI defenses. Based on the fact that Indirect Function-Call Checks (IFCC) [261] (also valid for VTV) spills critical pointers onto the stack, the authors show how CRAs can be built even in presence of a fine-grained CFI defense. Compared to LLVM-CFI which is based on control flow bending to legitimate targets, the authors of StackDefiler show an alternative approach for crafting CRAs. More specifically, Conti *et al.* show that information disclosure poses a severe threat. They also show that shadow stacks which are not protected through memory isolation are an easy target for the attacker.

Argument Corruptible Indirect Call Site (ACICS) [80] gadgets are detected during runtime by the ADT tool, in a similar way as Newton's detects gadgets. Note that the ACICS gadgets are more constrained then those of Newton. For example, only attacks where the function pointer and arguments are corruptible on the heap or in global memory are taken into consideration. As Newton does not impose so many constraints on the gadgets, it can find depending on the analyzed program more sophisticated attacks where the attack elements can be corrupted in many more complex and indirect ways. Similar to LLVM-CFI, the ADT tool is able to craft an attack in the face of IFCC's CFI defense policy by finding pairs of indirect callsites that match to certain functions which can be corrupted during runtime. In contrast, LLVM-CFI, is not program input dependent as it is not a runtime tool. Therefore, it can find all corruptible indirect callsite and function pairs under a certain modeled CFI policy.

3.5.4 Existing Metrics vs. Our Metrics

AIR [290], fAIR [261], and AIA [90] are the only available metrics which can be used to assess the protection level offered by a CFI-based policy w.r.t. only forward-edge transfers. These metrics provide average values which shed limited insight into the real offered protection level. Further, these metrics can not be used to compare tools against each other since they provide average values.

In contrast, in this thesis we provide new metrics (see Appendix) for assessing not only the absolute forward-edge reduction set, but also the backward-edge target set reduction, the runtime damping of each CFI policy as well as the gadget availability set after a program was hardened with a CFI policy.

Additionally, two other relevant metrics are proposed by Burow *et al.* [32] as follows: (1) qualitative metric, based on the underlying analysis provide by each of the assessed techniques, and (2) quantitative metric, the product of the number of equivalence classes (EC) and the inverse of the size of the largest class (LC).

Finally, we note that the existing forward-edge CFI metrics are mostly used without specifying the total number of callsites contained in the hardened program and relating this to the total number of callsites which are protected.

3.6 Protecting Backward Edges

According to Burow *et al.* [32], backward-edge CFI-based protection approaches can be classified according to their level of protection as follows: low, medium and high level of protection of the backward edge. Further, Burow *et al.* classify the backward-edge protection tools into binary-based, source-code-based and other types of tools (*e.g.,* with hardware (HW) support, runtime-based).

3.6.1 Source Code based Tools

Clang's SafeStack [49], Hypersafe [274], CF-Locking [19], MIP [209], CF-Restrictor [224], KCoFi [61], RockJIT [211], CCFI [160], Kernel CFI [90], MCFI [210], and CFL [19] have relatively high callee target set precision (except Clang's SafeStack which has a one-to-one target set mapping precision) w.r.t. enforced address return set for each callee. These tools have in general a high coverage, *i.e.,* almost all backward edges are protected.

CIFXX [23] is a source code tool used to protect forward-edges only by instrumenting the program with CFI checks before object dispatches. The OTI policy used by CFIXX guarantees, that polymorphic objects have the correct type associated with them at run time. As such, the dynamic dispatch is per object basis protected. As such, CFIXX imposes a one-to-one mapping between legal callsites and calltargets. Compared to ρFEM, CFIXX does not protect backward edges, but its forward edge policy, in our opinion, could be successfully used to protect backward edges by enforcing the caller/callee function calling convention.

CFL [20] is a compiler-based tool built upon the GCC compiler used for protecting backward edges, only by instrumenting the source code (only for 32-bit) of a compiled program. CFL uses a statically pre-computed program control flow graph (CFG). Thus, this technique (as it can be observed) relies on the precision of the computed CFG. By the so called *locking* and *unlocking* operations between each indirect control flow transfer and legitimate backward edge return targets (addresses), the CFL's technique is enforced. This results in a 1 to N relation. CFL does not provide any numbers w.r.t. how many backward edge targets are allowed on average for each function return (backward edge). CFL can protect against code reuse attacks for statically linked 32-bit binaries which violate the statically precomputed CFG. CFL provides three modes of operation: (1) just alignment, (2) single-bit CFL, and (3) full CFL, which each have different performance overheads. The authors claim that the control flow of the program can not deviate more than once. The underlying technique enforces control flow integrity (CFI) by assigning different values ($k = 0$ unlocked, $k = 1$ indirect `jmp` or `call`, k is greater then 1 return from a non-indirect-callable function, k is less then 0 return from an indirectly-callable function) to k. By setting the value of k to certain values (see above value ranges), different indirect control flow transfers are allowed or forbidden. The overhead of full CFL for certain SPEC CPU2006 programs (see Figure 4 in [20]) ranges from 1% to 16% and from 1% to more than 20% for SPEC CPU2000 programs. This makes CFL not applicable to scenarios where performance is key. Finally, no average or geomeans runtime overhead values are provided and no evaluation on a real software system was performed by its authors, making real-world usage of CFL questionable.

piCFI [212] is a compiler-based solution which lazily builds a CFG on the fly during program execution. Indirect edges are added in the CFG before indirect branches need those edges. piCFI disallows adding edges, which are not present in the statically computed all input CFG (this CFG serves as an upper bound for the runtime constructed CFG). piCFI activates a function return address when the function is called. In such a situation, the function is allowed to return only to the activated address. As the CFG in which return addresses are stored increases monotonically (because addresses are not deactivated), this reduces the security (since more and more addresses become available over time to return to) and performance as well. Compared to piCFI, the return target set of backward edges of ρFEM does not change during runtime and thus the performance and security do not vary during runtime. Further, no cleanup operations (garbage collection or address deactivation) are needed when using ρFEM.

PittyPat [74] introduces a fine-grained path-sensitive CFI approach for protecting both forward and backward edges. It uses the processor trace (PT) CPU feature to ensure that a program satisfies a stronger, path-sensitive variation of CFI. Additionally, two Clang/LLVM instrumentation passes are used to build precise points-to target sets and to insert checks, which try to determine if the indirect transfer is legitimate. PittyPat generates a smaller points-to target set then piCFI for forward edges only. The size of PittyPat's points-to target set is larger then ρFEM's calltargets set, which is obtained through a virtual table sub-hierarchy analysis. For example, PittyPat allows up to 218 targets per callsite for the SPEC CPU2006 `403.gcc` program (see Figure 3(a) in [74] for more details), whereas the return target set of ρFEM is much smaller, as it is directly proportional to the enforced forward-edge sub-hierarchy and thus much smaller than 218 entries. This is because ρFEM's return target set is direct proportional to: (1) the depth of the class hierarchy for a certain object type, and (2) the number of entries (*i.e.,* number of parameters, their types, return type *etc.*) used to compute function signatures.

3.6.2 Binary-based Tools

The original CFI implementation from Abadi *et al.* [1], MoCFI [156], kBouncer [216], CCFIR [289], bin-CFI [290], O-CFI [187], PathArmor [266], and LockDown [221] mostly suffer from imprecision (high number of reused labels), have low runtime efficiency, and do not support shared libraries at all. ρFEM is similar to Abadi's *et al.* CFI [1] approach w.r.t. the fact, that we also use IDs in order to check to which calltarget return sites it is allowed to return. Compared to the original CFI implementation, ρFEM does not rely on hardware support (*i.e.,* `GS` register) or code segments. Further, our approach is applicable for both 32-bit and 64-bit systems.

Marx [218] is a binary-based tool used for protecting only forward edges resulting from C++ object dispatches. It reconstructs a quasi-class hierarchy from the binary and it enforces only certain virtual table ranges to be legal for each previously detected callsite. Compared to ρFEM, this tool does not have the goal to protect backward edges. Further, the size of the forward edge mappings are larger than the sizes of ρFEM's mappings, due to a less precise analysis compared to a source-code-based tool which is true in general when comparing source code and machine code analysis tools against each other.

TypeArmor [268] is a binary-based tool used to protect indirect forward-edge control flow transfers resulting from only object dispatches. TypeArmor imposes a CFI policy on legal indirect transfers between callsites, which provides up to six parameters and calltargets, which are allowed to consume up to the number of parameters a compatible callsite provides. Compared to ρFEM, TypeArmor does not provide protection for backward edges and its forward edge mapping precision is lower than the one offered by ρFEM, for the same reasons mentioned above for the Marx tool.

3.6.3 Other Types of Tools

ROPecker [44], HW-asst. CFI [66], and CFIGuard [284] are mostly relying on HW features which were not specifically built for protecting backward edges and are not runtime effective. Further, their precision is rather low and most of them have a questionable effectiveness against state-of-the-art CRAs.

HAFIX [65] is a HW and compiler-assisted approach usable for protecting backward edges by introducing three new instructions `CFIBR`, `CFIDEL`, and `CFIRET` for monitoring during runtime when entering a certain function and determining which is the active label set where this function can return to. Finally, HAFIX was bypassed in the work presented by Theodorides *et al.* [259] and for this reason, we present in Section 9.4.2 the five available bypassing techniques and assess them w.r.t. ρFEM.

Intel CET [118] is the first hardware feature specifically designed to assist with the protection of the integrity of backward edges by for example, helping to implement a more efficient shadow stack. It introduces the `ENDBRANCH` instruction to the ISA to mark the legal target for an indirect branch or jump. The `ENDBRANCH` instruction can be used to develop compiler and/or OS-based attack mitigation solutions. Compared to ρFEM, Intel CET's new HW feature has the potential to outperform check-based tools w.r.t. security and performance; however, real-world implementations and experimental results are not yet available as open source.

Windows CFGuard [167] (as of 2017 removed from Windows beta) is a protection mechanism, which is based on the interplay of a user space agent and operating system (OS) kernel space support. This feature was deployed in Windows 8.1 and Windows 10 and can only protect forward edges. As a consequence, Microsoft has recently released return flow guard (RFG) [282] in Windows 10 Redstone in order to protect the return edges as well. RFG saves each function return address to `fs:[rsp]` at the entry of each function, and compares it with the return address on stack before returning. As CFGuard, RFG requires compiler (*i.e.,* Microsoft Visual Studio) and OS support. Each hardened program is instrumented with the help of the compiler. The compiler instruments the program file by reserving a certain number of instruction spaces in the form of `NOP` (no operation) instructions. These `NOP` operation will be replaced, on an operating system, which supports this feature, during runtime of the binary, by RFG instructions, which will be used to check function return addresses. In case the OS does not support the RFG functions, the `NOP` instructions will not interfere with the normal execution flow of the hardened program. Important to note is the main difference between RFG and GS (Buffer Security Check) is that in the case of GS, the stack cookie can be obtained using an information leak or by brute-force. In contrast, RFG drastically increases the difficulty of retrieving the stack cookie, by writing each cookie to the Thread Control Stack (TCS). We point out that the TCS is out of attacker reach. Unfortunately, no efficiency or runtime performance measurements are currently available thus, it is hard to reason herein how effective this OS feature is.

PathArmor [266] emulates a shadow stack by validating the Intel Last Branch Record (LBR). This technique is based on a buffer with limited size, which can hold up to 32 *from* and *to* addresses pairs. By validating these addresses and comparing these addresses against a previously computed control flow graph (CFG), PathArmor can, in some situations, detect if the caller-callee function calling convention was violated. Unfortunately, due to the fact that addresses pairs arrive into the LBR buffer at a pace considerably higher than the rate of which these address pairs can be extracted and checked against the CFG, this tool has limited control flow violation detection rate.

Intel Processor Trace (Intel PT) [117] is a new hardware feature introduced by Intel with the fifth generation of the Intel Core processors (Broadwell architecture). This feature helps provide execution and branch tracing information by being able to monitor the five types of control flow, affecting instructions (called Change of Flow Instruction (CoFI)) specified by Intel. In contrast to the Intel Last Branch Record (LBR), the size of the output buffer when Intel PT is used is no longer limited by special registers. The output can be stored in main memory. In case the output is repeatedly and timely emptied, traces of arbitrary length can be generated thus the limitations of the LBR register are avoided. Intel PT is a very promising hardware feature which was already used to protect backward edges in GRIFFIN [89], which has 9.5% runtime overhead on average for the combination policy, and PT-CFI [102], which has in average a runtime overhead above 20%. Finally, we believe that this feature could be further used in the future to design other, more efficient backward edge protection techniques, which could be based for example on OS and compiler interplay.

3.6.4 Backward Edge Attack Mitigation

According to the currently most comprehensive survey by Burow *et al.* [32] on CFI-based tools, which assesses backward-edge protection tools, there are tools offering low, medium and high-level backward edge protection. Further, this survey classifies the backward edge protection tools as binary-based, source-code-based or other types (*i.e.,* with HW support, *etc.*) as follows.

Binary-based tools. Original CFI implementation from Abadi et. al. [1], MoCFI [156], kBouncer [216], CCFIR [289], bin-CFI [290], O-CFI [187], PathArmor [266], and Lock-

Down [221] mostly suffer from imprecision (high number of reused labels), have low runtime efficiency, and do not support shared libraries at all.

Source code based tools. SafeStack [49], Hypersafe [274], CF-Locking [19], MIP [209], CF-Restrictor [224], KCoFi [61], RockJIT [211], CCFI [160], Kernel CFI [90], MCFI [210], and piCFI [212] have relatively high precision w.r.t. enforced address return set. None of them support shared libraries (except MCFI). Further, these techniques have high backward edge protection coverage, since almost all backward edge transfers can be protected.

Other types of tools ROPecker [44], hardware (HW)-assisted CFI [66], and CFIGuard [284] mostly rely on HW features which were not specifically built for protecting backward edges. Thus, these tools are not runtime effective. Their precision is rather low and they are mostly ineffective against state-of-the-art code reuse attacks.

3.7 Protecting Against Code Reuse Attacks

In this Section, we briefly review related work by focusing on binary-based tools.

3.7.1 Mitigation of Simple Code Reuse Attacks

In the last couple of years researchers have provided many versions of new Code Reuse Attacks (CRAs). These new attacks were possible since DEP [164] and ASLR [219] were successfully bypassed mostly based on Return Oriented Programming (ROP) [31, 133, 242] on one hand, and due to the discovery of new exploitable hardware and software primitives on the other.

ROP started to present itself in the last couple of years in many faceted ways such as: Jump-oriented Programming (JOP) [20, 39, 64] which uses jumps in order to divert the control flow to the next gadget and Call Oriented Programming (COP) [37] which uses calls in order to chain gadgets together. CRAs have many manifestations and it is out of scope of this work to list them all.

First, CRAs can be mitigated in general in the following ways: (i) binary instrumentation, (ii) source code recompilation and (iii) runtime application monitoring. Second, there is a plethora of tools and techniques which try to enforce CFI-based primitives in executables, source code and during runtime. Thus, we briefly present the solution landscape together with the approaches and the techniques on which these are based: (a) fine-grained CFI with hardware support, PathArmor [266], (b) coarse-grained CFI used for binary instrumentation, CCFIR [289], (c) coarse-grained CFI based on binary loader, CFCI [291] (d) fine-grained code randomization, O-CFI [187], (e) cryptography with hardware support, CCFI [160], (f) ROP stack pivoting, PBlocker [227], (g) canary-based protection, DynaGuard [223], (h) runtime and hardware support-based on a combination of LBR, PMU and BTS registers CFIGuard [284], and (i) source code recompilation with CFI and/or randomization enforcement against JIT-ROP attacks, MCFI [210], RockJIT [211] and PiCFI [212]. The following tools address vTable protection through binary instrumentation, but fail to mitigate against COOP: vfGuard [226], and vTint [287].

The above list is not exhaustive and new protection techniques can be obtained by combining available techniques or by using newly available hardware features or software exploits. However, notice that none of the above mentioned techniques and tools can be used to mitigate COOP attacks.

3.7.2 Mitigation of Advanced Code Reuse Attacks

Recursive-COOP [59], COOP [238], Subversive-C [143] and the attack of Lan *et al.* [137] are forward-edge-based CRAs which cannot be addressed with: (i) shadow stacks techniques and hardware-based approaches such as Intel CET [118] (*i.e.,* since advanced COOP do not violate the caller/callee convention), (ii) coarse-grained Control-Flow Integrity (CFI) [1, 2] techniques, and (iii) OS-based approaches such as Windows Control Flow Guard [167] since the precomputed CFG does not contain edges for indirect callsites which are explicitly exploited during the COOP attack.

Binary based Forward Edge Protection. TypeArmor [268] is a binary instrumentation tool that can protect against COOP. TypeArmor uses a fine-grained CFI-policy-based on caller/callee (but only indirect callsites) matching, which checks during runtime if the number of provided and needed parameters match. TYPESHIELD is related to TypeArmor [268], since we also enforce strong binary-level invariants on the number of function parameters. Further, TYPESHIELD also aims for exclusive protection against advanced exploitation techniques, which can bypass fine-grained CFI schemes and vTable protections at the binary level. However, TYPESHIELD offers a better restriction of calltargets to callsites, since we not only restrict based on the number of parameters, but also on the width of their types. This results in much smaller buckets that in turn can only target a smaller subset of all address-taken functions. However, we rely for that on the variety of parameter types and when there is none, we will degrade into a parameter count policy.

We are aware that there is still a long research path to go until binary-based techniques can recuperate program based semantic information from executable with the same precision as compiler-based tools. This path could even be endless since compilers are optimized for speed and are designed to remove as much as possible semantic information from an executable in order to make the program run as fast as possible. In this light, TYPESHIELD is another attempt to recuperate just the needed semantic information (types and number of function parameters from indirect callsites) in order to be able to enforce a precise and with low overhead primitive against COOP attacks.

VCI [78] is a binary-based tool (7.9%) based on DynInst which can protect forward edge indirect control flow violations based on reconstructing a quasi program class hierarchy (*i.e.,* no class root node and the edges are not directed). The authors claim that VCI is 10 times more precise w.r.t. reducing the calltarget set per callsite. In contrast to TYPESHIELD VCI can not protect backward-edge violations and we arguably due to the conservative analysis the VCI could skip some corner situations allowing not legitimate calltargets.

Marx [218] is most similar to VCI and as VCI this tool reconstructs the same type of quasi program class hierarchy. No runtime efficiency numbers were provided in the thesis. The authors claim that Marx can recuperate a class hierarchy which is more precise than that of IDAPro. The thesis is geared towards first providing a tool which can be used by analyst in order to reverse engineer a binary. The precision of the calltarget set reduction per callsite should be similar to those of VCI but no comparison was compared in the thesis. Compared to TYPESHIELD Marx can not protect against backward-edge violations and arguably has in common with VCI several limitations.

VTPin [234] is a runtime-based tool ($\approx$5%) used for protecting against VTable hijacking, via use-after-free vulnerabilities. VTPin pins all the freed VTable pointers on a safe VTable under VTPin's control. For each object deallocation, VTPin deallocates all space allocated, but preserves and updates the VTable pointer with the address of the safe VTable. As a consequence, a dangling pointer can invoke a method provided by VTPin's safe object. TPin needs to keep track of metadata in order to detect runtime dangling pointer violations. The tool can not protect against the COOP attack since the COOP attack does not rely on dangling pointers. In contrast with TYPESHIELD, this tool cannot protect against backward-edges violations.

In this thesis, rather than claiming that the invariants offered by TYPESHIELD are sufficient to mitigate all versions of the COOP (as [268] does) attack we conservatively claim that TYPESHIELD further raises the bar w.r.t. what is possible when defending against COOP attacks on the binary level.

Source Code Based Techniques. Indirect callsite targets are checked based on vTable integrity. Different types of CFI policies are used such as in the following tools: SafeDispatch [124], IFCC/VTV [261] LLVM and GCC compiler. Additionally, the Redactor++ [59] uses randomization vTrust [286] checks calltarget function signatures, CPI [135] uses a memory safety technique in order to protect against the COOP attack.

There are several source code based tools which can successfully protect against the COOP attack. Such tools are: ShrinkWrap [106], IFCC/VTV [261], SafeDispatch [124], vTrust [286], Readactor++ [59], CPI [135] and the tool presented by VTI [24]. These tools profit from high precision since they have access to the full semantic context of the program trhough the scope of the compiler on which they are based. Because of this, these tools mostly target other types of security problems other than the ones binary-based tools address. For example, some of the last advancements in compiler-based protection against code reuse attacks address mainly performance issues. Currently, most of the above presented tools are only forward edge enforcers of fine-grained CFI policies with an overhead from 1% up to 15% (see [32] for more details).

Runtime Based Techniques. Several promising runtime-based defenses against advanced CRAs exist but currently none of them can successfully protect against the COOP attack.

IntelCET [118] is based on `ENDBRANCH`, a new CPU instruction which can be used to enforce an efficient shadow stack mechanism. The shadow stack can be used to check during program execution if caller/return pairs match. Since the COOP attack reuses whole functions as gadgets and does not violate the caller/return convention than the new feature provided by intel is useless in the face of this attack. Nevertheless, other highly notorious CRAs may not be possible after this feature will be implemented main stream in OSs and compilers.

Windows Control Flow Guard [167] is based on a user-space and kernel-space components which by working closely together can enforce an efficient, fine-grained CFI policy-based on a precomputed CFG. These new feature available in Windows 10 can considerably raise the bar for future attacks but in our opinion advanced CRAs such as COOP are still possible due the typical characteristics of COOP.

PathArmor [266] is yet another tool which is based on a precomputed CFG and on the LBR register which can give a string of 16 up to 32 pairs of from/to addressed of different types of indirect instructions such as `call`, `ret`, and `jump`. Because of the sporadic query of the LBR register (only during invocation of certain function calls) and because of the sheer amount of data which passes through the LBR register this approach has in our opinion a fair potential to catch different types of CRAs but we think that against COOP this tool can be used only with limited success. First, because of the fact that the precomputed CFG does not contain edges for all possible indirect callsites which are accessed during runtime. Second, the LBR buffer can be easily fooled by interleaving legitimate with illegitimate indirect callsites during the COOP attack.

3.7.3 Mitigation of Forward Edge based Attacks

Binary-based tools. τCFI is closely related to TypeArmor w.r.t. the forward edge analysis. TypeArmor [268] ($\approx$3% runtime overhead in geomean) enforces a CFI policy based on the parameter count policy. Compared to τCFI, TypeArmor does not use function parameter types and assumes a backward-edge protection is in place. VCI [78] and Marx [218] are both based on approximated program (quasi) class hierarchies; they (1) do not recover the root class of the hierarchy, and (2) the edges between the classes are not oriented; thus both tools enforce for each callsite the same virtual table entry (*i.e.,* index based) contained in one recovered class hierarchy represented by father-child relationships between the recovered vtables. Finally, both tools use up to six heuristics and simplifying assumptions in order to make the problem of program class hierarchy reconstruction tractable. Compared to these tools, τCFI tries not to reconstruct a high-level metadata data structure (class hierarchy), but rather performs analysis on the usage of provided and consumed parameters at the callsites and calltargets.

Source code based tools. To the best of our knowledge, only the Clang-CFI [149] and IFC-C/VTV [261] (up to 8.7% performance overhead) compiler based tools are deployed in practice and can be used to check legitimate from illegitimate indirect forward-edge calls during runtime by checking if virtual object pointers comply with the program class hierarchy inheritance relationships. Furthermore, ShrinkWrap [106] is a GCC-compiler-based tool which further reduces the legitimate virtual table ranges for a given indirect callsite (*i.e.,* object dispatch) through precise analysis of the program class hierarchy and virtual table hierarchy. Evaluation results show similar performance overhead results as the previous mentioned tools but with more precision with respect to legitimate virtual table entries (calltargets) per callsite. We noticed by analyzing the previous research results that the overhead incurred by these security checks can be very high due to the fact that for each callsite many range checks have to be performed during runtime. Therefore, in our opinion, despite its security benefit, these types of checks are most likely not applicable to software where performance is key.

Other types of tools. Other highly promising source code based tools (albeit also not deployed in practice) which can overcome some of the drawbacks of the previously described tools are as follows. Bounov *et al.* [24] presented a tool ($\approx$1% runtime overhead) for indirect forward-edge callsite checking based on virtual table layout interleaving. The tool has better performance than VTV and better precision with respect to allowed virtual tables per indirect callsite. Its precision (selecting legitimate virtual tables for each callsite) compared to ShrinkWrap is lower since it does not consider virtual table inheritance paths. vTrust [286] (average runtime overhead 2.2%) enforces two layers of defense (virtual function type enforcement and virtual table pointer sanitization) against virtual table corruption, injection and reuse.

3.7.4 Mitigation of Backward Edge based Attacks

Backward-edge-based code reuse attacks exploit the indirection provided by the return instructions of a function. Usually each modern compiler builds caller/callee pairs by adhering to the so called caller-callee calling convention. This calling convention basically specifies that for each indirect call the return address of the function which returns after it was called lies at the next address of the call instruction. This calling convention is violated by all ROP attacks and also by more recent advanced code reuse attacks. Intel CET [118] is a promising technology from Intel in which the X86 instruction set is updated with new instructions (*i.e.,* `END_BRANCH`) instruction which should facilitate an efficient implementation of shadow stack implementations. Currently, this technology is not available and it is not foreseeable when these features will be available in mass production. According to a comprehensive survey by Burow *et al.* [32], tools that provide backward-edge protection offer low, medium, and high levels of protection w.r.t. backward edges. Further, this survey provides runtime overhead comparisons, classifies the backward-edge protection techniques into binary-based, source code based, and other types (*e.g.,* with HW support, *etc.*). Due to page restriction, we review only binary tools.

Binary based Backward Edge Protection. The original CFI implementation of Abadi *et al.* [1], MoCFI [156], kBouncer [216], CCFIR [289], bin-CFI [290], O-CFI [187], PathArmor [266], LockDown [221] mostly suffer from imprecision (high number of reused labels), have low runtime efficiency, and most of them protect either forward edges or backward edges assuming a perfect shadow stack implementation is in place. In contrast, τCFI makes no assumptions about the presence of a backward-edge protection. Further, τCFI provides a technique for protecting forward edges and does not rely on a shadow stack approach for protecting backward edges.

The CFI-based implementation of Abadi *et al.* [2] is the first binary-based implementation of a shadow stack. While at first glance promising, this implementation suffers from high performance overhead of around 21% due to the fact that the inserted checks before each function return instruction are not runtime efficient. Further, this tool has high imprecision due to the fact that labels are reused and in this way not legitimate return addresses become legitimate, thus these could be exploited by a skilled attacker.

Compiler-based Techniques. SafeStack [49], Hypersafe [274], CF-Locking [19], MIP [209], CF-Restrictor [224], KCoFi [61], RockJIT [211], CCFI [160], Kernel CFI [90], MCFI [210], piCFI [212] relatively high precision w.r.t. enforced address return set, and all do not support shared libraries (except MCFI), have high coverage (almost all backward edges are protected).

LLVM SafeStack [135] is a compiler-based approach in which for each function stack a shadows stack copy is build with the help of the Clang compiler. These additional stacks are hidden by at least one level of indirection from the attacker such that she can not interfere with it. This approach is effective but suffers from a big binary blow-up which is not acceptable in any usage scenario. Currently, it was demonstrated that this implementation can be bypassed [94].

Other types of Techniques. ROPecker [44], HW-asst. CFI [66], CFIGuard [284] are mostly relying on HW features which were not specifically built for protecting backward edges and for this reason the tools are not runtime effective and their precision is rather low and mostly ineffective against state-of-the-art code reuse attacks.

Windows CFGuard [167] is a technology by Microsoft deployed into Windows 8.1 and Windows 10. This technology allows to protect backward-edges by checking in a shadow stack like fashion for backward-edge violations. The implementation is based on an interplay between user space and kernel space thus there is high potential that this implementation is highly efficient even though no official evaluation results are available.

PathArmor [266] is a runtime-based tool based on a Linux loadable module which can emulate shadow stack checks by using the `LBR` register, which stores callsite and target address pairs. The capacity of the LBR register is limited to 32 address pairs which can be stored. The tools suffers from high runtime overhead and is imprecise since the address pairs can not be analyzed at the same speed as they arrive in this register. For this reason, some pairs are skipped and thus the attacker has the chance to mount and attack.

Chapter 4

IntDetect: Static Detection of Integer Overflow Based Memory Corruptions

In this Chapter, which belongs to the first part of this thesis, we present a framework, inside which we first designed, implemented and, integrated an integer overflow detection tool, called INTDETECT into our static source code analysis engine. INTDETECT is capable of efficiently detecting integer overflows in `C` source code programs by employing static symbolic execution. With this approach, we answer **RQ1** by providing a tool which can reliably detect integer overflows, does not suffer from false negatives, and is integrated in a well established and widely used IDE (*i.e.,* Eclipse IDE). Finally, note that parts of this Chapter have already been published by Muntean *et al.* [200].

4.1 Introduction

What is the overall view? In the 2011 top 25 most dangerous software errors [186] MITRE classifies integer overflows as one of the main sources for different types of software vulnerabilities. Integer overflow errors are responsible for a series of vulnerabilities in OpenSSH [183] and Firefox [185], for example, which allow attackers to execute arbitrary code (CRA). In reality, software failures can materialize, for example, during the crash of the Ariane 5 flight 501 in 1996 due to an attempt to cast a floating point value to a 16-bit integer value which resulted in a truncation error.

Integer numerical errors in software applications are costly, hard to detect—there are several types of integer overflows and some of them are inserted into code intentionally and others unintentionally—and exploitable. According to Brumley *et al.* [29] state that there are four types of integer related problems. First, integer overflows occur at run-time when the result of an integer expression exceeds the maximum value for its respective type. Second, integer underflow appear at run-time when the output of an integer expression is less than the minimum value that the assignee can hold, thus *wrapping* to the maximum integer for the type. Third, integer signedness occur when a signed integer is interpreted as unsigned, or vice-versa. Fourth, integer truncations appear when an integer with smaller width—number of bits—has to hold an integer with larger width. Finally, there are intentional and unintentional uses of integer overflows and illegal uses of operations such as shifts which can lead to integer related errors.

What is the problem? Integer related problems are typically exploited indirectly, contrary to *e.g.,* buffer overflows which can be exploited directly or indirectly. Typical indirect integer bug exploitations are as follows. First, Denial of Service (DoS) attacks where the exploit causes infinite loops or excessive memory allocation. Second, arbitrary code can be executed when an integer vulnerability results in insufficient memory allocation which afterwards can be exploited by buffer overflows, heap overflows and overwrite attacks. Third, an upper bound sanitization check can be bypassed when unexpected negative integer values are used. Fourth, a logic error, where a reference counter in the NetBSD OS (CVE-2002-1490) [184] was manipulated by an attacker that resulted in the premature freeing of an object from memory. Finally, array index attacks are caused by a vulnerable integer value which can be used as array index so that attackers can precisely overwrite arbitrary bytes in memory.

What are the existing solutions? These types of integer related problems can be addressed with code analysis techniques (*e.g.,* static and dynamic code analysis) UQBTng [280], IntScope [272] formal modeling of `C` semantics, Extended Static Checking (ESC)—usage of code annotations, Satisfiable Modulo Theories (SMT) solver, test cases to trigger the bug, compiler integration (Clang), formal modeling of integer typing rules, dynamic analysis RICH [29], BRICK [42], SmartFuzz [188], SAGE [93] and IOC [73]. RICH [29] instruments programs to detect safe and unsafe operations based on well-known sub-typing theory. BRICK [42] detects integer overflows in compiled executables using a modified Valgrind [203] version.

What are the limitations of the solution? None of the exiting tools care can systematically analyze the program and therefore do not achieve a sufficiently high patch coverage and all other static analysis tools suffer from false negatives, whereas INTDETECT can only have false positives but no false negatives. To the best of our knowledge INTDETECT is most similar to PreFix. INTDETECT compared to PreFix focuses only one one type of integer related problem (*i.e.,* integer overflows) by using `C` function models whereas PreFix does not use function models. Furthermore, the dynamic analysis approaches compared with INTDETECT can neither exercise all bugs due to dynamically generated test cases, nor can full path coverage be achieved. For example, Ceesay *et al.* [38] added type qualifiers to detect integer overflow problems. Their approach relies on expensive system extensions, which are linked into the compiler and are used to check the code for integer errors. Their approach requires user annotation whereas our approach does not require annotations. On one hand, typically dynamic analysis have low overhead $\approx$5%, low true positives and false negatives rate, reduced path coverage, *etc.* On the other hand, side static analysis offers environment models, bit precision, low number of false positives, high path coverage, *etc.*

What is our insight? In this Chapter, we address integer overflow vulnerabilities detection through precise symbolic "modeling" of C language semantics which are responsible for integer overflows and `C` function models. Concretely, we extended the C statement processing component of our engine and carefully remodeled each external (C standard library and any other used API) used function through usage of symbolic function models. Thus, providing a precise modeling of C language semantics is the key to a high false positives and low false negatives rate.

What are our contributions? In summary, in this Chapter, we make the following contributions:

- We provide precise symbolic modeling of `C` related semantics needed for integer overflow detection, in Section 4.3.
- We design and implement an integer overflow checker (called INTDETECT) as an Eclipse IDE plug-in based on our static execution engine in Figure 4.3 and automated testing based on automatically generated jUnit test cases and Eclipse projects in Section 4.4.2 and in Section 4.4.3.
- We present an experimental evaluation of our approach based on the open source `C/C++` test case CWE_190_Integer_Overflow [207], in Section 4.4.

4.2 Threat Model

Defensive Assumptions. We align our threat model (*e.g.,* STRIDE [165]) to the general integer overflow threat model. We assume that ALSR [219] and DEP [164] are in-place and correctly functioning. We assume that the code is not compiled with any integer overflow bounds checking. Lastly, we assume that the defender has access to the source code of the application and potentially knows how to repair the integer overflow.

Attacker Capabilities. We assume a skilled attacker with sufficient resources and time to exploit the integer overflow. Further the attacker can use the integer overflow to escalate privileges or to manipulate the program counter according to his desires. The attacker is equipped with the necessary tooling to find the integer overflow in the program binary and to experiment with the code in an live interactive debugging session. Next, we don not exclude the possibility that the attacker has access to the source code of the application. Lastly, the attacker is aware of the existence of other vulnerabilities (*e.g.,* buffer overflow) which might be triggered through the previously found integer overflow. This requirement is not mandatory but a nice to have as opposed to buffer overflows, integer overflows are exploited indirectly through integer overflow to buffer overflow (*i.e.,* IO2BO) vulnerabilities.

4.3 Design and Implementation

Figure 4.1 depicts the design of our engine on which INTDETECT is based. The starting point for the design of INTDETECT is the multi-threaded symbolic execution engine with backtracking

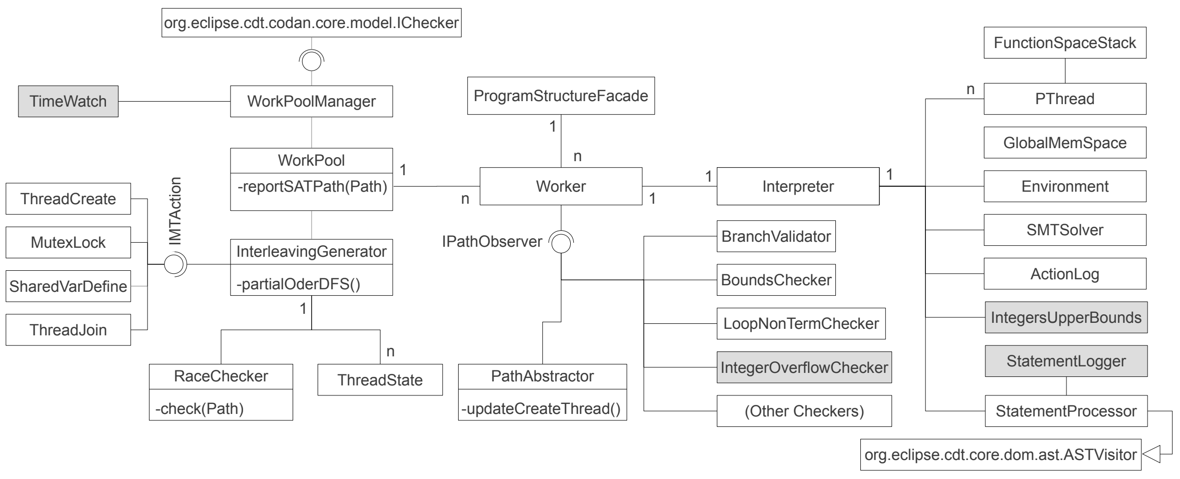


Figure 4.1: Main engine Java classes. These are also presented in more detail by Ibing [115].

support presented by Ibing *et al.* in [110], which can, analyze multi-threaded C programs, too. The engine is used to perform inter-procedural analysis and is implemented according to the tree-based interpreter pattern [217]. The engine implementation is multi-threaded, which means that control flow graphs and syntax trees are shared between worker threads. INTDETECT relies on a SMT solver as logic back-end and translates C code into SMT-Lib [16] logic equations in the logic of Arrays, Uninterpreted Functions, Non-linear Integer and Real Arithmetic (AUFNIRA). Figure 4.1 contains the main classes of our engine and the interface `IChecker`, which makes the plug-in usable from CDT's code analysis framework [138]. Several workers concurrently explore different parts of a program's execution tree. Each worker has an Interpreter together with a memory system model to store and retrieve symbolic variables (whose values are logic SMT-Lib equations). The translation of Control Flow Graph (CFG) nodes into SMT-Lib syntax is performed by the `StatementProcessor` (which extends CDT's abstract syntax tree visitor class) according to the visitor pattern [84]. The `BranchValidator` detects unsatisfiable branches in a program path with the help of the `SMTSolver`. `WorkPool` is used as synchronization object between the workers and the `WorkPoolManager`.

We briefly present the main classes of our engine denoted with capital letters. For a more detailed description see [115]. `WorkPoolManager` implements the interface `IChecker`, which is present in the Codan API. The `WorkPoolManager` starts workers and reports found errors through the `Codan` interface to the Eclipse marker framework. `ProgramStructureFacade` provides access to control flow graphs. `WorkPool` is used as synchronization object (synchronized methods) which is used to track the number of active workers and to exchange split paths. Each worker has a forward and a backward (backtracking) mode which passes references to control flow graph nodes for entry (forward mode) or backtracking to the Interpreter. The Interpreter follows the tree-based interpreter pattern, see Parr *et al.* [217] for more details. SMT syntax is generated by the StatementProcessor (which implements CDT's ASTVisitor) by bottom-up traversal of Abstract Syntax Tree (AST) sub-trees (visitor pattern), which are referenced by CFG nodes. Symbolic variables are stored in and retrieved from `MemSystem`. The interpreter further offers an interface to `BranchValidator` and to the checker classes. `SMTSolver` wraps the interface to the Z3 [68] external solver. `BranchValidator` is triggered when entering a branch node and generates a SMT-Lib query for the path constraint. For an unsatisfiable branch it throws an exception which is caught further on by the worker. The Environment provides symbolic models of standard library functions. `StatementProcessor` extends the `ASTVisitor` class contained in the Codan API. In this class, each statement contained on an execution path is visited in order to create our symbolic variables and the SMT constraint system which is attached to each symbolic variable. Among the *leave methods*—which visit each statement AST in bottom-up order—contained in the `StatementProcessor` we extended the functionality of the `IASTDeclaration`, `IASTDeclSpecifier` and `IASTDeclarator` methods. `BoundsChecker`, a buffer overflow checker which triggers on memory access with (symbolic) pointers, forms bounds violation satisfiability queries and reports buffer overflows, underflows, *etc.* `LoopNonTermChecker` is an infinite loop detection checker [113]. `FunctionSpaceStack` used to model the function space stack of symbolic functions. `GlobalMemSpace` modeling the global memory space of symbolic variables. `RaceChecker` is used for detecting race conditions [112].

The new classes which were added to support the detection of integer overflows in source code are shaded grey in Figure 4.1. The `IIntegerOverflowObserver` (for the sake of brevity not depicted in Figure 4.1) interface is extended to notify INTDETECT about integer overflows. The interface is implemented by `IntegerOverflowChecker`, which is described in detail in Figure 4.3. The `IntegerOverflowChecker` is used to trigger a bug report if an integer overflow has been detected. If a satisfiable path through the analyzed code is detected, additional checks are performed inside `IntegerOverflowChecker`. Further, if the resulted SMT-Lib system is still satisfiable, a bug report is generated indicating that an integer overflow error was detected.

Next, we will briefly describe the main features of our symbolic execution engine.

Unrestricted context depth. The symbolic execution engine supports Inter-procedural path-sensitive analysis with a call string approach [131, 243]. The path sensitivity is based on per function control flow graphs without in-lining. The function call context is represented by a program path leading to its call. The symbolic execution can be constrained regarding to how many times it should run by setting a context bound (*e.g.,* number of loop iterations, which in general incurs accuracy degradation) or it can be used unconstrained. This may lead to non-termination (*e.g.,* endless loops).

Finding relevant program paths. A fixed deterministic thread scheduling algorithm runs the symbolic execution. The algorithm depends on the thread identity numbers. Lowest Thread-ID First (LTIF) scheduling, which is based on scheduling one of the active threads having the lowest thread-ID first, is used. The symbolic execution is run with approximate path coverage which uses Depth-First Search (DFS). During DFS, program states are backtracked and branch decisions are changed [110]. The loop iteration bound can be configured either to prune a path until the loop iteration bound is reached, or to bypass the loop by avoiding the `BranchValidator` check.

Automatic slicing. In order to keep the equation systems for satisfiability checks small, only relevant logic equations are passed to the solver for a certain verification condition. This corresponds to automatic slicing [262] over the control flow (for separate analysis of different program paths) and over the data flow (for verification conditions on a program path).

Context sharing for different checkers. All checks can be performed with one enumeration of the satisfiable paths, and any specific checker is allowed to share the contexts because it is separated from the symbolic path interpretation. The checkers are allowed, through an interface, to register for notifications (triggers) and to query context equations. The symbolic interpreter is queried whenever triggered, in order to resolve the dependencies of the variables at the triggered location into the relevant equation system slice and adds the verification condition formula for a satisfiability query.

Logic representation. The SMT-Lib sub-logic of arrays, uninterpreted functions and non-linear integer and real arithmetic (AUFNIRA) has been chosen for using high-level logic that can decide automatically. With a target and a symbolic integer as offset formula, pointers are handled as symbolic pointers by the interpreter. They are outputted as logical formulas when dereferenced. Symbolic variables are created for the fields of composite data structures (*e.g.,* structs and in case of C++ also classes) and are not translated, but rather treated like scopes.

Path validation. `PathValidator` is triggered for branch nodes and uses the same interface as checkers. For all path decisions up to the current branch the `PathValidator` queries the equation SMT-lib linear system slice based on the resolution of the variable dependencies. Next, it adds a satisfiability check. The `PathValidator` throws a `PathUnsatException` if the solver answers unsatisfiable, which is caught by the `PathExplorer` (which reports the un-satisfiable path to the `PathValidator` and symbolic execution proceeds with the next path).

SMT Solving. The common Eclipse distributions come with a `SAT` solver plug-in [140], a SMT solver plug-in is unfortunately not (yet) available. Therefore, the SMT solver Z3 described in [68] is used. It is wrapped by the `SMTSolver` class and started as an external process.

Eclipse extension. The `WorkPoolManager` implements the Codan `IChecker` interface by plugging in the extension point `org.eclipse.cdt.codan.core.checkers`. While the available Codan checkers are normally configured to be *run as you type* or *run with build*, the symbolic execution engine is only *run on demand* with a GUI command, because of higher complexity and larger run-time of path-sensitive analysis. The plug-in further uses Codan `ControlFlowGraphBuilder` to generate CFGs for parts of an AST which are rooted in a function definition.

Posix threads support and path-sensitive tracing of shared variables. The symbolic execution engine offers the possibility to specify symbolic models of library functions, which are used both for the `C/C++` standard library and for the operating system (Posix threads). Relevant thread interactions are based on read accesses and write accesses to shared variables (usage or definition actions for variables). All global variables are marked as shared when they are first accessed. Then, the *shared* property is inferred over data flow constructs like assignments, references, function call parameters, and return values *etc.*

Implementation. IntDetect is notified from inside the `StatementProcessor` when an assignment statement is encountered. Further a symbolic variable which contained a symbolic variable name and symbolic type is sent in the notification. The Interpreter is notified by calling `ps.notifyLimitChecker(ini_ssa);`. Next, the notification is delegated by the Interpreter to the appropriate integer overflow checker which checks if there could be an integer overflow. The slice of equations on which the `ini_ssa` (this is a symbolic variable used to statically model the run-time variable x, $x := \mathit{expression}$) variable depends, is queried by the checker and adds one satisfiability check. The check is used to verify if the symbolic variable `ini_ssa` can be greater than the used integer upper bound value (the upper bound values are extracted from the `C` standard library *limits.h* file). If the solver answers `SAT` (satisfiable) to the query, then the problem is reported. The bug report contains the problem ID (unique system string), file name where the bug was detected and line number where the bug is located. In principle CWE-190—integer underflow (wrap or wrap-around) and CWE-192—integer coercion error are detectable.

Any number of checkers can be added and share the symbolic execution contexts. The Codan extension point supports the addition of new problems and problem detail views. Detected problems are reported to the marker framework with their Id, file name, line number and problem description. We added our path-sensitive integer overflow checker alongside other existent checkers (*e.g.,* RaceCondition checker, InfiniteLoop checker, *etc.*).

The gray shaded classes depicted in Figure 4.1 (`TimeWatch`, `IntegerOverflowChecker`, `IntegersUpperBounds`, and `StatementLogger`) were added, ($\approx$1400 SLoC). In total we added in `StatementProcessor` ($\approx$700 SLoC) which represents code used for AST statement traversing contained in the `leave()` methods. This is used for dealing with the new types of `C` statements contained in the analyzed programs. `IntegerOverflowChecker` (105 SLoC) is triggered for variable assignments present in the analyzed code. It generates satisfiability queries used to check for violation of integer overflows and reports an error in case of satisfiability. `IntegersUpperBounds` (42 SLoC) was used to extract the actual values for the integer upper bounds (platform dependent) from the standard `C` library file `limits.h` by taking into account the current CPU architecture (32-bit or 64-bit). This makes our approach platform-independent. TimeWatch (22 SLoC) is an utility class used for time measurements of our checker. `StatementLogger` (38 SLoC) utility class was used to log statements coming from leave() methods present inside the `StatementProcessor`.

The symbolic function models: `AbsModel()`, `SqrtModel()` and `RandModel()` were added to the Environment inside our engine (not depicted in Figure 4.1) in order to symbolically model the mathematical functions: abs(), sqrt() and rand(). The mathematical function `abs()` was symbolically remodeled by using the symbolic function model `AbsModel` (40 SLoC) inside our engine. We attached to the symbolic variable of `AbsModel(var_symbolic)`, a SMT-Lib constraint (it checks the numeric value of the `abs()` function parameter. If the parameter value is positive then the value will be not changed, else if the parameter value is negative then the `-` sign will be removed) which was used to model the mathematical modulo function. As symbolic return of the function model `AbsModel(var_symbolic)` we used a symbolic copy of `var_symbolic` which contained the previous attached SMT-Lib constraint. Next, we simulated the execution of the mathematical `sqrt` function call inside the function model `SqrtModel`, (54 SLoC), by attaching to the symbolic parameter variable of SqrtModel(param_symbolic) the value of the sqrt operation and assigning this to the symbolic variable, `param_symbolic`. We implemented our own `sqrt` function which can deal with big integers based on the `java.math.BigInteger`. Furthermore, we added the symbolic function model `RandModel()`, (29 SLoC), used to statically model the mathematical rand() function contained in the `C` standard library.

The symbolic function models: `SocketModel()`, `ListenModel()`, `ConnectModel()`, `RecvModel()`, `AcceptModel()` and `BindModel()` were used (not depicted in Figure 4.1) in order to symbolically model the (f) communication API functions: `socket()`, `listen()`, `connect()`, `recv()`, `accept()` and `bind()` declared in `winsock2.h`, `windows.h`, `direct.h`, `sys/types.h`, `sys/socket.h`, `netinet/in.h`, `arpa/inet.h` and `unistd.h`. Note, that for sake of brevity parameters are not indicated in the previous functions. The above mentioned functions were used inside `if` branch containing the `C` `stop;` statement inside the `then` branch. The analyzed `C/C++` programs contained `if` branch which were used to check if the return value of the function calls: are equal to `SOCKET_ERROR (-1)`. In case the return value was equal to `-1` then `stop;` was called on the `if` branch—an `else` branch was not present in the code. The symbolic return value of these functions would stop the symbolic program execution if it would be equal to the macro `SOCKET_ERROR -1` or not initialized with a value. Thus, making the code located after this function calls unreachable with respect to the program execution paths and thus, it will not be possible to detect the integer overflow bug. We attached to the symbolic return variables of the following function models: `SocketModel` (39 SLoC), `ListenModel` (39 SLoC), `ConnectModel` (41 SLoC), `RecvModel` (42 SLoC), `AcceptModel` (41 SLoC), and `BindModel`, (41SLoC) numeric values through the usage of SMT-Lib constraints. This way the part of the code where the bug was located could be reached. Note, that every numeric value can be used as symbolic return value of the functions, except `-1`. Furthermore, the function models `HtonsModel` (32 SLoC) and `Inet_addrModel` (32 SLoC) were added in order to model the `htons()` and `inet_addr()` functions. Each symbolical function model has a constructor method, `getName()`, `getSignature()` and an execute() method which makes the creation and usage of new function models straight forward.

4.4 Evaluation

4.4.1 Experiments Methodology

Test Programs. We tested IntDetect with the open source integer overflow test case CWE_-190_Inger_Overflow contained in the Juliet test suite [207]. The used test case contains 54 baseline programs with 48 Control Flow Variants (CFV) each resulting in total of 2592 analyzed programs and 2592 (†) true positives. Every baseline test case—48 CFV—contains 38 `C` programs and 10 `C` programs.

Automated Experiment Assessment. First, we generated 26 jUnit test classes containing 100 jUnit test methods in each of the first 25 classes and respectively 92 jUnit test methods in the 26th class. Second, we ran each of the generated classes separately—due to Eclipse run-time limitations it was not possible to put all 2,592 jUnit test methods in one class and run everything at once—by using the jUnit testing environment.

Setup. We tested IntDetect on the Eclipse IDE v. Kepler SR 1, OpenSUSE 13.1 OS, 64bit; 12 GB RAM, CPU Q9550 2.83GHz, 64-bit.

The focus of our experiments is to find out the number of false positives, false negatives, true positives (accuracy) and the runtime timings (efficiency). Our research questions are as follows.

- **RQ1:** How accurate is IntDetect w.r.t. the number of false positives, false negatives, and true positives? (Section 4.4.2)
- **RQ2:** How much analysis time requires IntDetect in order to accurately detect the faults? (Section 4.4.2)

4.4.2 Automated jUnit Test Cases Generation

A script was developed to generate the jUnit test methods for 2592 analyzed `C` programs. Our script requires the directory location—path—as parameter followed by the Juliet test case name for which the jUnt test cases should be generated (CWE_190 in our case). The script uses a predefined jUnit template method and generates 100 jUnit methods per Java class file. For 2592 programs we obtained 26 jUnit test classes containing 100 jUnit test methods each and 92 jUnit test methods in the 26th test class. The dynamic parameters, which are added in each Java source files, are: the line numbers where the bug is located, method names and the class names. These are automatically generated based on the test cases names detected in the selected Juliet test case. For the class names we divided the test programs in groups of 100 and named the Java class file as `CWE_` followed by the name of the 1st test program of the group of 100, followed by underscore `_` and the last test program name contained in the group. The first name of the file (*e.g.,* CWE...) contained in each test program was used as names for the generated jUnit test methods. The line number where the true positive is located was determined by tokenizing the source files contained in each test program and by searching for the string (p) `/* POTENTIAL FLAW */`—this string was inserted into the code by the Juliet test suite creators in order to mark the true positive and false positive locations. The script identifies as bug location the next line number after the string (p) was detected. Multiple appearances of (p) are filtered out by using several flags and counter variables used to count the number of appearances and the locations (line numbers) of (p).

4.4.3 Automated Eclipse C/C++ Programs Generation

A second script was created in order to generate 2592 Eclipse IDE projects containing all analyzed `C/C++` programs. The projects generation script uses the directory containing the Juliet test case (`CWE_190` in our case) as parameter in order to iterate recursively through all folders of the main directory and generates Eclipse `C/C++` projects with the required header files contained inside. Briefly, the script uses the Eclipse C/C++ hidden project files .cproject and .project as templates. These were next inserted into each generated project. In each generated project the script replaces the names of the project with the names extracted from the names of the source files contained in each Eclipse test case program. The script creates the project folder using the same name and puts all the required files for the current test project into the folder (needed header files were also copied inside the folder). The project names and the appropriate project configurations are written in the hidden project description files (.cproject and .project). Next, the C code line #define INCLUDEMAIN is added after the code line #ifdef INCLUDEMAIN which is contained in each Juliet test case program by default in order to have a starting point for the static analysis.

4.4.4 Experimental Results

Note, that each number [1, 54] located on the X axis of Figure 4.2 has two bars associated. Figure 4.2 depicts for each of the 54 test cases (each containing 48 CFV—in total 2592 C/C++ programs) the run-time in seconds indicated on the left Y axis (*e.g.,* the left bar in Figure 4.2 located on the X axis for #1) and the contribution of each CFV in % depicted on the right Y axis (*e.g.,* the right bar in Figure 4.2 located on the X axis for #1). The main contribution in all 54 test cases has CFV 12 depicted in Figure 4.2 with blue color ■ CFV 12. We observe that baselines (#12, #24, #27, #36, #48 and #54) depicted in Figure 4.2 and in Table 4.1, first column, have high execution times—more than 200 seconds—compared to the rest of the programs. In each of the expensive—have run-time over 200 seconds—test cases CFV 12 has more than 80% contribution to the INTDETECT's run-time. Furthermore, we measured the total execution time with respect to successful triggering, 3,638.122 seconds (3,666.36 seconds − 28.238 seconds) and the total execution time without successful triggering, 28.238 seconds (3,666,36 seconds − 3,638.122 seconds) and found out that 0.77% (28.238 seconds out of 3638,122 seconds) additional performance overhead was induced by the programs in which no execution exception was raised and no bug was found.

Figure 4.2 was split—depicted with dashed lines "¦" in Figure 4.2—from left to right having 12 test cases (24 bars) in each segment for the first 4 segments and 6 test cases (12 bars) in the last segment located at the far most right in Figure 4.2. with the goal to depict commonalities between each of the 5 obtained segments. We observe that the execution times are rising in each segment to a peak value (segment 1 (#2), segment 2 (#18), segment 3 (#27), segment 4 (#47) and segment 5 (#54)) and then they abruptly drop, except segment 5 where it increases continuously until it reaches the peak execution time (#54) for the last baseline test case. By dividing the whole execution time among all *the expensive* execution baselines, (#12, #24, #36, #27 and #48) for the first 4 segments and the #54 baseline test case we observe that those 6 baselines significantly dominated the whole execution time having execution times of more than 200 seconds. The run-time for the above mentioned test cases is higher compared to other baseline test cases because these programs contain the C standard library function calls rand() and sqrt() in multiple nested if conditions which make the path conditions to be

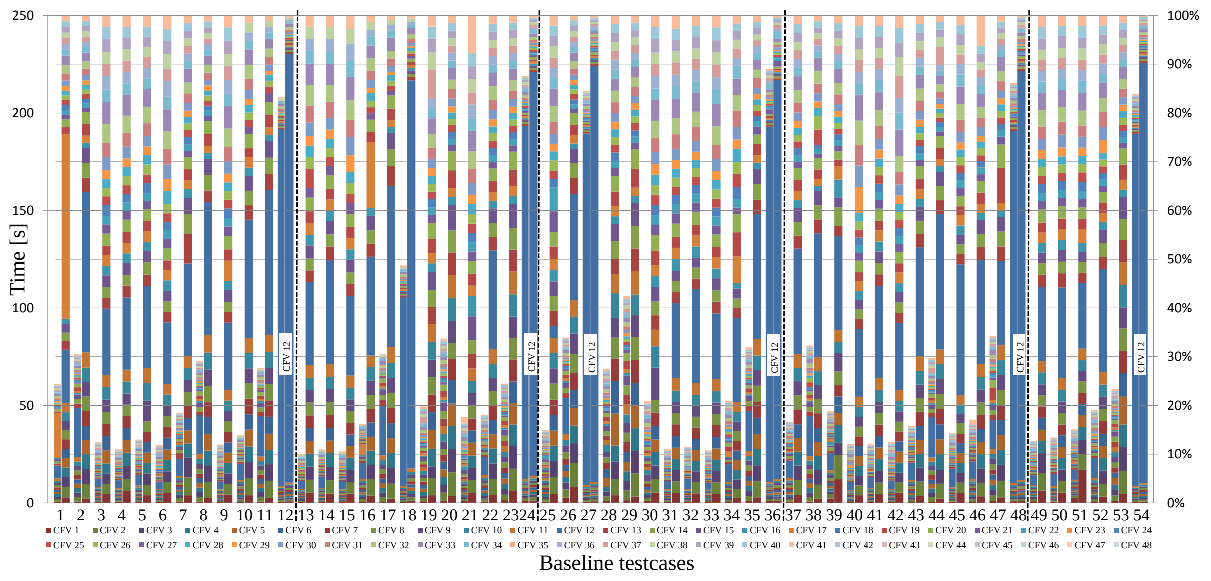


Figure 4.2: IntDetect run-time results for the Juliet's CWE-190 test case.

more complex than programs which do not contain such complex nested path conditions. Thus, incurring the additional computational overhead.

Table 4.1 contains the following abbreviations: Baseline Programs (BP, contains 48 CFV), Source Lines of Code (#SLoC), Total Bugs Triggered from 48 TP (TBT), Total Execution Time in seconds (TET [s]), Total Exceptions (TE), Exceptions with Trigger (0 NO/1 YES) (EwT), Total True Positives percentage w.r.t. 48 CFV and 37 CFV (48 CFV − 11 CFV, 10 C++ programs and 1 program containing the `C` `goto` statement) (TTP 48%/37%). Table 4.1 columns 5 and 8 depict IntDetect's run-time timings in seconds and true positive percentages for 37/48 CFV, respectively. False positives and false negatives are not depicted in Table 4.1 since they were not encountered during our experiment. Among the triggered bugs we have 100% success rate with respect to true positives. On the other hand, there were in total 11 (w.r.t. 2,592 programs) true positives detected where we got run-time exceptions but the bugs were still triggered correctly, Table 4.1 column seven. However, there were in total 43 (w.r.t. 2592 programs) programs where IntDetect successfully parsed the source code but failed to trigger any bug, see column seven contained in Table 4.1 for more details.

Table 4.2 contains the following abbreviation: % of Detected Bugs w.r.t. to the total number of true positives 2,592 (†), (% DB). Table 4.2 presents the results of IntDetect by running it on the 2592 programs. IntDetect triggered in total 1,908 true positives with a success rate of 73.61% DB in total as it can be observed in Table 4.2 columns 4 and 5. In terms of bug triggering for the 48 CFV for each base line test case our plug-in triggered at most 37 bugs in a single baseline which corresponds to a success rate of 48.53% DB while 27 being the lowest triggering number corresponding to 1.04% DB. The rest of the DB percentages are between 1.35% and 11.11% as depicted in Table 4.2, 5th column.

#	BP	#SLoC	TBT	TET [s]	TE	EwT	TTP 48%/37%
1	fscanf_add	5,233	37	14,857	11	0	77,08%/100%
2	char_fscanf_multiply	5,539	37	76,356	11	1	77,08%/100%
3	char_fscanf_square	5,309	37	31,213	11	0	77,08%/100%
4	char_max_add	5,236	37	27,566	11	0	77,08%/100%
5	char_max_multiply	5,541	37	32,54	11	0	77,08%/100%
6	char_max_square	5,309	37	29,631	11	0	77,08%/100%
7	char_rand_add	5,236	37	46,118	11	0	77,08%/100%
8	char_rand_multiply	5,541	37	72,947	11	1	77,08%/100%
9	char_rand_square	5,308	37	29,996	11	0	77,08%/100%
10	int64_t_fscanf_add	5,228	29	34,702	16	0	60,45%/78,37%
11	int64_t_fscanf_multiply	5,493	29	69,183	17	1	60,45%/78,37%
12	int64_t_fscanf_square	5,261	29	208,189	16	0	60,45%/78,37%
13	int64_t_max_add	5,188	29	25,117	16	0	60,45%/78,37%
14	int64_t_max_multiply	5,493	29	27,401	16	0	60,45%/78,37%
15	int64_t_max_square	5,261	27	26,483	16	0	43,75%/72,97%
16	int64_t_rand_add	5,188	29	40,497	16	0	60,45%/78,37%
17	int64_t_rand_multiply	5,493	29	76,312	17	1	60,45%/78,37%
18	int64_t_rand_square	5,261	29	121,849	16	0	60,45%/78,37%
19	int_connect_socket_add	12,168	36	49,439	12	0	75,00%/97,29%
20	int_connect_socket_multiply	12,473	36	84,226	12	0	75,00%/97,29%
21	int_connect_socket_square	12,241	36	44,115	12	0	75,00%/97,29%
22	int_fgets_add	6,377	37	45,184	11	0	77,08%/100 %
23	int_fgets_multiply	6,682	36	61,268	12	0	75,00%/97,29%
24	int_fgets_square	6,450	37	218,919	11	0	77,08%/100 %
25	int_fscanf_add	5,188	37	37,441	11	0	77,08%/100 %
26	int_fscanf_multiply	5,493	37	84,837	11	1	77,08%/100 %
27	int_fscanf_square	5,261	37	211,418	11	0	77,08%/100%
28	int_listen_socket_add	13,736	36	68,908	12	0	75,00%/97,29%
29	int_listen_socket_multiply	14,041	36	106,101	12	0	75,00%/97,29%
30	int_listen_socket_square	13,809	36	52,266	12	0	75,00%/97,29%
31	int_max_add	5,188	37	27,63	11	0	77,08%/100%
32	int_max_multiply	5,493	37	28,726	11	0	77,08%/100%
33	int_max_square	5,261	34	26,957	11	0	70,83%/91,89%
34	int_rand_add	5,188	37	52,185	11	1	77,08%/100%
35	int_rand_multiply	5,493	37	79,76	11	1	77,08%/100%
36	int_rand_square	5,261	37	222,582	11	0	77,08%/100%
37	short_fscanf_add	5,188	37	41,683	11	0	77,08%/100%
38	short_fscanf_multiply	5,493	37	80,723	11	1	77,08%/100%
39	short_fscanf_square	5,261	37	46,989	11	0	77,08%/100%
40	short_max_add	5,188	37	30,207	11	0	77,08%/100%
41	short_max_multiply	5,493	37	30,051	11	0	77,08%/100%
42	short_max_square	5,261	35	31,186	9	0	72,92%/94,59%
43	short_rand_add	5,188	37	39,372	11	0	77,08%/100%
44	short_rand_multiply	5,493	37	74,297	11	1	77,08%/100%
45	short_rand_square	5,261	37	35,885	11	0	77,08%/100%
46	unsigned_int_fscanf_add	5,188	37	42,742	11	0	77,08%/100%
47	unsigned_int_fscanf_multiply	5,493	37	85,65	11	0	77,08%/100%
48	unsigned_int_fscanf_square	5,261	37	215,401	11	0	77,08%/100%
49	unsigned_int_max_add	5,188	37	31,904	11	0	77,08%/100%
50	unsigned_int_max_multiply	5,493	37	33,569	11	0	77,08%/100%
51	unsigned_int_max_square	5,261	34	37,796	11	1	77,08%/91,89%
52	unsigned_int_rand_add	5,188	37	47,831	11	1	77,08%/100%
53	unsigned_int_rand_multiply	5,493	36	58,378	10	0	75,00%/97,29%
54	unsigned_int_rand_square	5,261	37	209,779	11	0	77.08%/100%
-	**Total**	**337,573**	**1,908**	**3,666.36**	**684,0**	**1(11) /0(43)**	**73,61% /95,49%**

Table 4.1: Bug detection results for CWE_190.

Table 4.3 shows the impact of expensive baselines test cases with respect to the total execution time, 3,666,36 seconds. The average execution time per baseline test case is 67.90 seconds

Triggered / 48 CFV	# of base-lines	Baseline #	# of bugs	% DB
37	34	[1, 9], 22, [24, 27], 31, 32, [34, 41], [43, 50], 52, 54	1,258	48.53%
36	8	19, 20, 21, 23, 28, 29, 30, 53	288	11.11%
35	1	42	35	1.35%
34	2	33, 51	68	2.62%
29	8	10, 11, 12, 13, 14, 16, 17, 18	232	8.95%
27	1	15	27	1.04%
Total			1,908	73.61%

Table 4.2: Integer overflows bugs triggered.

Execution Times	**T [s]**
Total time for 12th, 24th, 36th, 48th, 54th and 27th baselines	1,286.28
Average time for 12th, 54th and 27th	214.38
Execution time excluding higher value	2,380.07
Average execution time excluding higher value	49.58
Average execution time per baseline programs (48 CFV)	67.89
Total execution time	3,666.36

Table 4.3: Impact of expensive baseline test cases.

but if we exclude those six expensive baseline, then the average execution time decreases by 18.31 seconds to 49.58 seconds. In our experiments, we summed up the times where we had run-time exceptions with the times where we had no exceptions for all the programs where the bug was detected at the right place (true positive). Moreover, we included the execution times of the programs where the code was successfully parsed but no bug was found (no bug was triggered at all). We did not sum up the execution times for the test cases where we had exceptions and no bug was triggered.

Table 4.4 presents four types of exceptions that we have encountered during our experiments. The exceptions appeared because of current limitations of our framework. Note, that the ID numbers [101, 104]—used in Table 4.4 first column—were freely chosen to better depict the

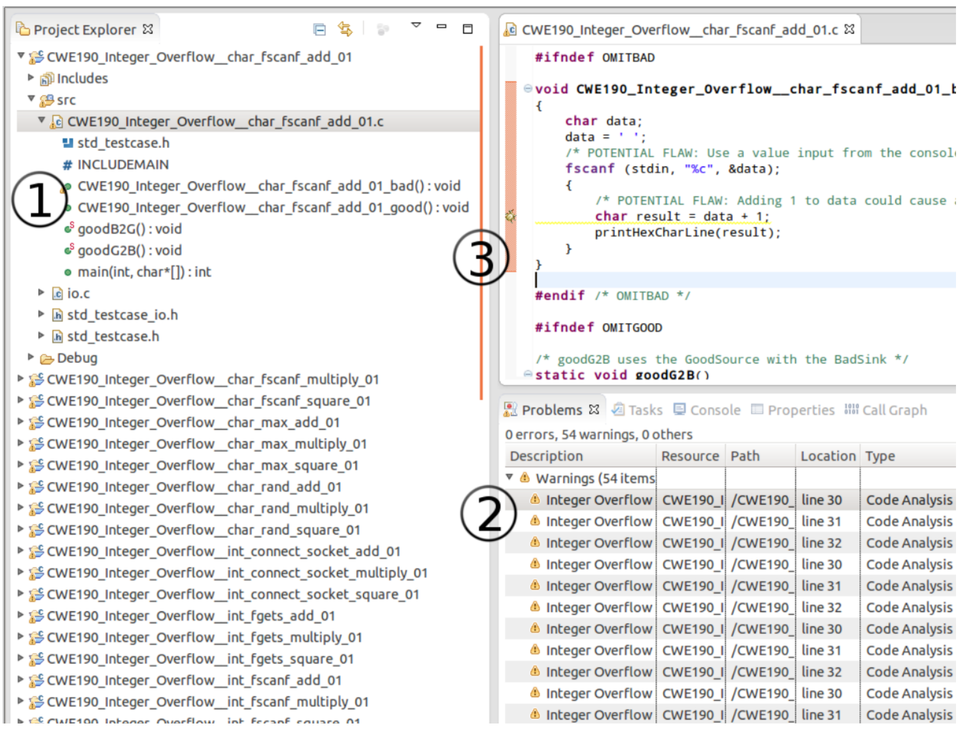
Exception ID	Error types
101	Jump node, `C` goto statement
102	java.lang.ClassCastException: org.eclipse.cdt.internal.core.dom.parser .cpp.CPPFunction cannot be cast to org.eclipse.cdt.internal .core.dom.parser.c.CFunction
103	java.lang.ClassCastException: org.eclipse.cdt.internal.core.dom.parser .cpp.CPPTemplateTypeParameter cannot be cast to org.eclipse .cdt.core.dom.ast.IBasicType
104	java.lang.NullPointerException smtcodan.interpreter.FunctionSpaceStack .enterFunction(FunctionSpaceStack.java:463)

Table 4.4: Types of exceptions encountered.

encountered exceptions. The exception `ID 101` was caused by the presence of the `C` language `goto` statement when present in the source code. INTDETECT triggered bugs in almost all programs except the `C++` programs and the programs containing the `goto` statement. `ID 102` was encountered because our framework can not currently deal with `C++` functions, only `C` functions. `ID 103` appeared when trying to convert a `C++` template type parameter into an `IBasicType` which were not compatible. The exception, `ID 104`, was encountered inside the method `enterFunction(SymFunctionCall nextCall, SymFctSignature fsign)` which resides in `FunctionSpaceStack`. The reason is that we do not create symbolic function signatures for `C++` function declarations. This limitations will be removed by making the CFG builder algorithm aware of object instantiations and other `C++` specific language semantics.

Figure 4.3 presents the bug reports after running INTDETECT on the 54 baseline programs (each contains one CFV). The circled numbers in Figure 4.3 represent: number ① depicts the inner structure of the first baseline program, number ② contains 54 bug reports for the 54 true positives contained in the 54 baseline programs (each contains the first CFV) and number ③ indicates the bug location (file name and line number) of the first bug report indicated in the Eclipse "Problems" view. Furthermore, the user has the possibility to trace back the detected bug—for the programs where bug reports were generated—to the bug location (file name and line number) by double-clicking on one of the bug reports depicted in Figure 4.3 with number ②.

However, by excluding all `C++` programs and those programs containing the `C` language `goto` statement (no bug reports were generated and contain 11 CFV per baseline test case) a total of 594 (11 $\times$ 54), 22.9% (594 out of 2592) test programs can be excluded meaning that only 1998 (2592 $-$ 594) programs are actually analyzable by our framework. Among 1998 programs containing the same number of bugs (true positives) we successfully detected the bugs in 1908 programs (true positives), which results in a successful coverage of 95.49% with no false positives. In the rest 90 programs (1998 $-$ 1908), 4.51% from the total analyzable programs (1998), INTDETECT did not trigger any true or false positives. Thus, the above results confirm that our approach is accurate and effective in detecting integer overflows.



```
CWE190_Integer_Overflow__char_fscanf_add_01
  Includes
  src
    CWE190_Integer_Overflow__char_fscanf_add_01.c
      std_testcase.h
      # INCLUDEMAIN
      CWE190_Integer_Overflow__char_fscanf_add_01_bad() : void
      CWE190_Integer_Overflow__char_fscanf_add_01_good() : void
      goodB2G() : void
      goodG2B() : void
      main(int, char*[]) : int
    io.c
    std_testcase_io.h
    std_testcase.h
  Debug
CWE190_Integer_Overflow__char_fscanf_multiply_01
CWE190_Integer_Overflow__char_fscanf_square_01
CWE190_Integer_Overflow__char_max_add_01
CWE190_Integer_Overflow__char_max_multiply_01
CWE190_Integer_Overflow__char_max_square_01
CWE190_Integer_Overflow__char_rand_add_01
CWE190_Integer_Overflow__char_rand_multiply_01
CWE190_Integer_Overflow__char_rand_square_01
CWE190_Integer_Overflow__int_connect_socket_add_01
CWE190_Integer_Overflow__int_connect_socket_multiply_01
CWE190_Integer_Overflow__int_connect_socket_square_01
CWE190_Integer_Overflow__int_fgets_add_01
CWE190_Integer_Overflow__int_fgets_multiply_01
CWE190_Integer_Overflow__int_fgets_square_01
CWE190_Integer_Overflow__int_fscanf_add_01
CWE190_Integer_Overflow__int_fscanf_multiply_01
```

CWE190_Integer_Overflow__char_fscanf_add_01.c

```
#ifndef OMITBAD

void CWE190_Integer_Overflow__char_fscanf_add_01_
{
    char data;
    data = ' ';
    /* POTENTIAL FLAW: Use a value input from the consol
    fscanf (stdin, "%c", &data);
    {
        /* POTENTIAL FLAW: Adding 1 to data could cause
        char result = data + 1;
        printHexCharLine(result);
    }
}

#endif /* OMITBAD */

#ifndef OMITGOOD

/* goodG2B uses the GoodSource with the BadSink */
static void goodG2B()
```

0 errors, 54 warnings, 0 others

Description	Resource	Path	Location	Type
Warnings (54 items				
Integer Overflow	CWE190_I	/CWE190_	line 30	Code Analysis
Integer Overflow	CWE190_I	/CWE190_	line 31	Code Analysis
Integer Overflow	CWE190_I	/CWE190_	line 32	Code Analysis
Integer Overflow	CWE190_I	/CWE190_	line 30	Code Analysis
Integer Overflow	CWE190_I	/CWE190_	line 31	Code Analysis
Integer Overflow	CWE190_I	/CWE190_	line 32	Code Analysis
Integer Overflow	CWE190_I	/CWE190_	line 30	Code Analysis
Integer Overflow	CWE190_I	/CWE190_	line 31	Code Analysis
Integer Overflow	CWE190_I	/CWE190_	line 32	Code Analysis
Integer Overflow	CWE190_I	/CWE190_	line 30	Code Analysis
Integer Overflow	CWE190_I	/CWE190_	line 31	Code Analysis

Figure 4.3: CWE_190 baseline programs bug reports.

4.5 Discussion

In this Section, we discuss internal and external threats to validity.

4.5.0.1 Threats to Validity

Internal Validity. The experimental results presented in Table 4.1 may not support our findings for several reasons. If IntDetect incorrectly skips some bugs or misses some constraints, then it might report false positives (we did not encounter any in our experiment). For this reason we carefully extended the `StatementProcessor` in order to be capable to deal with the new `C` language semantics present in `CWE_190`. If we misinterpreted the timing results, then the potential total execution time would not be feasible. We addressed these factors by testing our tool extensively on the open source programs contained in `CWE_190` which have a known number of integer overflow bugs. We repeated each run three times for each of the analyzed programs in order to be sure that the obtained results are right.

External Validity. The results cannot be generalized for `C++` programs since the open source programs contained in `CWE_190` were not analyzed due to current engine limitations. Our engine does not support `C++` language semantics due to the fact that we currently only focused on the `C` language. We mitigate this issue by extending the `StatementProcessor` to be capable to translate every kind of `C++` language semantics in SMT-Lib statement is just a matter of time. On the other hand the current CFG builder algorithm is not aware of `C++` control flow related semantics (*e.g.,* object instantiation, *etc.*). The required C++ data types are available in the Codan API and will be implemented into our CFG builder algorithm so that the CFG will contain program execution paths based on specific C++ language control flow semantics.

4.6 Summary

In this Chapter, we presented IntDetect which is a static integer overflow detection tool that runs on C source code and can detect integer overflows with a true positives rate of 95,49% with no false positives. Our checker was developed as an Eclipse plug-in and was used to automatically detect bugs in 2,592 programs. We used jUnit tests in order to automate the assessment of the accuracy of our integer overflow checker. Furthermore, we demonstrated that an integer overflow checker with low false positives rate can be built by formal modeling of C language semantics which are related to integer overflows and with function models which are used to simulate the program environment.

Chapter 5

IntRepair: Static Repairing of Integer Overflow Based Memory Corruptions

In this Chapter, which is part of the first part of this thesis, we extend our static `C` source code analysis framework by presenting an integer overflow detection and repair tool which can provide source code repairs, that help a programmer to automatically repair an integer overflow. Within this Chapter, we address **RQ2** by providing an integer overflow repair tool, named INTREPAIR, which is capable to efficiently remove a fault, can automatically validate a repair and does not introduce unwanted program behavior. Finally, note that parts of this Chapter have already been published by Muntean *et al.* [197].

5.1 Introduction

What is the overall view? Integer overflows are a well-known cause of memory corruptions, and a widely known type of vulnerability [272] that has threatened programs for decades. To address this challenge, it has been, for example, proposed to apply the concept of automatic repair [191] to fix integer overflows. This idea consists of using a system that produces code patches in order to fix integer overflows. A promising research thread consists of automatically repairing integer overflows, with notable approaches such as SoupInt [154], CodePhage [247], TAP [249], and Sift [154]. However, those approaches suffer from the main limitation that they do not assess correctness after modification.

What is the problem? Generating a correct repair for an integer overflow bug is a difficult task. First, the search space of all possible repairs is large. Second, the correctness assessment must go beyond the paths dynamically executed. In this thesis, we propose a solution to this problem. We devise an efficient repair method for integer overflows based on symbolic analysis. The prototype implementation, called INTREPAIR, generates correct repairs at source code level.

What are the exiting solutions? The main solutions, which do not rely on manually generated test cases are as follows. SoupInt [270] (runtime based binary patch generator) do not produce source code patches, CodePhage [247] (binary based horizontal code transfer) does not check for correctness beyond the failing input, and Sift [154] (static sound input filter generation for binaries) relies on test cases only.

What are the limitations of the solution? However, these integer overflow repair tools either: (1) introduce a high runtime overhead after the program was repaired, or (2) do not validate whether the generated patches are correct and program behavior is not changed in an unwanted manner.

What is our our insight? In this Chapter, we tackle these two problems in a novel system, INTREPAIR, usable for automatic repair of integer overflows. Our key idea is to use symbolic execution [34, 272, 35, 11, 159] in order to reason about the repair. In particular, our intuition is two-fold: (1) we fuse the fault localization and repair generation phases into a single algorithm via SMT solving, and (2) we ensure that the resulting repairs do not insert unwanted program behavior with symbolic reasoning. We present a novel integer overflow repair technique that operationalizes this idea and the corresponding prototype tool, INTREPAIR. To the best of our knowledge, INTREPAIR is the first approach for automated repair of integer overflow that does not require test cases.

How does it work? Given a program containing a defect, INTREPAIR generates a SMT system that captures integer overflow manifestation and allows for correct patch generation and validation. This is achieved by interweaving the bug detection SMT constraints with newly synthesized constraints used for correct bug generation. *Patch Correctness:* The repairs generated by INTREPAIR are correct in the sense that the repair correctly removes the bug and does not change program behavior. *Patch Validation:* By validation, we mean that the generated synthesized patches are guaranteed to be correct. As such: (1) the automatically generated patches are not program input dependent, (2) the generated patch removes the detected integer overflow on all program execution paths that reach the fault location, and (3) the patches do not introduce unwanted program behavior (*i.e.,* only the integer overflow is removed). The integer overflow detection checker on which INTREPAIR relies does not generate false negatives (*i.e.,* every detected integer overflow is a genuine integer overflow). Further, we do not distinguish between intended or unintended integer overflows, yet false positives may happen due to certain implementation limitations, such as loop unrolling or recursion depth.

The implementation of INTREPAIR is built upon the Codan static symbolic execution engine [196, 113, 114]. It uses the Z3 SMT solver [68] to solve the extracted constraints. Note that our repair generation technique is general enough to be implemented on top of other symbolic execution engines as well.

We evaluated INTREPAIR on 2,052 `C` programs contained in the SAMATE's `C/C++` benchmark suite [264], totaling more than one million lines of code (LOC). The evaluated programs contain all possible program control flows that may lead to an integer overflow. Further, we use a synthesized benchmark containing 50 large `C` programs (up to 20 KLOC) seeded with integer overflows. Our experimental results show that INTREPAIR is able to: (1) effectively detect all integer overflows, and (2) successfully repair the programs at source code level. In addition, we present the results of a user study with 30 participants showing that INTREPAIR is effective compared to traditional manual repair. To the best of our knowledge, there are no other open source tools with which we can compare against. In particular, DirectFix [161], Angelix [162], or SemFix [205] are not suitable for comparison purposes as these tools use test cases to locate the fault: on the contrary, we do not assume the presence of test cases.

What are our contributions? In summary, in this Chapter, we make the following contributions:

- We designed a novel source code repair generation technique for integer overflows in `C` programs.
- We implemented INTREPAIR, as a prototype of our novel integer overflow repairing technique, for `C` source code programs. It can be automatically used to repair integer overflows across multiple integer precision.
- We evaluated INTREPAIR thoroughly with 2,052 `C` programs contained in the currently largest open-source test suite for `C/C++` source code (NSA's Juliet) and with 50 synthesized programs which range up to 20 KLOC. We show that INTREPAIR's code repairs are effective and induce low runtime overhead.
- We evaluated INTREPAIR within a controlled experiment with 30 participants and determined that our tool is more time-effective than repairing the same programs manually.

5.2 Threat Model

Defensive Assumptions. We align our threat model (*e.g.,* STRIDE [165]) to the general integer overflow threat model. We assume that ALSR [219] and DEP [164] are in-place and correctly functioning. We assume that the code is not compiled with any integer overflow bounds checking. Lastly, we assume that the defender has access to the source code of the application and potentially knows how to repair the integer overflow.

Attacker Capabilities. We assume a skilled attacker with sufficient resources and time to exploit the integer overflow. Further the attacker can use the integer overflow to escalate privileges or to manipulate the program counter according to his desires. The attacker is equipped with the necessary tooling to find the integer overflow in the program binary and to experiment with the code in an live interactive debugging session. Next, we don not exclude the possibility that the attacker has access to the source code of the application. Lastly, the attacker is aware of the existence of other vulnerabilities (*e.g.,* buffer overflow) which might be triggered through the previously found integer overflow. This requirement is not mandatory but a nice to have as opposed to buffer overflows, integer overflows are exploited indirectly through integer overflow to buffer overflow (*i.e.,* IO2BO) vulnerabilities.

5.3 Design and Implementation

In Section 5.3.1, we present the architecture of INTREPAIR, and in Section 5.3.2, we present the supported overflow and underflow checks, while in Section 5.3.3, we introduce INTREPAIR's fault localization algorithm. In Section 5.3.4, we depict several of the repair patterns of INTREPAIR, and in Section 5.3.5, we present the repair generation algorithm, while in Section 5.3.6, we highlight implementation details. Finally, in Section 5.3.7, we describe how INTREPAIR was integrated and can be used inside an IDE.

5.3.1 Overview

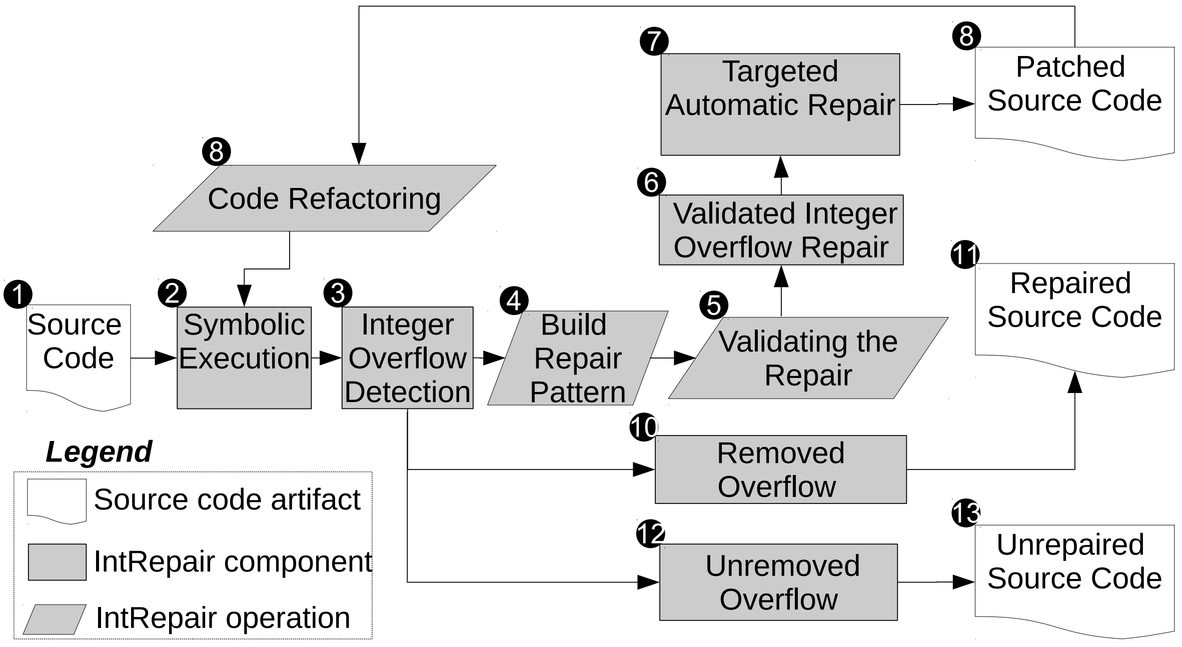


Figure 5.1: INTREPAIR depicted as gray shaded boxes.

Figure 5.1 provides an overview of INTREPAIR showing its main components. Initially, the ❶ *source code* is passed into the ❷ *symbolic execution* engine, which first constructs a CFG and then extracts all program execution paths. Constraints are collected along these paths and ❸ the *integer overflow detection* component identifies whether an integer overflow might occur for a given execution path. In ❹ a repair pattern is selected to fix the overflow. The ❺ *symbolic repair validation* component checks whether the negated constraints invalidate this integer overflow error. If this is the case, ❻ the *validated integer overflow repair* is validated and a ❼ *targeted automatic repair* is generated that removes the previously detected integer overflow. After applying the repair, we obtain the ❽ *patched source code*, which is again validated through ❾ *refactored code validation*. Further, in case ❸ *integer overflow detection* does not find an overflow, INTREPAIR produces ❿ a *removed overflow*, indicating that the bug was successfully removed and ⓫ *repaired source code* has been synthesized. Finally, in case the integer overflow was not removed, then INTREPAIR produces ⓬ an *unremoved overflow*, and the result is ⓭ *unrepaired source code*. Note that INTREPAIR checks for the introduction of unwanted behavior after a patch was inserted by re-analyzing the program with the help of symbolic execution and deciding that no new bug was inserted. For verifying a patch after it was applied, INTREPAIR does not require any test case since the integer overflow component can decide if the bug was removed. However, a test case *can* be used in conjunction with the fault detection component in order to confirm that the bug was removed from the source code location where it was first detected before patching.

5.3.2 IntRepair Overflow and Underflow Checks

In the following, we present the three types of overflow and underflow checks that are supported by INTREPAIR. The checks are preconditions, as shown below. If one of the preconditions evaluates to true, then an integer overflow has been found [73].

First precondition. The addition of two integers, in which one is a variable and the other is a positive constant, will lead to an integer overflow if the following expression evaluates to true: $((s_1 > 0) \wedge (s_1 > (\text{INT\_MAX} - s_2)))$. Note that s_2 is the positive constant.

Second precondition. The multiplication of two integers, in which one is a variable and the other is negative, will lead to an integer overflow or underflow if the following expression evaluates to true: $((s_1 > 0) \wedge (s_1 > (\text{INT\_MIN}/s_2))) \vee ((s_1 < 0) \wedge (s_1 < (\text{INT\_MAX}/s_2)))$. Note that s_2 is the negative constant.

Third precondition. The multiplication of two equal integers will lead to an integer overflow or underflow if the following expression evaluates to true: $((s_1 > 0) \wedge (s_1 > (\text{sqrt(INT\_MAX)})))$ $\vee$ $((s_1 < 0) \wedge (s_1 < (-\text{sqrt(INT\_MAX)})))$. Note that s_1 and s_2 are the same variable.

The above preconditions are used over multiple types of inputs for s_1 or s_2 (*i.e.,* `RAND32()`, this is the wrapper for the `C` random function, `fscanf()` *etc.*). The preconditions can be applied over multiple integer precisions, meaning that INT_MAX can take different values depending on the currently used integer precision in the analyzed program. We call this value the maximum admissible upper bound value of an integer. This value is determined automatically during program analysis. Further, the variables s_1 or s_2 can take different types: `char`, `int64_t`, `int`, `short`, `unsigned int`. In contrast to IOC's preconditions, which aim to avoid an unconfirmed integer overflow during program runtime, our preconditions help to guide repairs at the right code location, where the integer overflow was detected and confirmed. This results in avoiding the bug during runtime. Note that we always use symbolic execution again as a last step to confirm that the bug was completely fixed after a repair was inserted.

Other operations, which may lead to integer overflows, are: truncation, bit shifts and subtraction operations (see Figure 6 in Pereira *et al.* [222] for more details). Currently, INTREPAIR does not support these operations, but we plan to support them in an updated version of INTREPAIR.

5.3.3 Fault Localization

In order to generate code repairs, IntRepair initially needs to detect the precise location where the integer overflow resides in the program. This is the goal of IntRepair's repair location search algorithm, which we now present. First, IntRepair constructs the CFG of the analyzed program. Next, the following steps are performed consecutively in order to detect a fault.

1. Each program execution path is extracted from the previously generated program CFG.
2. The extracted path is traversed and path satisfiability checks are performed at branch nodes.
3. When IntRepair encounters an integer error prone code location (*i.e.,* assignment statement) on the analyzed path, an integer overflow check is performed by notifying the interpreter.
4. The notification is delegated to the appropriate checker (*i.e.,* integer overflow checker) by the interpreter.
5. The slice of SMT equations of the symbolic variable, which potentially may overflow, together with corresponding integer overflow satisfiability checks is queried by the integer overflow checker.
6. The check verifies if the symbolic variable, which caused the integer overflow, can be greater (*i.e.,* if this is true then there is an integer overflow) than the currently used integer upper bound value (*i.e.,* `INT_MAX`). These upper bound values are extracted from the `C` standard library contained in the `limits.h` file. The lower bound is obtained by negating the currently used upper bound value and and adding one to the result.
7. In case the SMT solver replies `SAT` (satisfiable, integer overflow bug present) to the previously submitted SMT query, then a problem report with problem `ID` (unique system string, the `ID` refers to which checker detected the bug), the file name where the bug was detected, and the line number in the file where the bug is located will be created and stored.

5.3.4 Repair Patterns

In this Section, we present the repair patterns available in IntRepair. We explain how these patterns can help to repair integer overflows and how the patterns can produce correct fixes. Repair patterns are stored in a decision tree. However, in the following graphics we will depict the criteria needed to choose a specific repair pattern instead of the pattern itself. Before the bug repair analysis is started, the programmer manually constructs such a decision tree. This is stored in a tree data structure for later usage. The rationale behind the decision tree is help IntRepair to perform a time efficient search for each detected fault and to pick a core repair that fits best.

Figure 5.2 depicts a decision tree used to show repair patterns available to IntRepair. The series of dots indicate other operations (*i.e.,* subtraction, bit shift, *etc.*) or arrows not depicted

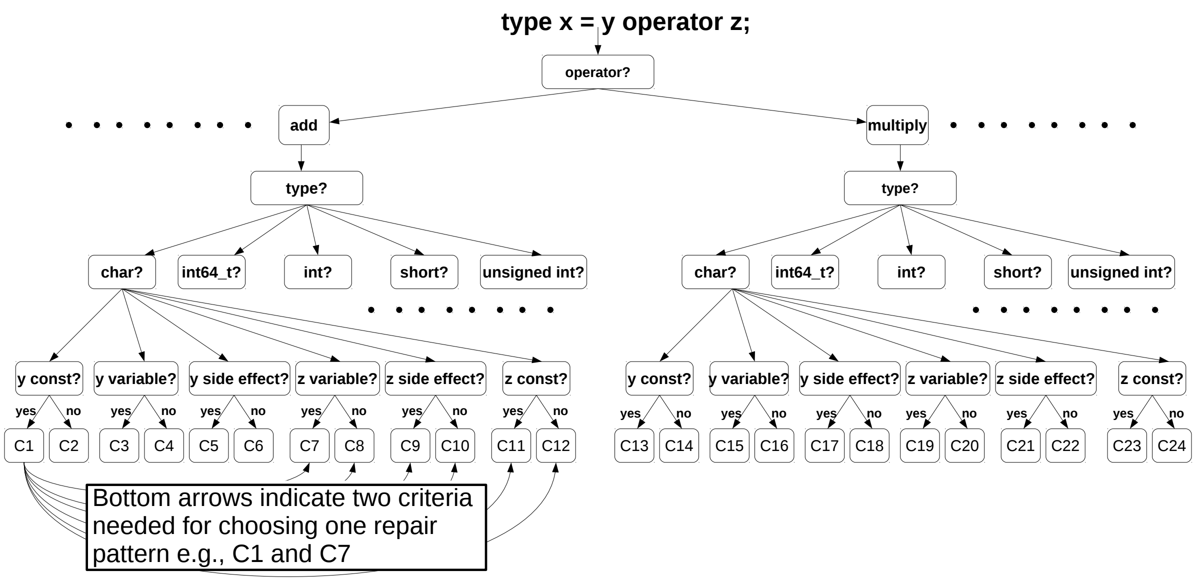

type x = y operator z;
operator?
add
multiply
type?
char?
int64_t?
int?
short?
unsigned int?
y const?
y variable?
y side effect?
z variable?
z side effect?
z const?
yes
no
C1 C2 C3 C4 C5 C6 C7 C8 C9 C10 C11 C12
C13 C14 C15 C16 C17 C18 C19 C20 C21 C22 C23 C24
Bottom arrows indicate two criteria needed for choosing one repair pattern e.g., C1 and C7

Figure 5.2: Decision tree used for storing repair patterns.

in Figure 5.2. Other repair patterns for `int64_t`, `int`, `short`, and `unsigned int` types are not depicted due to space limitations. *C*1 up to *C*24 represent the criteria needed by IntRepair for choosing a repair pattern. A repair is selected when at least two criteria are met. The six arrows on the left bottom pointing up depict six repair patterns determined between *C*1 and one of the criteria from *C*7 − *C*12. Depending on whether one of the components of the analyzed statement `type x = y operator z;` is a constant, a variable or a variable with side effects (*e.g.,* `int z = i++;` or `short y = foo();`) a different repair pattern will be proposed.

More specifically, from top to bottom, the decision tree contains an addition (left node) and a multiplication (right node) operation that are used to determine the kind of operation, which is performed in the buggy statement. At the next level in the tree (top to bottom), the type of result variable is determined. Further, at the following level in the tree, it is determined if the parameters of the analyzed statement are constants, variables or have side effects. Figure 5.2 depicts in total 360 repair patterns (360 possibilities to combine two criteria from the range *C*1 to *C*12 with each other, *e.g.,* *C*1 and *C*8, see corresponding arrow in Figure 5.2) multiplied by five (data types) and the result is then multiplied by two, two main tree branches, *e.g.,* add and multiply) for the `C` statement `type x = y operand z;`.

Table 5.1 depicts four notable repair patterns in more detail. Note that in Table 5.1 we omit the contents of the *if* and *else* branches in order to keep the graphic readable. The *if* branch contains the code to log the occurrence of an integer overflow or the code which terminates the execution of the program[1]. The *else* branch contains a statement where the integer overflow was detected.

[1]Note that terminating the program is an option that is not safe in critical program executions, *i.e.,* in case monetary transactions depend on the program execution or when the program is driving a train, *etc.*

Criteria	Statement	Repair Pattern Format	Description

C15 & C19	`char a=y*y;`	`if(a > sqrt(INT_MAX) ||`
		`a < -sqrt(INT_MAX)){log_or_die();}`
		`else{...}`
			multiply two equal variables

C15 & C17	`char a=y*3;`	`if(a > INT_MAX/2 ||`
		`a < INT_MIN/2){log_or_die();}`
		`else{...}`
			multiply a variable with a constant

C3 & C7	`char a=y+z;`	`if(a > INT_MAX/2 ||`
		`a < INT_MIN/2){log_or_die();}`
		`else{...}`
			add two variables

C3 & C11	`char a=y+4;`	`if(a > INT_MAX-2){log_or_die();}`
		`else{...}`
			add a variable to a constant

Table 5.1: Four repair patterns of IntRepair.

Consider the `C` statement at line number 544. $a = b*b$; where an integer overflow was detected at line number 544 in a program's source code file. In this case, the pattern depicted in Table 5.1 on row one ($C15$ & $C19$) will be used to avoid the bug. This pattern will be selected based on the fact that two equal variables are multiplied. As a result, the statement will be surrounded with the code of the pattern in which parts are replaced based on the type of statement, 543. $if(a <= sqrt(INT\_MAX) \ || \ a < -sqrt(INT\_MAX))$ $\{$545. $log\_or\_die();\}$ 546. $else\{ \ a = b*b;\}$. The numbers (*i.e.,* 543 up to 546) used in this example represent the source code line numbers of the repaired program, respectively.

How does a repair help to fix a bug? A generated repair helps to fix a fault by providing to the programmer a ready to use code snippet which contains complex mathematical constraints, that are not always obvious and in some situations difficult to manually infer.

Why are the produced repairs correct? A produced repair is correct w.r.t. to the definition of repair correctness given in the Section 5.1, since after the repair generation and insertion IntRepair checks if the fault was correctly removed and if the program behavior was changed. Additionally, after repair insertion, the whole program together with the repair is recompiled; thus, semantic errors potentially caused by the patch are in this way excluded.

Finally, note that building a repair is not straightforward, as the repair construction relies on complex mathematical constraints that need to be inferred based on the currently detected fault. Further, these constrains need to be carefully plugged into the selected repair pattern in order to provide the ready to use patch. Next, we will present the algorithm used to generate these mathematical constraints and the repair.

5.3.5 Integer Overflow Repair Algorithm

The automated repair generation algorithm of IntRepair consists of the following steps.

1. Determine the integer upper bound value;

2. Generate a SMT constraint system;
3. Select constraint values;
4. Re-compute the bound checking constraints;
5. Determine the bug type;
6. Select the repair pattern;
7. Determine the new constraint SMT system;
8. Generate code repair;

The algorithm corresponds to the box ❹ depicted in Figure 5.1. These eight steps are performed in consecutive order as highlighted next.

5.3.5.1 Determine Integer Upper Bound Value

In order to determine the currently used integer upper bound value in the analyzed program, IntRepair performs the following procedure, which allows IntRepair to be a multi-precision tool in the sense that it automatically determines the integer precision needed for each analyzed program. First, IntRepair retrieves the hardware overflow limits[2] for each integer type. Second, during the symbolic execution-based traversal of the analyzed program path, IntRepair searches for previously defined and used integer upper bound program variables. This search is realized by comparing each declared or used variable name (*e.g.,* `data` in the code snippet) contained in the currently analyzed program execution path with one of the supported integer upper bound values (*i.e.,* `CHAR_MAX`, `INT_MAX`, `LLONG_MAX`, `SHORT_MAX`, and `UINT_MAX`). Third, in case such an upper bound value is found, it will be set to be the currently used integer upper bound value. IntRepair can automatically detect for each environment (system), in which it runs, the currently used integer overflow upper bound limit (*i.e.,* `INT_MAX`) value. This way, the integer precision is determined for each analyzed program individually. Next, this upper bound value will be used for validating generated candidate code repairs and also to search for integer overflows.

5.3.5.2 Generate a SMT Constraint System

Next, the symbolic variables and the constraints used inside the integer overflow checker are grouped together and stored for later processing. In particular, IntRepair stores: (1) the statement where an integer overflow is detected, (2) the SMT formula used to detect the bug in the first place, (3) the bug ID of the integer overflow checker which was used to detect the bug, (4) the symbolic variable which was used to detect the integer overflow, and (5) other symbolic variables on which the integer overflow triggering variable may depend.

[2] *e.g.,* by parsing `/usr/include/limits.h` on Linux OS.

For instance, let us consider a `C` language assignment statement `int result = varA + varB;` and its SMT counterpart `(assert (= resSymbolic ( + varAsymbolic varBsymbolic)))`. The `C` assignment statement depicts the addition of two variables and the result is stored in a third variable, with a potential integer overflow bug. For this statement, the information collected by INTREPAIR is: (1) `int result = varA + varB;`, (2) `... (assert (= resSymbolic ( + varAsymbolic varBsymbolic))); ... checksat`, note that this SMT constraint represents a part of the SMT system used to detect the bug, (3) `ID-Integer_-Overflow_Bug`, (4) `resSymbolic`, and (5) and the `varAsymbolic` variable.

5.3.5.3 Select Constraint Values

INTREPAIR selects the relevant SMT constraint variables based on the type of `C` language statement where the integer overflow is detected. For example, consider assignment $x = a1 + a2$; where the variable x and potentially the variables $a1$ and $a2$ need to be taken into consideration, because these directly influence the occurrence of the integer overflow.

The symbolic variable, *i.e.,* `resSymbolic`, was selected by INTREPAIR to be further constrained. This is done in order to check if the code repair that will be generated could remove the previously detected integer overflow bug. For bug removal checking, the whole SMT constraint system slice of this symbolic variable will be used. Note that this constraint system was previously (before running the bug removal analysis) determined during fault localization. This variable will overflow in case of using too large values that cannot properly be stored in the result variable. Note that depending on the complexity of the analyzed statement, more or fewer variables can be taken into consideration in order to determine if the previously detected integer overflow would further manifest after re-constraining. This depends on how the selected symbolic variables were re-constrained. The intuition is that the integer overflow can be detected before a certain variable will overflow. This type of behavior is useful when generating repairs that are constraining more than one program variable. Finally, the selected variables will be used in the next step when re-constraining the bounds during the checking of SMT constraints of the SMT system. Note that this SMT system was used upfront to detect the integer overflow.

5.3.5.4 Re-Compute the Bound Checking Constraints

In order to solve the problem of determining a suitable variable range for avoiding an overflow, INTREPAIR applies a specific technique for re-constraining the symbolic variable which has overflown and which was selected in the previous step. The variable can be re-constrained after collecting the program execution path constraints for a single program execution path, which were used to check for the presence of an integer overflow. The presence of an integer overflow bug is indicated if for the selected SMT system the Z3 solver reports `SAT` (satisfiable, integer overflow bug present).

INTREPAIR re-constraints (for example, through integer upper bound negation) the variable(s) selected from the previous step in order to determine a potentially safe interval which will not lead to a second integer overflow of the symbolic variable. Note that other iterative techniques are possible (*i.e.,* iterating backwards through a vector (from large to small values) of consecutive large values and checking by selecting each value once as integer upper bound if the check conditions will hold). The goal is to determine if there is a second integer overflow if INTREPAIR re-constraints the selected variables as mentioned above. For this purpose, INTREPAIR will check in the next step if for the new SMT constraint system it gets an `UNSAT` (unsatisfiable, no integer overflow bug present) solver reply. The new constrained SMT will be composed of the old SMT constraint system, which was used to detect the integer overflow, complemented with the re-constrained SMT equations. If INTREPAIR gets an `UNSAT` solver reply, then it determines that there will be no integer overflow if it re-constraints the selected variable(s) with the new constraints (*e.g.,* variable range negation, *etc.*) and as such the integer overflow can be avoided. Finally, the information collected at this step will be used in later steps.

5.3.5.5 Determine the Bug Type

For a unique detected overflow, INTREPAIR attributes a report containing a unique bug identifier. Based on the generated bug identifier (ID), INTREPAIR can determine which bug type it currently deals with. This information is extracted from data stored during step two. With this ID, INTREPAIR checks in the list of currently supported checkers to which checker this stored information belongs to and determines which repair patterns can be used to repair the previously detected bug. This way, repair patterns usable for other types of bugs will not be suggested when repairing integer overflows.

5.3.5.6 Selecting Repair Pattern

Based on the previously determined bug identifier (ID), INTREPAIR selects the suitable repairs for this integer overflow from the pool of repair patterns using the decision tree (see Figure 5.2 for more details).

Pattern. The repair patterns of INTREPAIR need to address the following challenges. Each pattern needs to incorporate complex conditions with multiple branches depending on the type of potentially overflowing code location. An INTREPAIR pattern needs to be pre-classifiable for typical code locations (*i.e.,* statements) where it would best apply depending on the AST of the buggy statement(s). Further, each pattern needs to have at least two branches: (1) in the case the check succeeds, and (2) in case the check does not succeed for error logging. The pattern needs to have several customizable components which can be altered during static analysis with: context dependent values, mathematical functions, or `C` functions extracted from the buggy statement. As such, INTREPAIR's repair patterns need to be nontrivial fragments of incomplete code, which have to be versatile and applicable to different integer overflow repair scenarios.

Patching. The repair patterns impose the following challenges on INTREPAIR's patching process. After the programmer/developer selects a repair pattern, the automated repair mechanism of INTREPAIR needs to be able (1) to extract the components of the buggy statement such as variables, functions (*i.e.,* `malloc`, *etc.*) and reuse them for augmenting the selected pattern; to achieve this, the patching mechanism needs to be able to match these extracted components with the parts of the pattern where these fit, (2) next, the patching mechanism needs to be able to precisely incorporate the buggy statement(s) inside the newly selected pattern, (3) the patching mechanism needs to be able to precisely delete the previous statement(s) where the bug was detected and to rewrite the existing source file with the newly generated repair pattern at the correct source code location. As a consequence, the challenges imposed by the repair patterns and IntRepair's patching process are nontrivial and require high precision during source code file rewriting.

Repair Pattern Description. IntRepair's repair patterns consist of `C` code skeletons where different repair parts will be replaced with: (1) concrete values after their values have been computed, (2) mathematical operations (*e.g.,* division by a value, *etc.*), and (3) standard `C` library functions. Depending on the context, placeholder variables will be replaced with corresponding mathematical functions such as the square root function `sqrt` or other functions. In the situation in which a mathematical function is used IntRepair does not need to compute the value of the function upfront (during static analysis) but rather leave this to be computed later during symbolic execution analysis (repair validation) or program runtime. This offers the advantage that IntRepair does not need to be able to compute any possible mathematical function.

Further, the code repair patterns used by IntRepair are based on preconditions similar to IOC's [73] checks (see Section 5.3.2 for more details). These preconditions can cover different types of mathematical operations such as multiplication of numbers and addition of variables. At the same time, the repair patterns are highly configurable and versatile. For example, the programmer can easily change, if needed, the error handling function inside the repair pattern or can extend the precondition such that it captures more complex bug avoiding preconditions. This can be achieved by modifying a few lines of code inside an existing pattern or by creating a new pattern and defining the conditions (*i.e.,* depending on the structure of the AST of the statement where the bug was detected) when such a pattern fits best.

Selecting Repair Patterns. For the sake of brevity, we will describe the steps used for selecting a code repair based on a `C` statement having three components, `leftHandSide`, `rightHandSide`, and `operator`. However, IntRepair can deal with more complex statements as well, see Figure 5.2 for more details.

IntRepair follows the following steps to select a suitable repair pattern. First, the code statement where the integer overflow error was detected is divided into its components based on its AST. For example the AST components of a `C` statement such as `int result = varA + varB;` are `leftHandSide=varA`, `operator=+` and `rightHandSide=varB`. Second, a series of rules are checked against the AST of the previous `C` statement as follows: (1) is `leftHandSide` equal/different than `rightHandSide`, (2) what type of operator do we have in the statement, and (3) how many components does the statement have after the = sign, and so on. Third, based on these rules the repair pattern which satisfies the highest number of constraints will be selected. Note that each repair pattern has a list of properties (*i.e.,* use when `rightHandSide = leftHandSide` and `operator = +`, *etc.*) attached to it that are checked against the above stated rules. This list of properties is statically defined at the moment when the repair pattern was manually added to the pool of available repairs contained inside IntRepair. Further, in case there are more repair patterns that fulfill the same number of rules, IntRepair selects the first repair pattern occurring in the list. In case IntRepair cannot determine the pattern selection criteria due to, for example, a complex C language statement, then no repair is proposed.

Finally, note that if needed this approach can be updated such that all legitimate repair patterns will be proposed and for each a repair can be generated and selected with a human-in-the-loop approach.

Repair Pattern Example. In the following, we present a repair pattern used by IntRepair.

```
1if(leftHandSide.equals(
2rightHandSide)&&
3operator.equals("*")) {
4...
5return "if(sqrt("+leftHandSide+") <= sqrt("+value4+") &&
6        -sqrt("+ rightHandSide+") >= -sqrt("+value5 +"))
7         "+"{"+"\n"+"\t"+"\t"+"\t"
8           +buggyStm10 + "\n }else{ \n"
9           +"FILE *fp=fopen (\"IO_"
10          +"error_log.txt\", + \"w+\");"
11          +"\n fprintf(fp, \"IO_ID:%s
12          +FileName:%s LineNumber:%d",
13          +FileName+","+IO_bug+","
14          +LineNumber+");"
15          +"\n fclose(fp);" + " }\n";
16}
```

Figure 5.3: Repair pattern example.

Figure 5.3 depicts a code repair pattern used by IntRepair during integer overflow error repairing. This code repair pattern contains C code compatible snippets (shaded in red color) interleaved with several stub variables which will be replaced with concrete variables names, values or mathematical functions depending on the type (depending on its AST structure) of code statement containing the bug. For example, the repair pattern depicted in Figure 5.3 will be used by IntRepair when the `leftHandSide` equals (*i.e.,* string wise comparison) the `rightHandSide` and the `operator` equals (*i.e.,* string wise comparison) the product operator $*$. After these checks have been performed, the repair will be assembled as follows.

First, `value4`, `value5` and `buggyStm10` located at lines 5, 8, and 10 in Figure 5.3, are replaced with: (1) the squared root of the currently selected integer upper bound value `value4` $\leftarrow$ `sqrt(2,147,483,647)`, (2) the negated integer upper bound value `value5` $\leftarrow$ `-sqrt(2,147,483,647)`, and (3) the program code statement that contains the previously detected integer overflow error `buggyStm10` $\leftarrow$ `int result = data * data;`.

Second, the variables `FileName`, `IO_bug` and `LineNumber` located at lines 15-16 in Figure 5.3 are replaced with concrete values obtained during bug detection. Finally, note that: (1) other code repair patterns can be selected and used based on the format of the AST of the program buggy statement, (2) our technique can be easily generalized to more complex C code statements than the ones mentioned herein, and (3) each generated repair can easily be customized to fit to different types of integer overflow mitigation scenarios, *i.e.,* error logging, *etc.*

5.3.5.7 Determine New Constraint System

In this step, IntRepair assembles the new SMT constraint system which is used to determine if the previously detected integer overflow is still present. During this step, IntRepair takes the constraints determined at the previous step and inserts them in the SMT constraint system which was used to detect the integer overflow. Before inserting the new constraints in the SMT system, IntRepair needs to remove the original SMT constraints which were used to detect the presence of the integer overflow. The decision which SMT constraints have to be removed from the SMT system (used to detect the integer overflow presence) is made based on the modular aggregation of these SMT statements inside the integer overflow checker in which these constraints are put together. More precisely, since the SMT constraints used to check for an integer overflow are added in the integer overflow checker in an incremental manner, IntRepair can precisely determine which constraints can be safely removed and replaced with new ones determined at the previous step. Next, this SMT system will be fed into the Z3 solver. In case the solver replies `SAT`, then the constraints added represent valid constraints which can be used to avoid the previously detected integer overflow. This new SMT constraint system serves as ground truth w.r.t. the fact that the integer overflow can be removed if certain symbolic variables are re-constrained in an appropriate way. Finally, note that the above-described symbolic variables have concrete counterparts values, which will be inserted inside the code repair when assembling it.

5.3.5.8 Generate Code Repair

This step consists of putting together the final code repair(s) and saving them into a list in case multiple repairs are suggested. After the repair components have been inserted into the previously selected code repair pattern, IntRepair generates a `C` code repair which is syntactically correct, can be compiled and can be further on edited after insertion (if desired). Note, that IntRepair generates correct patches w.r.t. the correctness definition provided in Section 5.1, which essentially means that IntRepair re-checks the program after a patch was applied in order to determine if the bug was really removed, and the program behavior was not changed.

5.3.6 Implementation

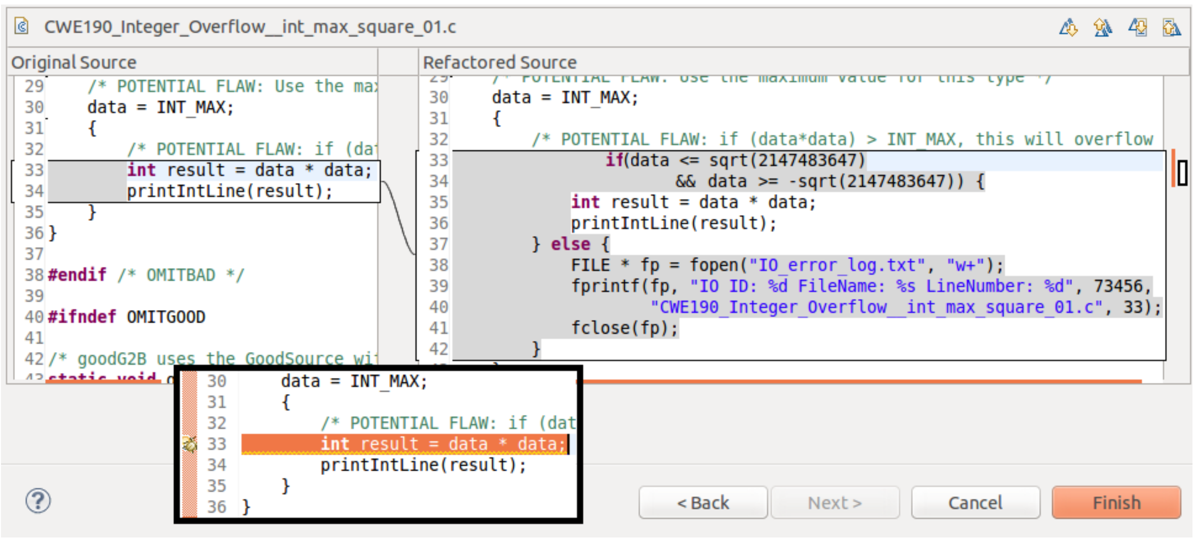
We have implemented the approach presented in Section 5.3 inside a system called IntRepair. IntRepair is an Eclipse [77] plug-in based on the Codan [138] API. IntRepair consists of approx. 10 KLOC and is developed as an Eclipse plug-in mainly because: (1) the Eclipse CDT API can be easily reused, (2) a GUI is easily obtainable, and (3) the obtained tool can be used in both online (during code typing) and off-line (after finishing code typing) modes. Note that Codan [138] is used by IntRepair to construct the program CFG, analyze the program AST statements, and perform bottom-up traversals by using a `C` program statement visitor in order to construct SMT constraints.

The assembled repair generated in subsubsection 5.3.5.8 is sent to the Eclipse language tool kit (LTK) API [155] based component of our engine, which will assemble the repaired code.

The LTK component adds information on how to position the repair in the buggy program such that the integer overflow will not occur after the repair was inserted. In case multiple repairs have been generated, IntRepair saves these repairs in a list of repair candidates for the previously detected integer overflow. Finally, the repair which was selected by the programmer will be passed to the IntRepair repair insertion component, which will create two differential views (*i.e.,* with the repair inserted in the file containing the bug and without).

Three auxiliary tools are implemented as well to work together with IntRepair. Next, we briefly describe the three additional tools which we implemented and which offer support to IntRepair during program patching: (1) an Eclipse CDT projects generator tool which is used for generating Eclipse CDT compatible projects, (2) a source code refactoring tool based on the Eclipse LTK [155], JFace [127], and Eclipse CDT [77], for inserting program repairs, and (3) a test case generator tool which can help to assess the accuracy of integer overflow detection using test cases.

5.3.7 Graphical User Interface in an IDE



```
    /* POTENTIAL FLAW: Use the ma
    data = INT_MAX;
    {
        /* POTENTIAL FLAW: if (da
        int result = data * data;
        printIntLine(result);
    }
}

#endif /* OMITBAD */

#ifndef OMITGOOD

/* goodG2B uses the GoodSource wi
```

```
    data = INT_MAX;
    {
        /* POTENTIAL FLAW: if (data*data) > INT_MAX, this will overflow
            if(data <= sqrt(2147483647)
                && data >= -sqrt(2147483647)) {
            int result = data * data;
            printIntLine(result);
        } else {
            FILE * fp = fopen("IO_error_log.txt", "w+");
            fprintf(fp, "IO ID: %d FileName: %s LineNumber: %d", 73456,
                    "CWE190_Integer_Overflow__int_max_square_01.c", 33);
            fclose(fp);
        }
```

```
    data = INT_MAX;
    {
        /* POTENTIAL FLAW: if (dat
        int result = data * data;
        printIntLine(result);
    }
}
```

Figure 5.4: Screenshot of the GUI of IntRepair.

Figure 5.4 shows the IntRepair Graphics User Interface (GUI). First, the integer overflow triggers a bug marker depicted in the black bordered box with a yellow bug icon. This means that within the C statement an integer overflow error was detected. Second, by right clicking on this bug marker the user can start the code re-factoring wizard. The code re-factoring wizard is composed of two windows. The first window is used to make repair type decisions (currently only in-place repairs are available). The second window depicted in the background of Figure 5.4 contains a differential files view visualizing the differences between the original file containing the bug and the modified file with the selected repair inserted. This second window helps the developer analyze the patch before it is applied. Finally, it is possible to navigate between these two windows and in case the programmer decides to insert the repair generated in Section 5.3.5 then this can be achieved by left-clicking on the `Finish` button.

5.4 Evaluation

In this Section, we present the results of our experimental evaluation and address the following research questions (RQs):

- **RQ1:** How effective is INTREPAIR in repairing integer overflows? To what extent does INTREPAIR produce false positives? (Section 5.4.2)
- **RQ2:** To which degree does INTREPAIR ensure that a repair completely removes the detected integer overflow? (Section 5.4.3)
- **RQ3:** What is the performance impact of INTREPAIR? (Section 5.4.4)
- **RQ4:** How do the bug repairs of INTREPAIR preserve program correctness? (Section 5.4.5)
- **RQ5:** How effective is INTREPAIR compared to manual repair of integer overflows? (Section 5.4.6)

5.4.1 Evaluation Setup

Subject Prog.	LOC	# Programs	Description
CWE-190	up to 638	2,052	Open source testsuite
Benchmark	up to 20 KLOC	50	We introduce a benchmark of synthetic programs

Table 5.2: Overview of the subject programs.

Table 5.2 depicts a summary of the programs used to evaluate INTREPAIR. Next, we give details about the used programs.

Juliet benchmark. First, we selected 2,052 `C` programs contained in the SAMATE's Juliet test suite [264], which is the largest open source test suite for `C/C++` code to the best of our knowledge. In particular, we consider the CWE-190 category. Each program contains on average 476 LOC with a maximum of 638 LOC. Each program has integer overflows: exactly one true positive and several false positives (see characteristics of the Juliet test suite [264] for more details). We used these program as they contain all possible program control flows which may lead to an integer overflow and further, these help to assess previously known faults in a systematic way.

Synthesized Code. In addition, we build a benchmark with 50 synthesized programs with complex control flows and with a high number of code lines. In each synthesized program, the exact number of seeded integer overflows is known. The benchmark allows us to demonstrate that IntRepair can scale efficiently to large and complex programs. The synthesis of programs is parameterized as follows: (1) the total number of function calls, (2) the number of loop iterations, and (3) each generated program contains exactly one true positive and a variable number of false positives, and the true positive should be located deep inside the program, *i.e.,* several thousands of branches nested inside the program execution tree. Using these limits, we generate 50 programs ranging from 6 to 20 KLOC. The synthesis algorithm iterates through several loops in which it adds functions in the program which are calling each other. These functions contain a variable number of branches which are also counted until a certain depth is reached.

Statistics. Table 5.3 depicts several characteristics of the 2,052 analyzed programs contained

Subject Prog.	Satisfiable Paths	Unsatisfiable Paths	Program Branch Nodes	Program Branches
CWE-190	115 K.	1.2 Mil.	5.73 Mil.	12 Mil.
Benchmark	45 K.	0.7 Mil.	4.4 Mil.	8.8 Mil.

Table 5.3: Descriptive statistics of our subject programs.

in SAMATE's Juliet test suite and the 50 programs of our benchmark. The programs contained in the CWE-190 contain on average: 56 satisfiable paths, 584 unsatisfiable program execution paths, and 5,800 program branches, which were counted during program analysis. In contrast, the programs contained in our own benchmark have on average 900 satisfiable paths, 14,000 unsatisfiable program execution paths, and 28,000 program branches, which were counted during the analysis of the programs.

User Study. We also performed a controlled experiment with 30 participants in which we assessed the efficiency of IntRepair during integer overflow repair. The used protocol is described in Section 5.4.6.

Experiment Setup. Experiments were conducted on a Dell desktop with an Intel CPU Q9550 `@` 2.83GHz, 64-bit, 12GB RAM, Eclipse IDE Kepler and OpenSuse 13.1 OS.

5.4.2 Effectiveness

As our main test-bed we selected 2,052 `C` programs contained in the Juliet test suites [264] and 50 programs from our customized benchmark.

Methodology. We prepare the 2,052 `C` programs in Juliet in order to be handled by IntRepair. Further, we generate 2,052 test cases which are used to dynamically assess if the detected integer overflow errors are detected at the correct source code location. For the automated generation of these test cases we used one of the tools presented in Section 5.3.6. We do the same pre-process with the 50 programs from our own benchmark.

(1) We assess the effectiveness of integer overflow detection by running the generated test cases for all programs (including the 2,052 `C` programs in the Juliet test suite and 50 `C` programs contained in our benchmark) and checking the generated reports. More precisely, we checked whether each report describes an integer overflow reported at the expected location. (2) For those reports, which are confirmed as true positives, IntRepair inserts a suitable repair into the overflow site. (3) Then, IntRepair was run again in order to check if this was a true positive and if the fault was removed.

We point out that, for the assessed programs, IntRepair did not detect any false positives. Further, IntRepair is able to detect and repair all previously detected true positives present in the analyzed programs without repairing any false positives. Finally, note that the ground truth is that each of the 2,052 `C` programs contains one true positive and multiple false negatives and the same is true for the programs contained in our benchmark.

Results. According to our experiments, IntRepair is able to detect all integer overflows (2,052 overflows contained in the Juliet test suite and 50 in our benchmark) at the expected locations in the analyzed programs. After the repairs conducted by IntRepair, the integer overflow reports are no longer generated. This means that IntRepair is able to perfectly repair all integer overflows for the analyzed programs.

5.4.3 Bug Removal

In this Section, we explain how the integer overflows are completely removed by IntRepair. Recall that IntRepair checks if a bug was completely removed after a repair was inserted in two operation modes.

Methodology. In this experiment, we use IntRepair's *Manual Mode*. In this mode, the user can again analyze the repaired program by manually restarting the static analysis. In case the symbolic analysis detects no other (or the previous fixed bug) integer overflow, then no bug report will be generated. This helps the programmer to conclude that the bug was completely removed and no new bug was inserted into the program.

Actually, we use IntRepair to re-analyze each patched program. For each of the repaired programs, we checked if after the program was repaired, a new bug report was filed.

Results. After patching and subsequently reanalyzing each program, no new overflow reports are generated, meaning that the bugs are completely removed without inserting new ones. For all programs, the errors were successfully removed by inserting the repair at the correct location.

5.4.4 Performance

In this Section, we evaluate the performance of IntRepair in three ways (1) the time IntRepair takes; (2) binary size blow-up, and (3) the runtime overhead caused by the overflow repair. Finally, we show that IntRepair has the potential to scale to large programs as well.

5.4.4.1 Static Analysis Time

The time that IntRepair spends in static analysis, *i.e.,* static symbolic execution and overflow repair processes, is an important criterion to assess the usability of our tool.

6 KLOC Programs	46 sec. (< 1 Min.)
11 KLOC Programs	151 sec. (< 3 Min.)
20 KLOC Programs	567 sec. (< 10 Min.)

Table 5.4: Average repair generation time in seconds.

Results. Table 5.4 depicts the average execution time over 10 runs for each of the benchmark programs grouped in three main categories based on their LOC. IntRepair handles all programs in under 10 Min. (567 sec.) on average. Hence, we believe that IntRepair is time-effective and usable in practice.

5.4.4.2 Binary Size Blow-up

We assessed the source code and binary blow-up by counting the increase in source code lines and in bytes for the programs before and after applying the repairs.

First, we compared the total number of lines contained in the source code against the number of lines of code which were added after inserting all the repairs into the 2,052 vulnerable programs. As already mentioned, the initial number of lines was 977.7 KLOC. After applying all repairs, we added in total around 10,045 LOC.

Results. IntRepair yields a binary blow-up of less than 0.56% in LOC. In our opinion, this represents an acceptable source code lines increase.

5.4.4.3 Execution Overhead

We evaluated the runtime overhead introduced by IntRepair by comparing the execution time of unrepaired programs against the one of the repaired programs.

Results. In our experiments, we recorded an average runtime overhead of 1.57%. Thus, the inserted repairs do not considerably influence the runtime overhead of the repaired program.

5.4.5 Correctness

We validate the correctness of our repairs by checking that our symbolic execution component generates semantically different code. In other words, we want to find out if the repairs can potentially influence the program behavior and the repair correctly removes the bug (see our patch correctness definition in Section 5.1 for more details).

To do so, we compare the generated SMT model before and after the repair. For this, IntRepair's symbolic execution component stores the SMT system which was used to trigger the bug report. Next, this constraint system is compared with the new SMT system which was generated after repair validation. This results in the following analysis process:

1. For all programs contained in our benchmark and the Juliet test suite, we stored the SMT system for each program and the node (*i.e.,* source code line number and file name) where the bug is located.
2. We applied the repair to the detected integer overflow by automatically rewriting the program.
3. We re-run the symbolic analysis on the patched program and store the new SMT system.
4. We semantically checked each SMT system stored at step (2) and each SMT system collected at step (4) based on string comparison of the node with another node, where a node is a complete statement, *e.g.,* `x = p + 5`. This enables IntRepair to detect the differences between these two systems.
5. We compare the SMT system differences and proceeded as follows: (1) if the new SMT system only reflects the semantics of the inserted repair: we report no program correctness violation, (2) otherwise, we reported that the program correctness will potentially be affected by inserting this repair.

We automatically compared in total 225K SMT models and all patched programs. These differ by up to five AST nodes and by less than three SMT constraints on average, respectively. Further, all checked SMT models differ only with respect to the repair semantics and no additional program semantics were introduced in the program due to the repairs.

Additionally, we performed the following evaluation process in order to increase the level of confidence w.r.t. repair correctness.

Re-check Patched Program. We re-run the symbolic analysis, after applying the repair, on each of the programs in order to see if there is a potential new integer overflow error or the old one is still present in the repaired program.

Check Syntactical Program Correctness. We recompiled (we used the GCC compiler) each of the programs to investigate if the repaired program is compilable and thus syntactically correct. The compiler reported no syntactical errors.

Results. The normal behavior of the repaired programs does not change after applying the repairs, thus program correctness is preserved.

5.4.6 User Study

In order to evaluate IntRepair's efficiency w.r.t. integer overflow repairing compared to manual repairing, we performed a user study with 30 graduate students (16 males and 14 females). They all had on average between one up to two years of industry programming experience.

Methodology. The experiment was conducted with each participant individually at a single computer with an additional person in the room who overlooked the whole experiment. The computer used in our experiment was equipped with two versions of Eclipse CDT, *i.e.,* one Eclipse version with INTREPAIR installed and the other Eclipse version without INTREPAIR installed. We split the 30 participants evenly into two groups[3]. We randomly selected three programs from our benchmark, and we told each participant of both groups to search for a single integer overflow in each of the three programs and to repair it. The participants were allowed to read and execute the program. A debugger was not available and no test case for executing the overflow was provided. Further, all participants had similar experience with the task at hand, particularly fault detection, as they had 1-2 years of experience as developers in a software company. Finally, in the case of manual debugging no test cases where prepared for simplicity reasons. The experiment supervisor assessed each bug detected manually by manual inspection after this was reported by the experiment participant.

Group 1. Each participant had access to the latest Eclipse CDT IDE and to the GCC (v. 4.9.3) compiler [82] through terminal access. The Eclipse CDT IDE in this group were installed without INTREPAIR.

Group 2. Before the experiment, each participant from this group got a short one minute demo movie showing how to use INTREPAIR. Each participant could search for bugs and fix them with the help of INTREPAIR.

We measured the time needed for each participant to locate the bug and repair it and the success rate for each analyzed program after the participant decided that he/she was finished with all three programs at the end of their analysis. After the experiment, we asked each participant if (1) *he/she would reuse* INTREPAIR *in his/her daily routine* and if (2) *he/she would recommend it to other peers*. Finally, note that each question should be answered with *yes* or *no*.

Results. (1) In total, the participants of *Group 1* needed 6534 seconds (108 Min.) to repair the bugs in the experiment, whereas participants from *Group 2* only needed 362 seconds (6 Min.) to repair them. That means, programmers using INTREPAIR in our experiment were over 18 times (6534/362) faster compared to relying on manual repair.

(2) We inspected the repairs inserted manually by *Group 1* and those inserted by INTREPAIR. 41% of the manual repairs were correct, *i.e.,* the integer overflow bugs were removed and no further vulnerabilities were introduced. By construction, all the repairs with the help of INTREPAIR were correct (see *RQ2* for more details). Hence we conclude that INTREPAIR is more effective w.r.t. overflow repair than manual analysis.

(3) Out of the total participants of *Group 2*, 94% (14 participants) found INTREPAIR to be very useful and 87% (13 participants) would further recommend INTREPAIR to others.

[3]The number of females and males was split evenly between the two groups.

5.5 Discussion

In this Section, we discuss the threats to validity of our experiments and the limitations of IntRepair.

5.5.0.1 Threats to Validity

Internal validity. depends on the correctness of our prototype implementation and may be affected by the evaluation setting and the execution of the conducted experiments. To mitigate this, we carefully tested our prototype by repeating the experiments ten times and by taking average performance values. To validate the correctness of our automatically generated patches, we re-ran the analysis on the program after patching and inspected each program patch manually in order to check for correctness.

External validity. threats may derive from the selection of programs which we used in our evaluation. We validated our repairs on 2,052 programs and also on seeded programs having up to 20 KLOC contained in our, especially for this purpose designed, benchmark. In more complex applications, program repairs can dramatically increase the runtime overhead when, for example, these reside in *hot* code such as loops. Thus, different results could be obtained for different analyzed programs.

Construct validity. threats may appear from the fact that our patches may introduce overhead or unwanted program behavior. We point out that our repairs are suggested only for code locations where previously an integer overflow was confirmed. Further, our experiments show that our patches do not harm performance in any significant way, do not change program behavior, are automatically validated, and are applicable only at previously confirmed bug locations.

5.5.0.2 Limitations

First, at this stage of development the used static symbolic analysis framework (Codan) supports a subset of the `C/C++` programming language semantics. Thus currently only certain types of statement and function headers can be handled by IntRepair. Nevertheless, we do not consider this to be a major limitation, adding all `C/C++` statement types and function headers can eliminate this. Investing sufficient time and manpower can address this limitation.

Second, IntRepair's implementation depends on program loop and recursion depth unrolling which incurs well-known precision penalties for all symbolic execution-based techniques. These operations can introduce false positives due to loop unrolling and recursion depth. Note that, by design, IntRepair does not produce false positives (*i.e.,* each detected fault is a genuine one). Our static analysis is time-consuming. Its accuracy and performance will affect IntRepair's results. IntRepair does not have access to runtime information and a real environment is also not available. In order to deal with this limitation, IntRepair uses function stubs which can help to simulate the environment (interaction with third-party libraries). Note that loop unrolling and recursion depth have to be bounded when using symbolic execution-based approaches in general since we want to avoid the path explosion problem and thus infinite analysis time. We want to address this potential insufficiency by a prior analysis of program loops in order to derive loop invariants. Further, we want to extend the list of currently supported stub functions for the main `C` most used system libraries.

Third, complex program expressions cannot be handled due to limited support in the currently available open source state-of-the-art of SMT solvers for non-linear computations. Thus, we want to address this potential limitation by carefully providing SMT constraints which capture as good as possible the original program constraints and by experimenting with different SMT solvers.

Fourth, as IntRepair relies on static symbolic analysis of source code, it has several limitations which are common to all static analysis techniques when comparing to dynamic analysis techniques such as the well-known path explosion problem. Note that static searching through the program paths is different than dynamic path analysis where program path coverage is driven by the provided program input. We addressed this by implementing in IntRepair different path exploration techniques in order to analyze more error-prone paths first.

Fifth, IntRepair's evaluation is based on the currently largest open source test suite for `C/C++` programs, *i.e.,* SAMATE's Juliet test suite, due to the fact that it contains complex control flows and a large number of situations where integer overflow can occur. The programs contained in our benchmark are comparable with real-world programs. As a result, the findings of our evaluation do not necessarily reflect the behavior of IntRepair when applied to larger programs to some extent. However, we do not think that this limits the applicability of IntRepair, since our tool is highly scalable due to its configurable analysis. Further, we want to evaluate IntRepair on even larger programs as well.

Finally, we tested IntRepair in a controlled experiment with a restricted number of participants. For this reason our findings may differ from industrial settings where real development conditions are available. But we think that our tool can help to drastically cut down the time needed for bug detection and repair due to its usability and low intrusiveness. Thus, we also want to evaluate IntRepair in industry settings as well.

5.6 Summary

In this Chapter, we presented IntRepair, a novel framework which provides comprehensive integer overflow detection, correct repair generation, and validation. To the best of our knowledge, we provide the first static symbolic execution-based technique that combines detection, generation and validation of source code repairs for `C` programs. We applied IntRepair to 2,052 `C` programs (approx. 1 million LOC) and to 50 programs contained in our seeded benchmark, which includes programs that have up to 20 KLOC. Our experimental results show that IntRepair was able to effectively detect integer overflows and successfully repair them with source code patches. Our controlled experiment shows that IntRepair is 18 times more time wise effective and has a higher repair success rate than manual repairs.

Chapter 6

BuffRepair: Static Repairing of Buffer Overflow Based Memory Corruptions

In this Chapter, which belongs to the first part of this thesis, we extend our static source code analysis framework in order to be able to detect and repair buffer overflows which are one in many situations a main prerequisite for CRAs. In this Chapter, we provided a tool, called BUFFREPAIR, which can automatically generate buffer overflow repairs, does not suffer from false negatives, and can validate a repair. At the same time, we fully addressed **RQ3** and integrated our tool into a well used and established IDE as we did with both our tools, INTDETECT and INTREPAIR. Finally, note that parts of this Chapter have already been published by Muntean *et al.* [196].

6.1 Introduction

What is the overall view? "On one hand, if one tries to put or to retrieve data from a non existing place/index he is going to make a mess. On the other hand, if one tries to put more or retrieve more data from a smaller place/index then he is going to cause a mess, too." *Mother Nature Law*. According to the 2011 CWE/SANS top 25 of most dangerous software errors [171] which can lead to serious vulnerabilities in software, buffer overflows are ranked on 3rd place after SQL injection and OS command injection.

What is the problem? Buffer overflows can generate risky resource management vulnerabilities as the recent Heartbleed bug [173] confirms or can lead to a memory corruption which is usually needed during a CRA. This bug generates a buffer over-read in the OpenSSL library by leaking sensitive information to the outside world without the need for the attacker to have root access on the attacked system and without leaving any trace on the attacked system. This proves that buffer overflows can lie undiscovered in software for many years and can lead to extremely dangerous information leaks in highly used open source software.

What are the existing solutions? There are many solutions which can generate buffer overflow repairs [109], such as, from: (1) free form bug reports [148, 277, 3], (2) statically defined patch patterns [132, 105], (3) test suites using SMT solvers [69, 205], (4) test suite and genetic programming [141, 278], by replacing the unsafe *libc, alexey:patch, Sauciuc:reverse*, functions with safe functions, [58]. Further, in recent years, many *quick fix generation tools for buffer overflows* have been proposed, such as: AutoPaG [146], SafeStack [40], DYBOC [246], TIED [12], LibsafePlus [12], LibsafeXP [145], and HeapShield [79].

What are the limitations of the solution? None of the existing tools can reason as precisely about the generated buffer overflow repairs with respect to: repair validation, program behavior preserving, and precise fault localization before the repair is inserted, as BUFFREPAIR does. More specifically, to the best of our knowledge, AutoPAG [146] is most similar to our approach from the backward visiting of program statements located on a program execution path perspective. BUFFREPAIR can not be compared with AutoPAG from the point of view of computation time and quick fix quality at this stage of development since AutoPAG has several limitations which we will briefly list. Our algorithm stops the search after encountering the first *non-in-place* bug fix location whereas AutoPAG tries to detect all possible *non-in-place* bug fix locations by running a repeated inefficient data flow analysis (no program execution paths are used). AutoPAG is not aware of program execution paths and uses a rudimentary backward information flow propagation approach based on the sequential ordering of program statements. The analysis (no SMT solver used) is repeated until there are no visited variables in the previously constructed set of tainted variables. This set can contain all program variables and can generate a significant overhead as already mentioned in the AutoPAG paper.

What is our insight? In this Chapter, we focus on fault localization and repairing of buffer overflow bugs by leveraging precise information (*i.e.,* failure detection, bug diagnosis, buggy variables (program variables which are directly responsible for bug appearance), *e.g.,* buffer index or buffer size) provided by our buffer overflow checker [111]. The failure detection and bug diagnosis data is used to generate quick fixes for buffer overflows and to support the repair process of removing the bug with a refactoring wizard. A novel algorithm is used to detect possible insertion locations in code for the generated code patches ((a) *in-place*—directly before the statement which contains the bug and (b) by searching for other, *non-in-place* locations where the bug can be fixed). Our approach for generating program repairs is based on: code patch patterns, SMT solving and possible quick fix locations searching in program execution paths which could affect the program behavior by inserting a patch at a *non-in-place* location. The generated patches are sound (*e.g.,* do not change the behavior of the program for input which does not trigger the bug), final (no further human refinement needed), human readable (no alien code), syntactically correct and compilable.

We address offline behavioral repair [190] (by modifying the source code). Others have addressed state [70] or test-suite based program repair such as GenProg [141] and PAR [132]. The defect class which we address is inappropriate index variable assignment which results in an incorrect usage of the buffer index range. The fix defect class consists of input checks based on semi-defined patch patterns. The aim of the quick-fix is fail-secure error mitigation (*e.g.,* to prevent that an attacker exploits the error in order to gain system access). The final version of the patch is determined using SMT solving.

Program repair lies at the conjunction of two dimensions (first, an *oracle* is needed to decide what is incorrect in order to detect the bug (first dimension) and another *oracle* to tell what should be kept correct for sake of non-regression (second dimension)) of software correctness [190]. We used the same SMT constraint system which was used to trigger the bug for defining what is incorrect in the program. Additionally, we created a second SMT-Lib (constraint system definition language used by the Z3 [68] solver) constraint system consisting of the previously mentioned constraint system and new SMT-Lib constraints used to impose input saturation constraints on the buggy variable.

Our patches are generated automatically and inserted semi-automatically off-line with the possibility to insert them also online.

Our problem statement: Provide code patches (*in-place* or *non-in-place*) which can be used independently to remove a buffer overflow bug using a bug detector (checker).

What are our contributions? In summary, in this Chapter, we make the following contributions:

- An algorithm for generation of *in-place* and *non-in-place* buffer overflow quick fixes.
- A novel tool callled BUFFREPAIR usable for fault repair generation based on program input saturation.
- Semi-automated patch insertion based on source code files differential views.
- Automated check for program behavior preserving after a patch was applied.

6.2 Motivation

In this Section, we present two real-world bug repairs as an example to highlight the fact that bug patch generation is not a trivial task. It requires deep insights into the functionality of the program and merits further study. There are typically an endless number of programs who adhere to a formal specification. As such, a bug can be fixed with an infinite number of functionally correct patches. Thus, the automatically generated repairs may change the behavior of the program.. We present two distinctive patches depicted in Figure 6.1 on lines 5–6 and 11–13 with the + sign and by using an italic font. Note that these two repairs do not change the program behavior for program input which does not trigger the bug. Figure 6.1 contains on line six code comments which we present in order to indicate other possible repairs usable to remove the buffer overflow bug located at line 12 which most likely will change program behavior.

Figure 6.1 depicts a `C` code snippet extracted from the test case CWE-121 [172] which is contained in [206]. The code snippet contains a buffer overflow bug at line 12 which can be removed using one of the two patches depicted in Figure 6.1 on lines 5–6 and 11–13. Note that the patch structure, the used constraint variables and the bug repair insertion locations are different for each buggy `C` program.

```
void foo_bad(){
int data = -1;
char input_buf[CHAR_ARRAY_SIZE] = "";
if (fgets(input_buf,CHAR_ARRAY_SIZE,stdin) != NULL){
  data = atoi(input_buf);
+ if (data > 9 || data < 0)
+ exit(EXIT_FAILURE); // data = 9; or data = rand() % 9; or return 0;
}else{
  printLine("fgets() failed.");}
int i, buffer[10] = { 0 };
if (data >= 0){
+ if (data <= 9 && data >= 0){
     buffer[data] = 1;  // Buffer overflow bug, index out of range
+}else{exit(EXIT_FAILURE);} // stop program execution
  for(i = 0; i < 10; i++){printIntLine(buffer[i]);}
}else{
  printLine("ERROR: Array index is negative.");}
}
```

Figure 6.1: Buffer overflow bug due to missing input checks.

Finding the *right* program variables in order to impose a constraint through a patch is a hard task because in the worst case the values selection depends on all the other program variables. Determining *non-in-place* bug repair locations is not a trivial task and should be based on correct bug detection and on a kind of backward program execution technique on all program execution paths which contain the bug. In general, the insertion location and structural form of the quick repair patch can influence the overall program behavior. Thus, care should be taken that a patch is syntactically correct, compilable and does not change program behavior for program input which does not trigger the bug.

6.3 Threat Model

Defensive Assumptions. We align our threat model (*e.g.,* STRIDE [165]) to the general buffer overflow threat model. We assume that ALSR [219] and DEP [164] are in-place and correctly functioning. We assume that the code is not compiled with any buffer overflow bounds checking. Lastly, we assume that the defender has access to the source code of the application and potentially knows how to repair the buffer overflow.

Attacker Capabilities. We assume a skilled attacker with sufficient resources and time to exploit the buffer overflow. Further the attacker can use the buffer overflow to escalate privileges or to manipulate the program counter according to his desires. The attacker is equipped with the necessary tooling to find the buffer overflow in the program binary and to experiment with the code in an live interactive debugging session. Next, we don not exclude the possibility that the attacker has access to the source code of the application. Lastly, the attacker is able to probe the found buffer overflow in order to see how large the payload can be which he tries to insert in the vulnerable program.

6.4 Design and Implemnetation

In this Section, we present our quick fix locations search algorithm, the steps needed to automatically generate buffer overflow fixes and the mechanism for inserting the patches semi-automatically into the buggy program.

6.4.1 Quick Fix Locations Search Algorithm

Figure 6.2: Quick fix locations searching and repair generation algorithm.

```
Input: Satisfiable program execution paths set $S_{Paths} := \{s_k | 0 \leq k \leq n, \forall n \geq 0\}$
Output: Refactorings set $R_{set} := \{r_j | 0 \leq j < 2\}$
$W_{set} := \{w_k | 0 \leq k \leq n, \forall n \geq 0\}$; // set of working lists, k'th list
$N_{set} := \{n_t | 0 \leq t \leq n, \forall n \geq 0\}$; // set of nodes
$N_{set} := \emptyset; W_{set} := \emptyset$; // initializing both nodes set and working list set to empty set
$countBP := 0$; $countGQF := 0$; // init.  counters, count buggy paths and generated fixes
$R_{set} := \emptyset$;
while $((Sat_{paths}.hasNext()))$ do
    if $(hasBug(s_k))$ then
        $countBP := countBP + 1$; // count the buggy paths
        $i := startIndex(s_k)$; // set the start index of the path
        $w_k := setWorkList(s_k)$; // set the detected buggy path into the work list
        $NLocs := 1$; // number of quick fix locations
        $C := 0$; // quick fix locations counter
        // if the work list length greater than 0 else skip path
        if $(getLength(w_k) > 0)$ then
            $n_t := initNode(w_k)$; // the node at which the bug was detected
            $N_{set} := N_{set} \cup \{n_t\}$; // add a node for the in-place fix
            $r_j := refact(n_t)$; // create a new bug refactoring
            $R_{set} := R_{set} \cup \{r_j\}$; // add new refactoring to the set R
            while $(i > 0 \wedge C < NLocs)$ do
                $fNode := \{w_{k,i}\}$; // get next node from work list located at index i
                if $(isQuickFixNode(fNode))$ then
                    $n_{t+1} := fNode$; // store current node
                    $N_{set} := N_{set} \cup \{n_{t+1}\}$; // add the node for a non-in-place fix
                    $setConsObject(w_k)$; // store constraint
                    if $(notAffectedPaths(S_{Paths}, n_{t+1}))$ then
                        $pLoc := probLoc(n_{t+1})$;
                        $putMarker(pLoc)$; // put new marker
                        $r_{j+1} := refact(n_{t+1})$; // create a new bug refactoring
                        $R_{set} := R_{set} \cup \{r_{j+1}\}$; // add refactoring
                        $countGQF := countGQF + 1$; // count the generated fixes
                    end
                    $C := C + 1$; // increase non-in-place quick fix locations counter
                end
                $i := i - 1$; // go one step backwards on the path
            end
        end
        $k := k + 1$; // get next satisfiable program execution path
    end
end
```

Algorithm 6.2 is composed of two main phases. (a) Finding the first program execution path which contains the buggy statement and generating the *in-place* quick fix. This quick fix will be suggested to the user in the GUI only if it is sound (*e.g.,* the buffer size is equal on all buggy program executions paths). Note that the buffer index and size are context-sensitive. In case there are different buffer sizes on different paths then in order to preserve the soundness of the patched program a complex *in-place* patch can be generated containing one *if* branch and N *if else* branches (N represents the number of different buffer sizes on each buggy path). Furthermore, the size (LOC) of this patch grows exponentially with the number of paths containing different buffer sizes which renders such a quick fix to be not always practical (*).

(b) Traversing the current selected path in backward program execution order from the location where the bug was found until a *non-in-place* fix location is detected and generating a new quick fix at that location. At this program location the program execution can be safely finished (*e.g.,* exit(EXIT_FAILURE);) (this will not change the program behavior for input which does not trigger the bug) if the buffer index is out of bounds or a numeric value can be set if desired (this will not terminate program execution and most likely will change program behavior). The second quick fix is sound as it can be observed that it does not change program behavior for program input which does not trigger the bug—similar to the first *in-place* quick fix. Quick fix (b) represents an alternative to the first quick fix which is not always feasible (*e.g.,* (*)) and will be suggested in the GUI only if it does not change program behavior for input which does not trigger the bug and for each buggy program execution path at least one *non-in-place* quick fix was successfully generated. This is assessed with the counters *countBP* and *countGQF* indicated in Algorithm 6.2 which must be equal (each buggy path has a *non-in-place* quick fix associated) when the algorithm finishes the search. If the counters are not equal when the search algorithm finishes then there is at least one path where a *non-in-place* bug fix location was not found. Thus, the whole quick fix will not be offered in the GUI since there could exist one program execution path on which the bug was not fixed.

Phases (a) and (b) are repeated for all program execution paths which contain the buggy statement (line number and file name) where the bug was detected. First, the algorithm searches for possible insertion locations (*e.g., in-place* and *non-in-place*) for buffer overflow quick fixes and second, it generates bug fixes. The algorithm uses: *startIndex()* to set the start index from where to search on the initial path, *setWorkList()*, to initialize the buggy first path, *initNode()*, to initialize the node at which the bug was found and *refact()*, to create a new refactoring. Next we extend the notation, S_{paths}, consisting of all program execution paths, W_{set}, used to hold the current selected execution path, N_{set}, is a set of nodes used to store *in-place* and *non-in-place* path nodes (these represent program locations where refactorings will be later on inserted) and R_{set}, is the set of refactorings. In line 3, N_{set}, and, W_{set} are initialized to empty set. In line 6, the algorithm picks a new path from the S_{paths} in each new iteration. Upon verifying that the chosen path contains the buffer overflow bug (previously detected), *hasBug*(s_k), the start index, i, and the initial buggy path, w_k, are initialized. In line 11 and 12, the number of quick fix locations, *NLocs*, is initialised to 1 and the quick fix location counter, C, to 0 respectively. On encountering the condition statement getLength(w_k) > 0, the algorithm checks the working list, w_k, first, its length should be greater than 0 and, second it initializes the node where the bug was found by updating the nodes set, N_{set}. In line 17 a new refactoring is created. After this step the refactorings list located on line 18 is updated. From line 19 to line 35, the algorithm traverses the path backwards in order to find a *non-in-place* fix location of the bug until the index value, i, is greater than 0 and the counter value, C, is less than number of quick fix locations, $NLocs = 1$. While visiting each path node it checks for potential *non-in-place* locations, *isQuickFixNode(fNode)*. Upon encountering a *non-in-place* location, it stores the current node, n_{t+1}, and then N_{set} is updated. This node is used for generating the bug patches. In line 24, the constraint object is set at the index k. The algorithm traverses the current selected program execution path to check if there are any influenced paths using *notAffectedPaths(*S_{Paths}*,* n_{t+1}*)*. At this stage of development a simple check is performed in order to see if the context-sensitive buggy variable appears on the right hand side of an expression (*e.g.,* var = expr.; *e.g., expr.* = *a* binary expression, expr. contains our buggy variable which will be constrained with the patch). Furthermore, it will be checked if the other influenced variables (*e.g.,* var) are dead or live variables along a program path. In future we plan to compute a *distance-bounded weakest precondition* [103] (our engine supports the weakest precondition computation; loop and recursion invariants are not supported) in order to check if program behavior is preserved or not.

In case the algorithm finds no influenced paths then a new refactoring is created and added to R_{set}, line 29. Note that in case of using *exit(EXIT_FAILURE)* in the *non-in-place* quick fix than no check for influenced paths is needed. In line 32, the counter C is incremented by one, this indicates that a second refactoring was created. Next the index value, i, is decremented by one such that the algorithm proceeds one step backwards on the current path in line 34. Note that the algorithm can accommodate the search for more than one *non-in-place* location by increasing the value of *NLocs* and updating the detection rules.

6.4.2 Bug Detection with SMT

Our contribution lies in bridging the gap between a buffer overflow bug report provided by an existing buffer overflow checker and automated generation of one or more quick fixes (quick fix structure, insertion location and values used inside the patches) which remove (automatically assessed by re-running the bug detector on the patched program) the buffer overflow bug.

The bug localization is based on the buffer overflow checker contained in our static analysis engine presented by Ibing *et al.* [111]. The buffer overflow checker returns the location of the bug containing the file name, line number and a unique ID which defines the type of the bug. Based on the bug report ID the following steps are performed automatically. The SMT-Lib constraint system which was used to detect the bug (from the buffer overflow checker) is selected. After obtaining the system, a `SMTConstraintObject` object is instantiated containing the following attributes: the buffer size, the offset and the previous mentioned SMT-Lib constraint system. Next, we introduce the patch creation process consisting of the following seven steps.

Step 1. Input Saturation. Figure 6.1 contains at line four a *non-in-place* quick fix location for the buffer overflow. This bug can be addressed with a missing input check, see lines 5–6 for more details. Due to the missing input check the values of the index variable `data` used in buffer[data] can take values outside of the buffer index interval [0, 9] which leads to a buffer overflow or underflow. In order to determine if the index variable data can take values outside of the allowed interval [0, 9] a SMT constraint system is generated. The SMT constraint system is provided as input to the Z3 [68] SMT solver which will output the message SAT if data can take values outside the allowed interval. In order to remove the buffer overflow bug, we decided to generate two types of quick fixes (*in-place* and *non-in-place*) which are based on the input saturation principle. The input saturation principle basically consists of in limiting the possible values which the index variable data can take to only values which are contained in the buffer index range. The generated quick fixes represent additional checks which limit the upper and lower values of data (see Figure 6.1 for more details). The upper allowed value for data should not be larger than the allowed buffer[data] upper index bound value, 9, and not smaller than 0.

Step 2. SMT Constraint System used for Bug Detection. The original SMT constraint

```
0. (set-logic AUFNIRA)
1. (declare-fun b () Int)
2. (declare-fun c () Int)
3. % c is the buffer size
4. (assert (= c 10 ))
5. (assert (>= b c))
6. (check-sat)
7. (exit)
```

Listing 1: (a) First SMT based oracle.

```
 0. (set-logic AUFNIRA)
 1. (set-option:produce-models true)
 2. (declare-fun saturation () Int)
 3. (declare-fun b () Int)
 4. % c is the buffer size
 5. (declare-fun c () Int)
 6. (assert (= c 10 ))
 7. (assert (>= b c))
 8. (assert (< saturation c))
 9. (assert (>=(saturation (c-1)))
10. (check-sat)
11. (get-value (saturation))
12. (exit)
```

Listing 2: (b) Second SMT based oracle.

Figure 6.3: First and second oracles used for fault detection and repair generation.

system used to detect the buffer overflow bug (excerpt presented in Figure 6.3 (a)) had 317 LOC. We depict only the SMT-Lib statements which matter most in our context and changed the names of the symbolic variables for brevity. During buffer overflow/underflow detection the checker uses SMT-Lib statements which represent path constraints and other specific statements for buffer overflow or underflow checking. The statement in bold font located on line 5 in Figure 6.3 (a) represents the constraint which we get from the our checker (assert (>= data bufferSize)) in case of checking for an buffer overflow. In the case of a buffer underflow check, the checker adds to the constraint system the statement (assert (< data 0)). If one of these two constraints are satisfied, then this means that the variable data can take values outside the range of the buffer. Thus, a buffer overflow or underflow bug report will be issued.

The value of data depicted in Figure 6.3 (a) with b is constraint to be greater or equal 10. The solver answers to this constraint system from Figure 6.3 (a) as SAT. Thus, the set of possible solutions for b is contained in the set $[10, \infty)$. This means that if the program variable `data` takes any value greater or equal to 10 then a buffer overflow bug will be detected. The checker is checking each possible execution path by asking the Z3 solver if the SMT constraint system is satisfiable or not. If a bug report is issued then a `SMTConstraintObject` will be instantiated. The buffer size, buggy variable and the SMT constraint system used to trigger the bug are added as attributes to the previously generated `SMTConstraintObject` object.

Step 3. Bug Type Classification. The bug type classification is based on the checker which was used to detect the bug. The bug checker generates a report containing a unique identifier for each type of bug detected. Currently we have other checkers (information flow checkers [194], infinite loop checkers [113], integer overflow checkers and race condition checkers [112]) which can run in parallel and have unique checker identifiers. We used for our checkers a unique ID which was saved in the checker bug report. Based on the generated bug identifier we can decide which type of bug we are dealing with. After obtaining the unique bug identifier the bug patch pattern is selected which will be used for bug fixing.

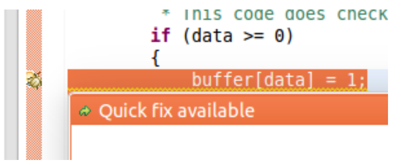
Step 4. Patch Pattern Selection. Based on the bug type classification we select the patch pattern(s) which can be used to fix this type of bug. Our patch patterns consist of empty `C` code skeletons where certain values have to be computed based on the SMT solver used (*e.g.,* (1) +if (buff_size > N || buff_size < 0); *e.g.,* (2) +if (buff_size <= N && buff_size >= 0);). N represents the buffer size determined during static analysis. Note, that N can have different values on different program execution paths.

Step 5. Constraint Values Selection. We construct our SMT constraint system based on the attributes stored in the `SMTConstrintObject` object. These attributes have to be added to the SMT constraint system which was used to detect the bug. After solving the SMT constraint system we will obtain the numeric values which will be inserted into the previous selected patch patterns.

Step 6. Generating SMT Constraint Values. The generation of the constraint values is based on the previously stored SMT-Lib system as an attribute of the `SMTConstraintObject`. The new SMT constraint system (see Figure 6.3 (b)) contains the same SMT statements presented in Figure 6.3 (a) plus some new SMT-Lib statements used to perform the calculation of the needed value which will be later on used inside our selected patch(es). Note that the newly added SMT-Lib statements are marked with bold font in Figure 6.3 (b). The added SMT-Lib statements are used to perform input saturation on the variable `data`. The solver answers to this constraint system from Figure 6.3 (b) as `SAT`. After solving the generated SMT system, we obtain the value 9 for `data`. This value will be used later when we generate our final code patches. `b` represents the symbolic variable `data` and the symbolic variable `saturation` represents our constraint variable used to constrain the variable `data`. The symbolic variable `saturation` is used to constrain the solution space of the real variable (source code variable) `data`. The symbolic variable `saturation` can have the value 9 as single possible legal solution.

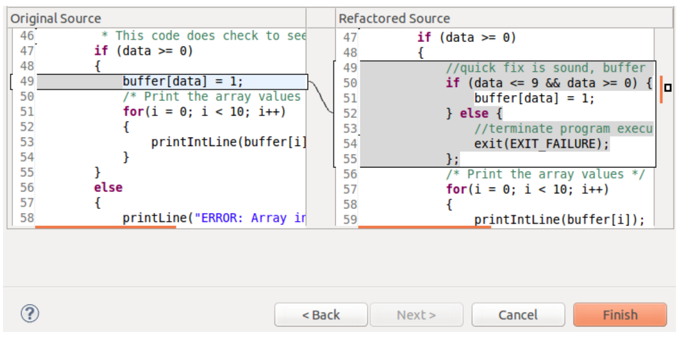
Step 7. Generating Final Code Patches. After solving the constraint system from Figure 6.3 (b) we obtain the numeric value `9` as solution for the symbolic variable `saturation`. The value 9 will be inserted in the previously selected patch patterns in order to constrain the possible values which `data` can take. After this step, we obtain code patches which are syntactically correct, can be compiled and could be further on edited after insertion if desired.

6.4.3 Semi-Automatic Patch Insertion Wizard



```
 * This code does check
if (data >= 0)
{
    buffer[data] = 1;
```

(a)



Original Source

```
         * This code does check to see
        if (data >= 0)
        {
            buffer[data] = 1;
            /* Print the array values
            for(i = 0; i < 10; i++)
            {
                printIntLine(buffer[i]
            }
        }
        else
        {
            printLine("ERROR: Array in
```

Refactored Source

```
        if (data >= 0)
        {
            //quick fix is sound, buffer
            if (data <= 9 && data >= 0) {
                buffer[data] = 1;
            } else {
                //terminate program execu
                exit(EXIT_FAILURE);
            };
            /* Print the array values */
            for(i = 0; i < 10; i++)
            {
                printIntLine(buffer[i]);
```

(b)

Figure 6.4: Patch insertion GUI based wizard.

Figure 6.4 (a) depicts a bug marker indicating the position of the buffer overflow. This marker was placed by the buffer overflow checker and is depicted with a yellow bug icon, on the left of the `C` statement which potentially contains a buffer overflow bug. Next, by pressing on this bug marker, the user can start the code refactoring wizard. The code refactoring wizard is composed of two user pages. The first user page is used to make patches selections (*in-place* or *non-in-place* fix, only one can be selected at a time). The second page depicted in the Figure 6.4 (b) contains a differential files view presenting the differences between the original file containing the bug and the modified file with the selected patch(es) inserted. The user has the possibility to navigate between this two pages using the buttons *Back*, *Next* and *Cancel* in order to compare the results of applying the *in-place* or the *non-in-place* quick fix. Finally, by pressing on the *Finish* button the user accepts the selected quick fix, the patch will be written in the file and the wizard will be stopped.

6.4.4 Implementation

We have integrated BUFFREPAIR into our existing Static Analysis Engine (SAE) [111] which is developed as an Eclipse IDE plug-in. We implemented a refactoring wizard based on the Eclipse Language Tool Kit (LTK), JFace and CDT in order to insert the generated bug patches into the buggy program in a semi-automated fashion. Our bug patching technique is composed of two steps. First, the bug detection analysis is performed. If the bug is detected then this will be marked with a marker. Second, the bug fixing algorithm starts to search backwards on the buggy path until it detects a first *non-in-place* location. If such a bug fix location is found then BUFFREPAIR marks visually the location in code with another marker. The backward searching algorithm can be easily updated in order to accommodate the suggestion of multiple quick fixing locations which can be addressed with other techniques than input saturation.

6.5 Evaluation

6.5.1 Experiments Methodology

We ran BUFFREPAIR on each of the programs and generated two types of patches used for fully automatically fixing the detected faults. We used our previously developed buffer overflow checker for fault detection and classification. The time needed to generate the patches and the total time needed to run the fault detection were measured in milliseconds and then converted to seconds [s]. We used as test system a 64-bit Linux kernel 3.13.0-32.57, Intel i5-3230 CPU at 2.60GHz running on 4 CPU threads. Note that we replaced in the sound `non-in-place` quick fix depicted in Listing 6.1 the string `exit(EXIT_FAILURE);` with `data = 9;`, which is equivalent to `data = (bufferSize - 1);` (bufferSize can have different values for different program paths) in order to see if our apporach for detecting affected paths works. Note, that by using the `non-in-place` quick fix depicted in Listing 6.1 (contains `exit(EXIT_FAILURE);` instead of `data = 9;`) the program behavior is preserved w.r.t. program input which does not trigger the fault. We evaluated our approach by addressing three research questions (RQs).

RQ1: What is the overall computational overhead of BUFFREPAIR? (Section 6.5.2) We wanted to determine the overhead introduced by BUFFREPAIR w.r.t. the fault detection time.

RQ2: Are the generated patches useful for fault fixing? (Section 6.5.3) We wanted to find out if the final generated patches containing the values obtained from the Z3 solver are useful for the fault fixing.

RQ3: Is the behavior of the patched program preserved? (Section 6.5.4) We wanted to find out if the generated patches change the program behavior.

6.5.2 Performance

We addressed RQ1 by measuring the performance of BUFFREPAIR in terms of the patch generation overhead compared with the fault detection time.

Test Programs	#LOCS	# Paths	# Affected Paths	# Nodes	# *non-in-place* Locations	Patches Generation [s]	Prevented
CWE-121 memcpy	1,980	39	0	2,918	18	0.424	✓
CWE-121 fgets	8,771	641	20	231,337	38	0.755	✓
Total	10,751	680	20	234,255	56	1.197	✓

Table 6.1: Bug detection and patches generation results.

Test Programs	Bug Detection + Patch Generation [s]	GCC Recompile Time [s]	Total [s]	GCC Compilation [s]	Ratio
CWE-121 memcpy	21.454	2.813	24.267	2.813	8.6x
CWE-121 fgets	178.276	6.713	184.989	6.713	27.5x
Total	199.730	9.526	209.256	9.526	36.1x

Table 6.2: Comparison of time cost between our system and GCC.

Chapter 6

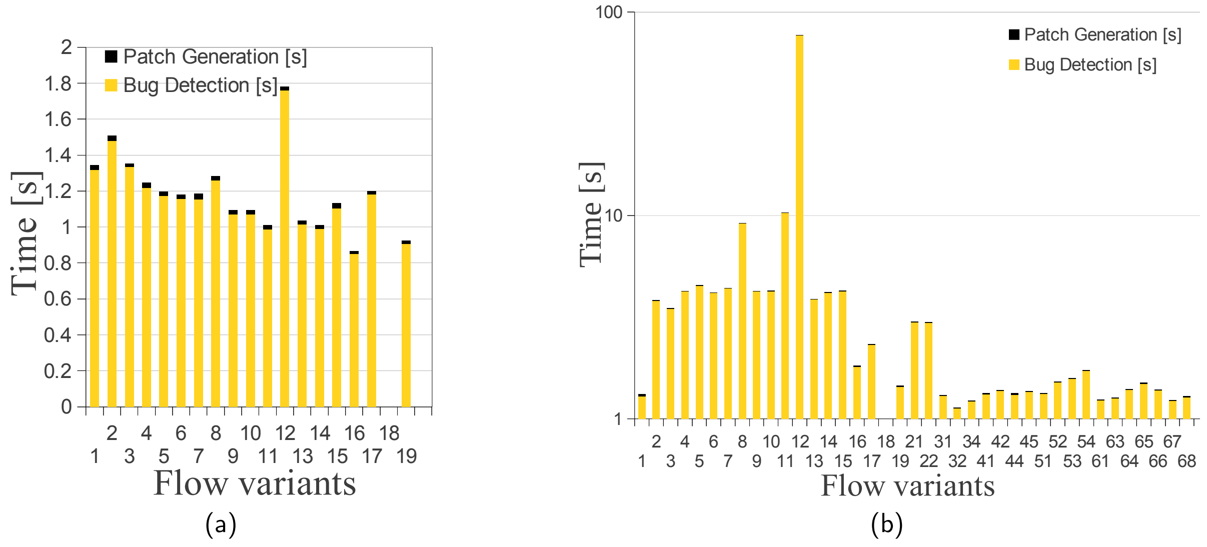


Figure 6.5: Quick fix generation for memcpy and fgets programs.

Figure 6.5 (a) presents the results of running BUFFREPAIR on 19 `memcpy` programs contained in CWE-121. The introduced overhead is calculated by comparing the times represented with black bars (patch generation time) located on top of the yellow bars (fault detection time). The total overhead of 1.97% was obtained by comparing the fault detection time, 21.030 [s] $(21.454[s] - 0.424[s])$, and the patch generation time, 0.424 [s], column seven of Table 6.1.

Figure 6.5 (b) contains the results obtained during patch generation for the 39 `fgets` programs contained in CWE-121. We used the same index enumeration which was used in the open source Juliet test suite [206] in order to have a clear mapping between analyzed programs and programs extracted from the test suite. In comparison with the Figure 6.5 (a) we used in Figure 6.5 (b) a logarithmic scale in order to make the results more readable. From Figure 6.5 (b) we observe that the patch generation times indicated with black bars on top of the yellow bars are considerably lower than the fault detection times indicated with yellow bars. The total overhead of 0.4% was obtained by comparing the fault detection time, 17,7521 [s], $(17.8276[s] - 0.755[s])$ and the patch generation time (0.755 [s]) contained in column 7 of Table 6.1.

The obtained results show that the patch generation time grows by a factor less than 2 (from 0.424 [s] to 0.755[s]) if the number of execution paths increases by a factor of 16,4x (641/39, see Table 6.1 3rd column) and the number of nodes by a factor of 79,2x (231337/2918, see Table 6.1, 5th column). We demonstrated that our approach is applicable to open source `C` programs and the induced overhead is under 1%.

Figure 6.6 presents the overall overhead with yellow bars (the fault detection time) for the `fgets` and `memcpy` programs. The black bars on top of the yellow bars represent the overhead introduced by the patch generation algorithm for the `fgets` and `memcpy` programs. The patch generation overhead is 1.197 [s] which represents 0.59 % from the fault detection time of 199,730 [s] indicated in column 7 of Table 6.1 and in column two of Table 6.2.

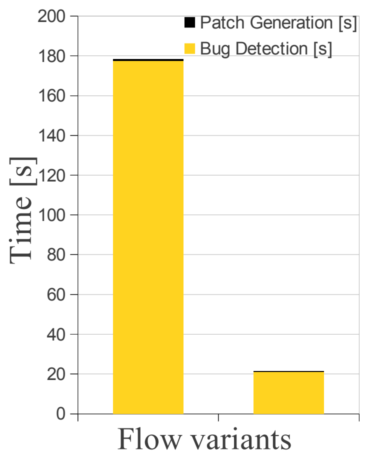


Figure 6.6: Total overhead.

Table 6.2 shows that there is no compilation difference between the patched programs and the un-patched programs. This is because our patches have a small size and introduce no compilation overhead. We observed an overhead decrease from 1,97% (memcpy programs) to 0.4% (fgets programs) for 79.27 times (231,337/2,918, see Table 6.1, 5th column) more nodes and for 16,4 times (641/39, see Table 6.1, 3rd column) more paths. With regard to RQ1, the results confirm that the patch generation overhead is 0.59% when compared to the fault detection time.

6.5.3 Repair Usefulness

We addressed RQ2 by considering following scenarios: First, the syntactical correctness of our generated patches and if the code can be recompiled after the patch was inserted. Second, if the fault patch was useful for removing the detected fault. Third, the usefulness of the *non-in-place* patch which is depicted in Table 6.3, column four. Table 6.3, column 2 shows if the resulted program after the insertion of the *in-place* or the *non-in-place* patches remained compilable. Columns 3 and 4 depicted in Table 6.3 indicate if the fault was removed by inserting the patch *in-place* (fault location) or at the *non-in-place* location.

Test Programs	**Recompile**	***in-place* Fix**	***Non-in-place* Fix**
`CWE-121 memcpy`	✓	✓	✓
`CWE-121 fgets`	✓	✓	✓*

Table 6.3: Bug fixing results.

Table 6.3, column 3 shows that all the fault could be removed by inserting the patch at the place where the fault was detected. Table 6.3, column 4 shows that all the faults were removed by inserting the patch at the *non-in-place* location except the ones indicated with ✓*. The notation `N (M)` was used to denote the control flow variant *N* and the number of detected affeted paths, *M* contained in the Juliet test case CWE_121_fgets [206]. In total 8 `C` programs: 42 (3), 45 (2), 61 (1), 63 (4), 64 (5), 66 (2), 67 (1), 68 (2) contained 20

((3)+(2)+(1)+(4)+(5)+(2)+(1)+(2)) affected paths. An affected path contained at least one usage of the constrained variable (*e.g.,* `data`) in another statement as the path was traversed in program execution order. Thus, the program behavior can be in this way influenced by the set of values that the constraint variable can take after it was constrained. Note, that this is not a sufficient condition to guarantee soundness. Thus, the results presented in Table 6.3 confirm RQ2, that the generated fault patches were useful for removing the faults.

6.5.4 Program Behavior Preserving

Test Programs	# Programs	# IPrograms	# IPaths	% Ratio
`CWE-121 memcpy`	18	0	0	0
`CWE-121 fgets`	38	8	20	14.2
Total	56	8	20	14.2

Table 6.4: Programs behavior preserving.

We addressed RQ3 by checking if the inserted patch at the *non-in-place* location influences other existing program paths. The abbreviations in Table 6.4 mean: Total number of programs containing influenciable paths (IPrograms), Influenciable Paths (IPaths), % Ratio represents the ratio between the total number of programs to the total number of programs containing at least one influenced path. The Algorithm 6.2 used by BUFFREPAIR to generate repairs visits *non-in-place* candidate nodes in backward program execution manner in order to find fault repairing locations. Next, it checks if by patching the found node contained on the affected path the behavior of the program will change. If the algorithm finds an affected path then the *non-in-place* quick fix will be not generated since it could influence other variables contained in the affected paths. We successfully avoided changing the behavior of all the analyzed programs by proposing the fix at the fault location which is indicated in column 3 of Table 6.3. For 14.2 % of the programs (56/8, **# Programs/# IPrograms** presented in Table 6.4 in columns 2 and 3) we avoided changing the behavior by not proposing the *non-in-place* quick fix. Thus, we can confirm that for 85.8% (100% - 14.2%) of the analyzed programs the program behavior did not changed with regard to RQ3.

6.6 Discussion

6.6.0.1 Threats to Validity

In this Section, we discuss the internal and external threats to validity.

Internal Validity. In case we did not interpret the results of our execution measurements right then the overhead of 0.59% could not be achievable. To avoid time measurement mistakes we carefully designed our own time measuring mechanism and repeated the measurements three times for each analyzed program. Some of the decisions we make are static (select type of patch patterns for a fault type) and some are dynamic (SMT constraint system solving). Thus, only the dynamic decisions can influence the overhead introduced by BuffRepair. We are aware that in case the fault checker cannot detect or diagnose the fault type then our approach would suffer from imprecision or would not work at all.

External Validity. We are aware that there are some threats which could hinder our approach from being generalizable for large programs. We think that BuffRepair's repair generation approach can be generalized since we followed the basic automatic program repair steps (failure detection, fault diagnosis, fault cause localization and repair inference). We think that 0.59% overall overhead is negligible and by addressing other types of checks than input saturation or by using other fault patterns no major time increase would be noticeable. Thus, programs containing long execution paths would not increase the overhead significantly with respect to the fault detection time.

6.7 Summary

In this Chapter, we presented BuffRepair, which is based on a novel technique for automatically repairing buffer overflow faults by generating bug repairs using static symbolic execution and SMT solving. Our automatically generated patches do not need any human refinement, are compilable and can be inserted semi-automatically into buggy programs with the help of our refactoring wizard. Our experimental results show that BuffRepair is efficient when detecting and repairing faults. We think that BuffRepair can successfully be applied to high quality projects since the generated quick fixes remove the bug and preserve program behavior. Finally, we think that our approach can be applied in future in conjunction with other types of bug checkers (*e.g.,* [112], [113], [194]) which we developed.

Chapter 7

CastSan: Runtime Detection of C++ Polymorphic Object Type Confusions

In this Chapter, which belongs to the second part of this thesis, we address dynamic memory corruption detections which can most commonly lead to (or are a prerequisite for) CRAs. We were motivated to go this path due to the intrinsic limitations of static analysis tools. More specifically, we developed a compiler based sanitizer tool, called CastSan, which can detect object type confusions during runtime and which is fully integrated into a well established compiler framework (Clang/LLVM). Within this Chapter, we address **RQ4** by providing a fully functional object type confusion tool which is based on a novel and efficient technique for the detection of object type confusion, which—if used consistently,—can considerably reduce the likelihood of CRAs. Finally, note that parts of this Chapter have already been published by Muntean *et al.* [201].

7.1 Introduction

What is the overall view? Real-world security-critical applications (*e.g.,* Google's Chrome, Mozilla's Firefox, and Apple's Safari web browser, *etc.*) rely on the C++ language as main implementation language, due to the balance it offers between runtime efficiency, precise handling of low-level memory, and the object-oriented abstractions it provides. Thus, among the object-oriented concepts offered by C++, the ability to use object typecasting in order to increase, or decrease, the object scope of accessible class fields inside the program class hierarchy is a great benefit for programmers. However, as C++ is not a managed programing language, and does not offer object type or memory safety, this can potentially lead to exploits.

What is the problem? Figure 7.1 depicts statistics w.r.t. object type confusion bugs reported by the United States National Institute of Standards and Technology (NIST) [208]. As of Sept. 2017, 131 type confusion bugs were reported, from which 79 lead to arbitrary code execution. Additionally, 8 illegal casts were found due to bit shifts, which can lead to arbitrary code execution as well. Specifically, this figure indicates that bad type casting is a serious concern. Bad type casting errors are based on a combination of lack of type safety and memory

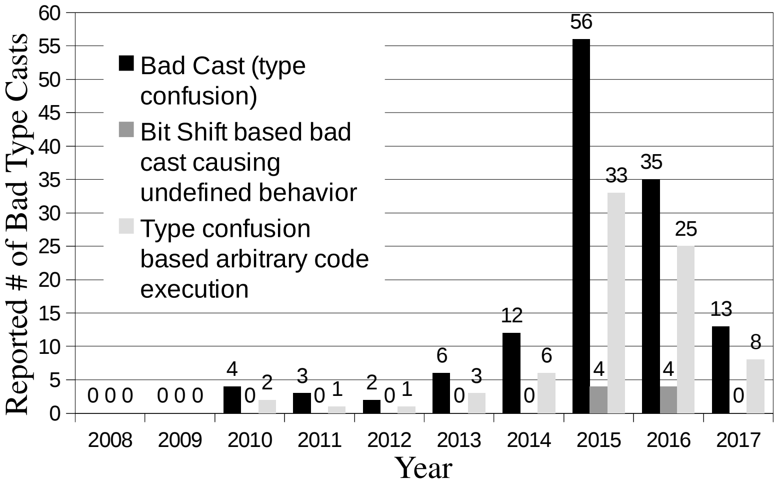


Figure 7.1: Ten years of object type confusion vulnerabilities.

safety. These vulnerabilities were reported in highly used real-world software such as: Google's Chrome and V8 JS engine, Mozilla Firefox, Microsoft's IE 10, Edge, and Chakra JS engine, Adobe Flash Player, & iOS/MacOS apps.

C++ object type confusions are the result of misinterpreting the runtime type of an object to be of a different type than the actual type due to unsafe typecasting. This misinterpretation leads to inconsistent reinterpretation of memory in different usage contexts. A typical scenario, where type confusion manifests itself, occurs when an object of a parent class is cast into a descendant class type. This is typically unsafe, if the parent class lacks fields expected by the descendant type object. Thus, the program may interpret the non-existent field or function in the descendant class constructor as data, or as a virtual function pointer in another context. Object type confusion leads to undefined behavior according to the C++ language draft [119]. Further, undefined behavior can lead to memory corruption, which in turn leads to exploits such as code reuse attacks (CRAs) [31] or even to advanced versions of CRAs including the COOP attack [238]. These attacks violate the control flow integrity (CFI) [1, 2] of the program, by bypassing currently available OS-deployed security mechanisms such as DEP [164] and ASLR [219]. In summary, the lack of object type safety and, more broadly, memory safety can lead to object type confusion vulnerabilities (*i.e.,* CVE-2017-3106 [96]). The number of these vulnerabilities has increased considerably in the last years, making exploit based attacks against a large number of deployed systems an everyday possibility.

What are the existing solutions? Table 7.1 depicts the currently available solutions, which can be used for C++ object type confusion detection during runtime. The tools come with the following limitations: (1) high runtime overhead (mostly due to the usage of a compiler runtime library), (2) limited type checking coverage, (3) lack of support for non-polymorphic classes, (4) absence of threads support, and (5) high maintenance overhead, as some tools require a manually maintained blacklist.

What are the limitations of the solution? We consider runtime efficiency and coverage to be most impactful for the usage of such tools. While coverage can be incrementally increased

Checker	Year	Poly	Non-poly	No blacklist	Obj. Tracking	Threads
UBSan [99]	2014	✓				✓
CaVer [142]	2015	✓	✓	✓	✓	limited
Clang CFI [149]	2016	✓		✓		✓
TypeSan [107]	2016	✓	✓	✓	✓	✓
HexType [125]	2017	✓	✓	✓	✓	✓
CastSan [201]	2018	✓	future work	✓	not required	✓

Table 7.1: High-level feature overview of existing C++ object type confusion checkers.

by supporting more object allocators (*e.g.,* child *obj=`dynamic_cast<*child>(parent)`, `ClassA *obj=new (buffer) ClassA();`, `char *str=(char) malloc(sizeof(S)); S *obj=reint erpret_cast<*S>(str);`, see TypeSan, HexType, for more details.) and instrumenting them for later object type runtime tracking, increasing performance is more difficult to achieve due to the required runtime of type tracking support on which most tools rely. Reducing runtime overhead is regarded to be far more difficult to achieve, since object type data has to be tracked at runtime and updating data structures at runtime (*i.e.,* red-black trees, *etc.*) has to be performed during a type check. As such, due to their perceived high runtime overhead, the tools currently available do not qualify as production-ready tools. Furthermore, the per-object metadata tracking mechanisms generally represent an overhead bottleneck in case the to-be hardened program contains: (1) a high volume of object allocations, (2) a large number of memory freeing operations, (3) frequent use of object casts, (4) *exotic* object memory allocators (*i.e.,* Chrome's `tcmalloc()`, object pool allocators, *etc.*) for which the detection tool implementation has to be constantly maintained.

	Programs		
Checker	soplex (C++)	xalancbmk (C++)	astar (C++)
Clang-CFI [149]	5.03%	4.49%	0.9%
CastSan [201]	2.07%	1.78%	0.3%
Speed-up	2.42 times	2.52 times	3 times

Table 7.2: Object type confusion detection overhead for SPEC CPU2006 benchmark.

Table 7.2 depicts the runtime overhead of CastSan compared against Clang-CFI. CastSan has significantly lower runtime performance overhead.

What is our insight? In this Chapter, we present CastSan, an Clang/LLVM compiler-based solution, usable as an always-on sanitizer for detecting all types of polymorphic-only object type confusions during runtime, with comparable coverage to Clang-CFI [149]. Its technique is based on the observation, that virtual tables (vtables) of polymorphic classes can be used as a successful replacement for costly metadata storage and update operations, which similar tools heavily rely on. Our main insight is that: (1) program class hierarchies can be used to store object type relationships more effectively compared to Clang-CFI's bitsets, and (2) the Clang-CFI bitset checks can be successfully replaced with more efficient virtual pointer based range checks. Based on these observations, the metadata that has to be stored and checked for each object during object casting is reduced to zero. Next, the checks only require constant checking time due to the fact that no additional data structures (*i.e.,* TypeSan and HexType use both red-black trees for storing relationships between object types) have to be consulted during runtime. Finally, this facilitates efficient and scalable runtime vptr-based range checks.

CastSan performs the following steps for preparing the required metadata during compile time. First, the value of an object vptr is modified through internal compiler intrinsics such that it provides object type information at runtime. Second, these modified values are used by CastSan to compute range checks that can validate C++ object casts during runtime. Third, the computed range checks are inserted into the compiled program. The main observation, which makes the concept of vptr based range checks work, is that range checks are based on the fact that any sub-tree of a class inheritance tree is contained in a continuous chunk of memory, which was previously re-ordered by a pre-order program virtual table hierarchy traversal.

CastSan is implemented on top of the LLVM 3.7 compiler framework [151] and relies on support from LLVM's Gold Plug-in [153]. CastSan is intended to address the problem of high runtime overhead of existing solutions by implementing an explicit type checking mechanism based on LLVM's compiler instrumentation. CastSan's goal is to enforce object type confusion checks during runtime in previously compiled programs. CastSan's object type confusion detection mechanism relies on collecting and storing type information used for performing object type checking during compile time. CastSan achieves this without storing new metadata in memory and by solely relying on virtual pointers (vptrs), that are stored with each polymorphic object.

We evaluated CastSan with the Google Chrome [98] web browser, the open source benchmark suite of TypeSan [107], the open source benchmark programs of IVT [24], and all C++ programs contained in the SPEC CPU2006 [252] benchmark. The evaluation results show that, in contrast to previous work, CastSan has considerably lower runtime overhead while maintaining comparable feature coverage (see Table 7.1 for more details). The evaluation results confirm that CastSan is precise and can help a programmer find real object type confusions.

What are our contributions? In summary, in this Chapter, we make the following contributions:

- We develop a novel technique for detection of C++ object type confusions during runtime, which is based on the linear projection of virtual table hierarchies.

- We implement our technique in a prototype, called CastSan, which is based on the Clang/LLVM compiler framework [151] and the Gold plug-in [153].

- We evaluate CastSan thoroughly and demonstrate that CastSan is more efficient than other state-of-the-art tools.

7.2 Threat Model

The threat model used by CastSan resembles HexType's threat model. Specifically, we assume a skilled attacker who can exploit any type of object type confusion vulnerability, but who does not have the capability to make arbitrary memory writes. CastSan's instrumentation is part of the executable program code and is thus assumed to be write-protected through data execution protection (DEP) or another mechanism. Further, CastSan does not rely on information hiding; as such the attacker is assumed to be able to perform arbitrary reads. This is not a limitation, as CastSan does not rely on randomization or code shuffling like other CFI schemes [59, 289]. As CastSan focuses exclusively on `C++` object down-cast type confusions, we assume that other types of memory corruptions (*i.e.,* buffer overflows, *etc.*) are combated with other types of protection mechanisms and that CastSan can work along these complementary defense mechanisms. Finally, we assume that for any large existing source code base, which is affected by object type confusions (*e.g.,* [47]), we assume that this fault cannot currently be fixed solely by inspecting the source code statically or manually and that the attacker has access to the source code of this vulnerable application.

7.3 Design and Implementation

In Section 7.3.1, we present the architecture of CastSan, and in Section 7.3.2, we explain how virtual table inheritance tree projections are used by CastSan, while in Section 7.3.3, we describe our object type confusion detection checks. Finally, in Section 7.3.4, we outline CastSan's implementation.

7.3.1 Architecture Overview

Figure 7.2 depicts the placement of CastSan's components within the Clang/LLVM compiler framework and the analysis flow, indicated by circled numbers.

CastSan's main analysis steps. CastSan instruments object casts as follows: (1) source code files are fed into the Clang compiler, which adds several intrinsics needed to mark all possible cast locations in the code, (2) CastSan uses the vtable metadata and the virtual table hierarchies, which were embedded in each object file in the Clang front-end, (3) placeholder intrinsic-based instructions are used for recuperating the vptr and the mangled name of the object type which will be later cast, and (4) placeholder intrinsic-based instructions for the final pre-cast checks are inserted, containing the per object cast range. The intrinsics will be removed before runtime and will be converted to concrete instruction sequences used to perform the object type cast check. The placeholder intrinsics are used by CastSan since part of the information needed for the checking of illegal casts is not available during compile time (the vptr value is computed during runtime). Finally, during link time optimization (LTO) [150], the following operations are performed: (1) the virtual table hierarchy is constructed

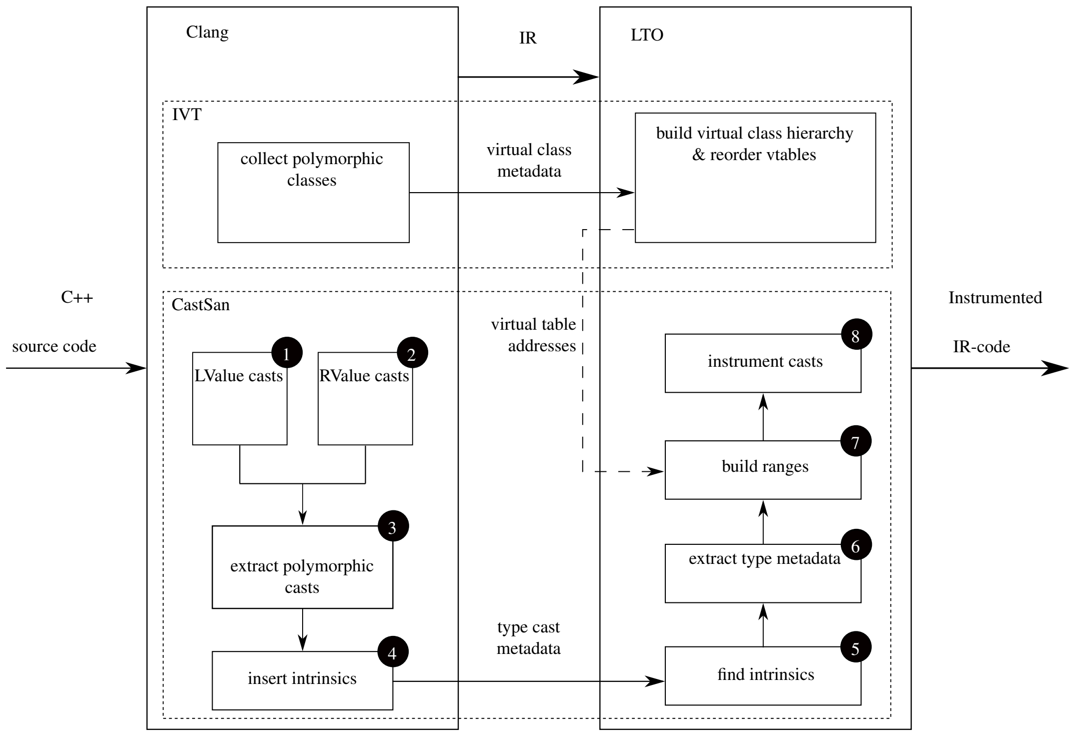


Figure 7.2: CASTSAN system architecture.

and decomposed into primitive vtable trees, and (2) the placeholder intrinsics used to check for down-cast violations are inserted based on the analysis of the previous primitive vtable trees.

Building virtual pointer based range checks. First, the `LValue` (LLVM data type) ❶ and `RValue` (LLVM data type) ❷ casts are instrumented inside the Clang compiler with additional C++ code. Second, only the polymorphic casts are selected from these casts ❸. Third, the polymorphic casts are flagged for instrumentation using an LLVM intrinsic ❹ during LTO. Fourth, the intrinsics inserted by CASTSAN with the help of Clang are detected ❺ for later usage during LTO. Fifth, the metadata of the intrinsics is read out ❻ to acquire all necessary information about an object cast-site. Sixth, the ranges necessary for checking object type confusions are built in ❼. Note that an object range is computed by using the virtual address of the object destination type and the number of all nodes (vtables) inheriting from the destination type. Finally, the object cast-sites are instrumented with a range check ❽.

7.3.2 Virtual Table Inheritance Tree Projection

CASTSAN computes virtual table inheritance trees for each class hierarchy contained in the analyzed program. Next, CASTSAN uses these vtable inheritance trees to determine if the ancestor-descendant relation between the types of the cast objects holds. The ancestor-descendant relations between object types rely on several properties of these ordered vtable inheritance trees, which we will explain next. The root of such a virtual table inheritance tree is a polymorphic class that does not inherit from other polymorphic classes (root type). Note that a class has only one vtable associated to it. Further, each vtable is broken into multiple primitive vtables. Also note that these vtables can occupy different places in this ordering. The children of any node in the vtable tree are all types that directly inherit from the ancestor class and are located underneath this class in the program class hierarchy. If a class inherits from multiple vtables, it has a node in any tree that the ancestor types are a part of. The leaves of a vtable tree are vtables which have no descendants. CASTSAN will put the vtables that are in any type of a descendant-ancestor relation to each other in a single virtual inheritance tree. Next, we show how a virtual table projection list is computed.

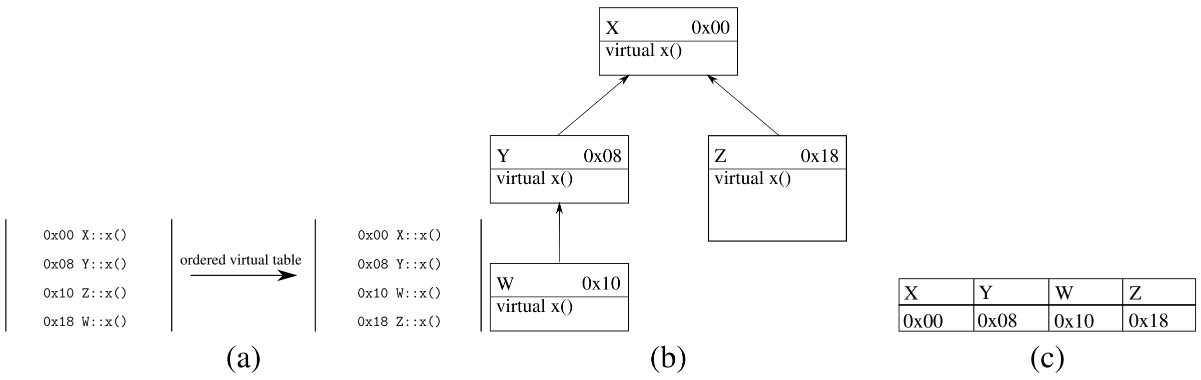


Figure 7.3: Unordered & ordered (a) vtables of the tree rooted in X. (b) contains the vptr of each type after ordering. (c) depicts the projected list corresponding to (b).

Figure 7.3(a) depicts the memory layout of the vtables of the class represented by the primitive hierarchy in Figure 7.3(b). The vtables contain their addresses laid out in memory (*i.e.,* consider address `0x08`) along with the pointers to the virtual functions (*i.e.,* `Y::x()`). Note that in the unordered table located on the left side of Figure 7.3(a), there is no relationship between the addresses of the vtables and the class hierarchy. For simplicity reasons, in Figure 7.3(a) we opted to depict each box of the vtable hierarchy to contain a single entry. In general, when multiple entries in each vtable are contained in the vtable hierarchy, the vtables will be interleaved to ensure that their base pointers are consecutive addresses in memory. After ordering the values of the addresses of the vtables (right table in Figure 7.3(a)) the addresses are in ascending order (*e.g.,* `W` inherits from `Y` directly, thus it comes directly after `Y` in the vtable). Further, after interleaving the addresses of the vtables, their values are in ascending order corresponding to the depth-first traversal, as shown in the projected list depicted in Figure 7.3(c). Next, CASTSAN uses a pre-order traversal of each vtable inheritance tree in order to construct a list of vtables, which represents a projection of a tree hierarchy onto a list. For example, if the type of a vtable (first row in a box, see Figure 7.3(b)) is the descendant of another type, it is inserted after the other type in the list. Further, any sub-tree of each tree is represented as a continuous sub-list of virtual tables by CASTSAN. This means that the types that inherit from the root type of the sub-tree will be inserted into the list in direct succession to the sub-tree root. Finally, the projected list will be used to compute object cast ranges which will subsequently be used to determine legal and illegal relations between the object types during a cast operation.

7.3.3 Object Type Confusion Detection

7.3.3.1 Virtual Pointer Usage as Runtime Object Type Identifier

CastSan uses the virtual pointer (vptr) of an object to identify its type at runtime. Note that any polymorphic type contains a set of virtual methods that are reachable from any object using its vptr. The vptr of a type is saved in any polymorphic object that is created using the type's constructor. By type constructor, we mean the function which is called when an object of a certain type is allocated. Furthermore, note that each legally cast instance of a polymorphic object can be uniquely identified by its vptr since the vptr of an object is always the first field of that object. CastSan therefore reads the vptr of any object at runtime to uniquely identify its runtime type. CastSan does this by loading the first 64-bit of the object into a register using an intermediate representation (IR) load instruction. This load instruction is inserted by CastSan during LTO for runtime usage.

7.3.3.2 Determining Object Type Inheritance during Runtime

As previously mentioned, CastSan checks object casts by using the projected virtual table hierarchy list (see Figure 7.3(c) for more details). A projected class hierarchy consists of ordered vtable addresses. The runtime type of an object must inherit from the destination type of the cast in order for the cast to be legal. This happens if the vtable of the runtime type is a child in the sub-tree of the vtable of the destination type. Further, if this is the case, the runtime type comes after the destination type in the depth-first list of the tree. Since all nodes of a sub-tree are placed successively in the projected list, this means that these nodes are located before the last element of the sub-tree in the list. Therefore, CastSan does not need to traverse the whole sub-list representing the sub-tree of the destination type to check if the runtime type is part of it. It is enough to check whether it is anywhere between the first and the last element in the list. This holds because the type of the object holding the vptr has to have a vtable in the sub-tree of the destination type, which means it inherits from the destination type. Otherwise, if the vptr is not in the range, it has no vtable inheriting from the vtable of the destination type and therefore its type does not inherit from the destination type. Therefore, the object cast is illegal in this situation. CastSan implements this mechanism at runtime using range checks on the vtable pointer of an object and additionally by using the values of the vtable addresses of the destination type sub-tree. During runtime CastSan checks if the value of the vptr is larger than the vtable address of the destination type and smaller than the address value of the last vtable entry located in the sub-list corresponding to the destination type. If this holds, then the runtime type must inherit from the destination type; therefore, the cast is legal. Otherwise, if the vptr value is not contained between the above mentioned boundaries, then the runtime type does not inherit from the destination type, thus the object cast is not legal.

7.3.3.3 Virtual Table based Range Checks

CastSan uses vtable based range checks in order to check if the vptr of an object resides between two allowed values. CastSan's range check is based on the observation that the addresses of the ordered vtables are re-arranged by interleaving them through a pre-order traversal of the inheritance trees in which these vtables are contained. Therefore, the addresses of any sub-tree lay continuously and gapless in memory. By continuously and gapless we mean that there is no starting address of another vtable not belonging to the sub-tree in between the addresses of a sub-tree, and the starting addresses of the vtable lie consecutively in memory, respectively. Further, if the vptr points to any address between the first and the last address of the sub-tree, then it has to be in the list of all addresses located in the sub-tree and therefore the cast is legal. In this way, CastSan can simplify the type check to a range check. CastSan builds a range check by using the vtable address V of the destination type X and the number c of all classes that inherit from X. V and c can be determined statically at compile time for each object cast performed in the program. To perform the check at runtime, the vptr value P is extracted from the object before the cast. Next, the following expression is evaluated by CastSan during runtime. *If* $V + c \geq P \geq V$ holds, then the cast is legal, otherwise the cast is illegal and program execution will be terminated or an error log output can be produced depending on the employed CastSan usage mode flag. Note that CastSan offers the possibility to include in the *else*-branch of the inserted cast check the option to log back-trace information instead of terminating the program which is obviously not always desired (see Figure 7.4 for more details).

The generated object cast range check has the following advantages compared to other state-of-the-art techniques. First, in terms of memory overhead, CastSan does not require any additional metadata at runtime to be recorded, deleted or updated in order to determine class hierarchy relationships. Second, the range check needed for the sub-typing check has $O(1)$ runtime cost compared to $O(n)$ runtime cost of other tools due to traversals of additional data structures (*e.g.,* red-black tree).

7.3.3.4 Instrumenting a C++ Object Cast

CastSan replaces the cast check intrinsics inserted into the code within the Clang compiler with a range based cast check (see ❽ depicted in Figure 7.2 for more details) during LTO. The check is substituted with an equality check if the count of vtables in the range is one. The equality check matches the vtable address of the range with the vptr of the object. If the addresses are equal, then the cast is legal, otherwise it is illegal. In case the range has more elements than one, then a range check will be inserted. The steps for building and inserting the final range check are as follows. First, the value of the start address of the range is subtracted from the vptr value by CastSan. Further, if the pointer value was lower than the start address of the vtable, then the result is negative and the cast is illegal. Second, the result of the subtraction is rotated by three bits to the right to remove the empty bits that define the pointer length. If the result of the subtraction was negative, this rotation shifts the sign of the result to the right, making it the most significant bit. Therefore, if the cast is illegal, then the result of the bit rotation is a large number. More specifically, the number is then larger than any result of a valid cast. This holds because the most significant bit, where the sign was shifted due to the rotation, would have been shifted to the right. This would make the number smaller than the illegal case. The result is either the distance of the destination type from the runtime type within the vtable hierarchy or an invalid large number. Finally, the value is compared to the number of vtables in the range. If the value is less than or equal to the count, then the cast is legal and program execution can continue, otherwise an illegal cast is reported. By using these instructions, the range check can ensure three preconditions for a legal cast using only one branch. If any of the following preconditions do not hold, CastSan will report an illegal cast. This is the case if the value of the vptr is: (1) higher than the last address in the range (*i.e.,* the type of the object is not directly related to the destination type), (2) lower than the first value of the vtable address range (*i.e.,* the runtime type of the object is an ancestor of the destination type), resulting in the negative bit being shifted to a significant bit of the subtraction result, or (3) not aligned to the pointer length (*i.e.,* the pointer is corrupted). Note that in (3) the unaligned bit is rotated to one of the significant bits or to the signing bit. Since the comparison is unsigned, the number would then again be larger than the last address in the vtable range.

Further, note that the vptr of an object can always be used to perform the check in the primary inheritance tree of the object source type. Finally, the primary inheritance tree, represents the tree which contains the virtual table of the object types as primary parent.

```
1 X *x=new W();
2 Y *y=static_cast<Y>(x);
3 y->x();
```

(a)

```
1 0x400fe0 mov   %r15,%rdi
2 0x400fe3 callq *(%rax)
```

(b)

```
1 0x400fc0 ud2
2 0x400fcb mov   $0x401080,%ecx
3 0x400fcf mov   %rax,%rdx
4 0x400fd1 sub   %rcx,%rdx
5 0x400fd6 rol   $0x3d,%rdx
6 0x400fda cmp   $0x2
7 0x400fde ja    0x400fc0
8 0x400fe0 mov   %r15,%rdi
9 0x400fe3 callq *(%rax)
```

(c)

Figure 7.4: Instrumented polymorphic C++ object type cast.

Figure 7.4 (a) depicts a C++ object type cast at line number two. Figure 7.4(b) presents the un-instrumented assembly code, and the assembly code instrumentation added by CastSan is depicted in Figure 7.4(c) (the range check is highlighted in gray shaded color). In Figure 7.4(a), without line number three the compiler does not generate code since the Clang/LLVM compiler is designed to not generate specific code for object casts. Only for the object dispatch (see line number three), assembly code is generated. The assembly code in Figure 7.4(b) corresponds to the object dispatch depicted in Figure 7.4(a) at line number 3. Finally, we assume that the OS provides an $W \oplus X$ protection mechanism (*e.g.,* data execution prevention (DEP)) and thus the assembly code depicted in Figure 7.4(c) cannot be modified (rewritten) by an attacker.

Next, we present the operations performed by the instructions contained in the range check (gray shaded code in Figure 7.4) in order to better understand how the check operates. First, the vtable address of type X (corresponding to line number one in Figure 7.4(a)) `0x401080` is loaded. In line number two, in Figure 7.4(c), the fixed value of the address is moved to the register `%rcx`. This is done in order to load the first value of the range. Second, the vptr of the object *x* is moved to register `%rdx` depicted in line number three. This is done in order to provide the second value of the subtraction of the range check. Note that the object pointer itself was already loaded in register `%rax`. This is not depicted in Figure 7.4 for reasons of brevity. Third, the `sub` instruction performs the subtraction of the vtable address (stored in `%rcx`) from the vptr (stored in `%rdx`). At line number five, depicted in Figure 7.4(c), the pointer alignment is removed from the result by using a rotation (*i.e.,* `rol`) instruction. This is done to obtain the distance of the vptr from the vtable address of the destination type located in the vtable hierarchy. Note that if the number of all types inheriting from the destination type is higher or equal to the distance, the cast is legal. Finally, the result is compared to the constant `$0x2`, which is the number of all types inheriting from the destination type *Y*, specifically these are *Y* and *W*. Then, the program execution either jumps to the address of the instruction `ud2` located at line number one in Figure 7.4(c) (address `0x400fc0`), which terminates the program; otherwise, the object dispatch (line number three in Figure 7.4(c)) will be performed similar as in Figure 7.4(b) and the program continues its execution.

7.3.4 Implementation

Components. CASTSAN is implemented as two module passes for the Clang/LLVM compiler [151] infrastructure by extending LLVM (v.3.7) and relies on the Gold plug-in [153]. CASTSAN is based on the virtual table interleaving algorithm presented by Bounov *et al.* [24] from which it reuses its interleaved vtable metadata by transporting it from the Clang compiler front-end to the LTO phase via new metadata nodes inserted into LLVM's IR code. More specifically, CASTSAN's implementation is split between the Clang compiler front-end, and a new link-time pass used for analysis and generating the final intrinsic based compiler cast checks. CASTSAN's transformations operate on LLVM's intermediate representation (IR), which retains sufficient programming language semantic information at link time to perform whole program analysis and identify all possible types of polymorphic `C++` casts in order to instrument them.

Usage Modes. CASTSAN's implementation provides three operation modes with corresponding compiler flags. First, *attack prevention mode* can be used in shipped program binaries to customers. This mode can be used, if desired, to terminate program execution when an illegal cast is detected, thus providing an effective mechanism for avoiding undefined behavior which may lead to vulnerability based CRAs. Second, *software testing mode* can be used during program testing in order to detect type confusion errors and to help fix them before the software is shipped by subjecting the analyzed program to a test suite with different possible goals (*i.e.,* program path coverage, *etc.*). Finally, *relaxed mode* can be used to detect and log illegal casts detected during development or deployment. This last mode is mainly intended as a replacement for the the situation in which it is not safe to stop program execution which is mostly the case for real-world programs.

7.4 Evaluation

We evaluated CastSan by instrumenting various open source programs and conducting a thorough analysis with the goal to show its effectiveness and practicality. The experiments were performed using the open source benchmarks TypeSan [107], IVT [24], Google's Chrome (v.33.0.1750.112) web browser, and the SPEC CPU2006 benchmark (only for the `C++` based programs), which were also used by HexType [125]. If not otherwise stated, we used the Clang `-O2` compiler flag for all our experiments. In our evaluation, we addressed the following research questions (RQs).

- **RQ1:** What is the **runtime overhead** of CastSan? (Section 7.4.1)
- **RQ2:** How **precise** is CastSan? (Section 7.4.2)
- **RQ3:** How **effective** is CastSan? (Section 7.4.3)
- **RQ4:** How can CastSan **assist a programmer** during a bug bounty? (Section 7.4.4)

Comparison Method. In addition to the runtime overhead and binary blow-up, the coverage and precision of HexType is compared to that of CastSan. For benchmarking SPEC CPU2006, the benchmark script of TypeSan, and the micro-benchmark of ShrinkWrap [106] was used.

Preliminaries. The script of TypeSan (approx. 606 Bash LOC) sets up a full environment consisting of: Binutils, Bash, Coreutils, CMake, Pearl. These are used for instrumenting the SPEC CPU2006, and UBench (consisting of 10 intricate `C++` testcases). After the benchmark is set up, the script compiles the programs and checks each program by starting it and checking it to see if it executed successfully.

The script of IVT (approx. 200 Python LOC) is used to compile up to 50 `C++` programs. Some of the programs contain object type confusions. After each instrumented program execution, the script checks if the program executed successfully or not.

Experimental Setup. We evaluated CastSan on an AMD Ryzen R7 1800x CPU using 8 cores with 16 GB of RAM running the Debian 8 Jessie OS. All benchmarks were executed 10 times to obtain reliable mean values.

7.4.1 Performance Overhead

Table 7.3 depicts the overall runtime overhead on only the relevant `C++` programs contained in the SPEC CPU2006 benchmark. The geomean value of the overhead in these benchmarks is under 1% (0.92%). As an outlier, `soplex` showed an overhead of 2.07%. For most benchmarks, the overhead is lower than 1.0%. Some SPEC CPU2006 benchmarks like `astar` do not contain static casts and thus no check is performed. These results show that the overhead is within the margin of error. This is to be expected as CastSan does not need to execute additional code on execution when no checkable casts are present in the code.

Benchmark	Vanilla	CastSan	Overhead
soplex	207.14	211.43	2.07%
povray	123.34	125.28	1.57%
omnetpp	269.14	270.06	0.34%
astar	334.96	335.96	0.30%
dealII	186.71	188.47	0.94%
xalanckbmk	413.67	421.03	1.78%
namd	266.42	266.43	0.00%
average			1.0%
geomean			0.92%

Table 7.3: Benchmark results of running various C++ programs contained in the SPEC CPU2006 benchmark with CastSan enabled and disabled (vanilla). The values represent the mean time needed to finish running the benchmark program over 10 runs.

Benchmark	High/Low	Vanilla	CastSan	Overhead
gc-sunspider [256]	<	123.4	124.1	0.57%
gc-octane [48]	>	29885	29889	-0.01%
gc-drom-js [75]	>	1987.21	1991.58	-2.18%
gc-balls [30]	>	216	215	0.47%
gc-kraken [134]	<	933.1	941.2	0.87%
gc-jetstream [126]	<	184.06	184.44	0.21%
average				-0.01%
geomean				0.31%

Table 7.4: Runtime overhead on Chrome with CastSan enabled and disabled (vanilla).

Table 7.4 depicts the average and geomean runtime overheads of CastSan in seven of the most popular JavaScript benchmarks. The greater/less symbols (in High/Low) next to the name describe if higher (>) or lower (<) values are better in the benchmark. More precisely, higher is better for jetstream, octane, balls and dromaeo benchmarks; lower is better in sunspider and kraken. The numbers in the columns Vanilla and CastSan represent aggregate benchmark scores and have no particular intrinsic meaning. The average value of the overhead of CastSan in these benchmarks is -0.01%, which is in the margin of error. The low overhead obtained when running JavaScript benchmarks in the instrumented Chrome demonstrates that CastSan can efficiently scale to large code bases with complex class hierarchies.

Figure 7.5 depicts the average and geomean runtime overheads of CastSan in comparison with the Clang-CFI cast checker when ran on several `C++` programs contained in the SPEC CPU2006 benchmark with the following compiler flags: `-fsanitize=cfi- cast-strict`, `-fsanitize =cfi-derived-cast`, and `-fsanitize= cfi-unrela- ted-cast`. Note that the Clang-CFI cast checker instruments the same set of static object casts as CastSan. We compared the Clang-CFI and CastSan runtime overhead w.r.t. the baseline LLVM 3.7 com-

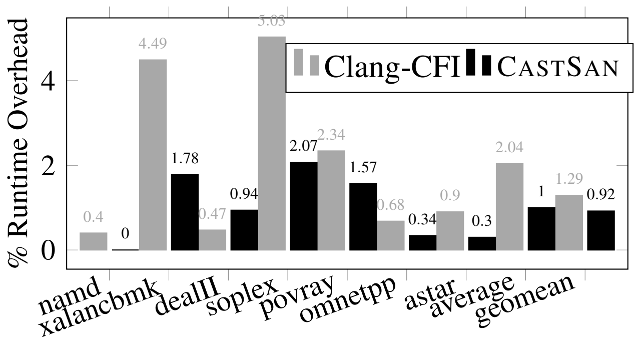


Figure 7.5: Clang-CFI (gray) vs. CastSan (black) SPEC CPU2006 benchmark overhead.

pilations. Note that for the baseline compilation no additional compiler flags and no LTO support (we compiled without the Clang's `-flto` compiler flag) was used. Finally, it can be observed that the overhead of CastSan is about two times lower on average than the overhead of Clang-CFI when running on the SPEC CPU2006 programs.

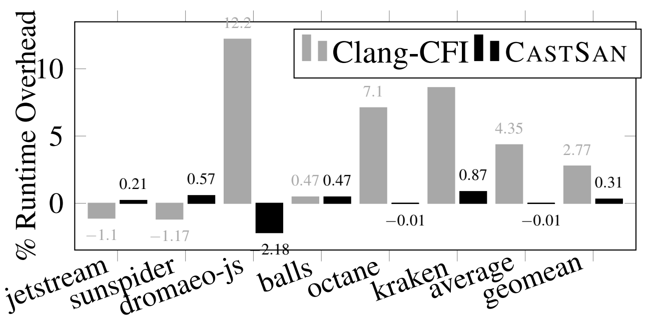


Figure 7.6: Clang-CFI (gray) vs. CastSan (black) Chrome runtime overhead.

Figure 7.6 depicts the runtime overhead of Chrome when ran on several JavaScript benchmarks. First, we compiled it with Clang-CFI, and second, with CastSan enabled and with the following compiler flags enabled: `-fsanitize=cfi-cast-strict`, and `-fsanitize =cfi-derived-cast`. We did not use the `-fsanitize=cfi-unrelated -cast` compiler flag, since Chrome was not able to start (crashed during start) after applying this flag. In total, the same amount of object casts where instrumented by each of the tools. However, we can observe that compared to Clang-CFI, the geomean and average overheads of CastSan are better on large code bases such as the Chrome browser. The lowest runtime overhead value, -2.18%, was obtained with CastSan when running the Dromaeo-js benchmark, while the lowest overhead, -1.17%, was obtained by Clang-CFI when running the Sunspider JavaScript benchmark. Overall, we observed a 54 times speed-up on average and 8.9 times speed-up in geomean for CastSan when compared to Clang-CFI cast checker.

7.4.2 Precision

We evaluated the precision of CastSan by using complex class hierarchies of programs contained in the open-source micro-benchmark of TypeSan [107] and the benchmark programs (in total more than 50 programs) provided by the IVT tool. This benchmark includes: (1) casts to secondary parents, (2) casts within a diamond inheritance, and (3) casts from unrelated trees.

The results indicate that each cast that is covered by CastSan can be precisely checked and the implementation leaves no room for unmitigated corner cases. Moreover, CastSan did not show the imprecisions described by the authors of ShrinkWrap. There, the authors show specific cases of class inheritances (*e.g.,* diamond inheritance) where vtable based function call sanitizers allow calls to illegitimate functions of sibling classes. Finally, CastSan was able to cope with all complex class hierarchies contained in these benchmarks and no false negatives or false positives were reported. Thus, we conclude that CastSan is precise and leaves no space for untreated corner cases.

7.4.3 Effectiveness

We evaluated the effectiveness of CastSan by selecting the last ten type confusions reported in Google Chrome which had common weakness enumeration (CVE) reports associated. All these type confusions have been reported and partially fixed in the current Chrome browser version. The goal of this experiment is to show that CastSan can find object type confusions in real-world software.

We recompiled the Chrome web browser with the CastSan checks in place and ran all JavaScript benchmarks, which we also used to check the performance of Chrome (see Figure 7.6 for more details). In total, out of the ten object type confusions, CastSan was able to report three type confusions at the correct location. We further investigated the other undetected type confusions and found out that these were not detected since the used JS benchmarks do not interact with the code of Chrome which contains these bugs. As such, this is an issue which can be addressed with more extensive test suites which were not detected previously. Finally, we conclude that CastSan is effective in detecting real-world type confusions.

7.4.4 Programmer Assistance

We evaluated how useful CastSan is in helping a programmer to find and fix a type confusion bug. For this reason, we used a well-known type confusion bug and depict the error log in order to show how the programmer is guided when fixing a type confusion bug. The goal of this experiment is to show that CastSan can effectively help a programmer to pinpoint the exact bug location.

Figure 7.7 depicts the backtrace that CastSan prints out when running the `xalancbmk` program contained in the SPEC CPU2006 benchmark. The SPEC CPU2006 `xalancbmk` has a known type confusion vulnerability, as mentioned in [24], which CastSan is able to detect. Thus, on execution, it prints the back-trace leading to the illegal cast. Line numbers 1 to 27

```
 1 ./Illegal Cast Detected. Printing Backtrace:
 2 ./Xalan_base[0x5a3de8]
 3 ./Xalan_base(_ZNK11xercesc_2_511DOMTextImpl13getParentNodeEv+0x6)[0x5cb1c6]
 4 ./Xalan_base(_ZN11xercesc_2_513DOMParentNode12insertBeforeEPNS_7DOMNodeES2_+0x25a)[0x5bae2a]
 5 ./Xalan_base(_ZN11xercesc_2_511DOMAttrImpl8setValueEPKt+0xb0)[0x59e6a0]
 6 ./Xalan_base(_ZN11xercesc_2_512XSDDOMParser12startElementERKNS_14XMLElementDeclEjPKtRKNS_
 7   11RefVectorOfINS_7XMLAttrEEEjbb+0x520)[0x6e15e0]
 8 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner14scanStartTagNSERb+0x1a0f)[0x61134f]
 9 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner11scanContentEv+0x171)[0x60e451]
10 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner12scanDocumentERKNS_11InputSourceE+0x67)[0x60e0c7]
11 ./Xalan_base(_ZN11xercesc_2_517AbstractDOMParser5parseERKNS_11InputSourceE+0x22)[0x5779e2]
12 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner20resolveSchemaGrammarEPKtS2_+0x685)[0x61b8c5]
13 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner19parseSchemaLocationEPKt+0xe0)[0x61b1a0]
14 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner28scanRawAttrListforNameSpacesEi+0x4b7)[0x61ad47]
15 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner14scanStartTagNSERb+0x3b2)[0x60fcf2]
16 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner11scanContentEv+0x171)[0x60e451]
17 ./Xalan_base(_ZN11xercesc_2_512IGXMLScanner12scanDocumentERKNS_11InputSourceE+0x67)[0x60e0c7]
18 ./Xalan_base(_ZN11xercesc_2_517SAX2XMLReaderImpl5parseERKNS_11InputSourceE+0x25)[0x6537f5]
19 ./Xalan_base(_ZN10xalanc_1_828XalanSourceTreeParserLiaison14parseXMLStreamERKN11xercesc_2_
20   511InputSourceERKNS_14XalanDOMStringE+0x120)[0x7c8070]
21 ./Xalan_base(_ZN10xalanc_1_824XalanDefaultParsedSourceC1ERKN11xercesc_2_511InputSourceEbPNS1_
22   12ErrorHandlerEPNS1_14EntityResolverEPKtSA_+0x1da)[0x7cb8ea]
23 ./Xalan_base(_ZN10xalanc_1_816XalanTransformer11parseSourceERKNS_15XSLTInputSourceERPKNS_
24   17XalanParsedSourceEb+0x303)[0x7cc983]
25 ./Xalan_base(_ZN10xalanc_1_816XalanTransformer9transformERKNS_15XSLTInputSourceES3_RKNS_
26   16XSLTResultTargetE+0x2a)[0x7ccaea]
27 ./Illegal instruction
```

Figure 7.7: Type confusion back-trace for the `xalancbmk` program.

are the verbose output of CASTSAN, notifying the user that an illegal cast happened during execution. In lines 25, 26 and 27 the mangled name of the exact function containing the illegal object cast is printed. Using the offset printed in the square brackets at the end of the line, a developer can find the line in the code where the illegal object cast was defined. The error log depicted in Figure 7.7 demonstrates that CASTSAN is able to detect real type confusion bugs in applications by running a program in backtrace-mode. Finally, we conclude that CASTSAN can help developers during bug bounties [293], and can protect against exploitable type confusions.

7.5 Discussion

In this Section, we present CASTSAN's limitations and discuss how these can be addressed.

7.5.0.1 Limitations

Non-Polymorphic Classes. CASTSAN provides type safety for objects stemming from polymorphic classes and low runtime overhead. Further, CASTSAN cannot check casts between non-polymorphic objects. This is because only polymorphic objects have a virtual pointer (vptr). The vptr is an integral requirement for checking object type casts using CASTSAN. This means CASTSAN cannot mitigate all types of object type confusion vulnerabilities. This limitation can be addressed by example by constructing for static classes an artificial virtual-table-like metadata on which CASTSAN's technique can be based such that our technique becomes usable for non-polymorphic object type casts.

Reinterpret-Cast. In C++, not only `static_cast` can lead to object type confusion. The misusage of `reinterpret_cast` can also pose threats. HexType addresses this threat by extending its type cast checking to `reinterpret_cast` in addition to `static_cast`. While this can effectively hinder a type confusion vulnerability from occurring, it is debatable if checking `reinterpret_cast` is viable. This question arises, as `reinterpret_cast` can be used as a legitimate way of breaking class hierarchy boundaries, if the memory layout of the cast types match. In this case, a type cast check based on class hierarchy information cannot be made. Therefore, if `reinterpret_cast` is checked for type safety, its purpose can potentially be circumvented. Similarly, as other object type confusion detection tools handle `reinterpret_-cast`, we could use compiler runtime checking support for checking for this type of confusions.

CastSan Metadata. CastSan is able to perform type cast checks without relying on metadata that has to be maintained during runtime. In order to achieve full object type safety in C++ programs without relaying on runtime metadata it is still an open question. Other solutions, such as HexType, are able to come close to full coverage of all object casts, but incur high overhead and use intens metadata update operations. Finally, CastSan is the first LLVM based sanitizer that is able to achieve a high degree of object type confusion safety without the need storing and updating metadata during runtime.

CastSan Coverage. CastSan coverage is not as high as that of other type confusion mitigation schemes [107, 125], for this reason its relevance needs to be discussed; its performance overhead is under 1% in most cases. Therefore, it is suitable for use in production software. While type safety can not be guaranteed in every case, CastSan still offers a degree of additional safety in contrast to code that is not instrumented at all. Finally, the other solutions that have higher coverage do not offer a performance overhead that is tolerable in production software, but CastSan has a tolerable overhead which proves its relevance and qualifies the tool as relevant and production ready.

Increasing CastSan's Coverage. The incremental research work between TypeSan and HexType shows that the main path for increasing object type confusion detection coverage is to support more types of memory allocators (*i.e.,* jmalloc, tcmalloc, *etc.*) or other more exotic ones. Further, the coverage of CastSan can be increased by supporting all types of C++ program locations (*i.e.,* statement types) where such vulnerabilities could manifest. Thus, CastSan's coverage can be consistently increased by instrumenting all these source code locations with the needed checks in place in order to check during runtime for object type confusions.

Finding New Vulnerabilities. Finding new object type confusion vulnerabilities is directly linked to increasing the tool coverage and is mainly driven by three lines of research. These are: (1) check new previously not possible to be instrumented program locations, (2) support new memory allocators (*e.g.,* object pool allocators, *etc.*), and (3) reduce the runtime overhead of an object type detection technique such that the technique becomes applicable in real-world deployment. Thus, in future work we want to increase the coverage of CastSan by addressing the above mentioned points.

7.6 Summary

C++ object type casting confusions have an important role in modern exploits as demonstrated by recent attacks against Mozilla's Firefox and Google's Chrome web browsers.

In this Chapter, we presented CastSan, a new polymorphic only object type confusion detection tool. CastSan's novel technique is based on an efficient and time constant virtual pointer range check which is possible by extracting virtual table inheritance trees out of a previously constructed virtual table inheritance hierarchy. CastSan constructs linear projections out of virtual table inheritance trees, which are subsequently used do build runtime object cast checks. Our evaluation results show that CastSan is more efficient than state-of-the-art tools (*i.e.,* Clang-CFI cast checker), and has comparable checking coverage with other state-of-the-art tools, which—in contrast—rely on runtime intensive type tracking for checking type confusions for both polymorphic and non-polymorphic objects.

Chapter 8

LLVM-CFI: Analyzing Control Flow Integrity Defenses

In this Chapter, which belongs to the second part of this thesis, we designed a static compiler based tool, called LLVM-CFI, which is usable for assessing static state-of-the-art CFI policies. The intuition behind this decision is motivated twofold. First, due to the fact that it is most likely that memory corruptions can not be fully eliminated we wanted to provide an active defense by hardening the program. Second, we wanted in the next Chapters to come up with novel CFI based techniques for protecting indirect program control flow transfers. Further, in order to effectively address this task, we wanted to learn how effective and which level of security the existing CFI defenses offer. As such, we addressed **RQ5** by providing a framework for assessing static CFI policies w.r.t. calltarget set reduction after a certain CFI defense was applied. Further, we gained important lessons which helped us prepare next design decisions for the tools developed in the next two Chapters. Finally, note that parts of this Chapter has already been published by Muntean *et al.* [198].

8.1 Introduction

What is the overall view? Ever since the first Return Oriented Programing (ROP) attack [242], the cat and mouse game between defenders and attackers has seen several peaks [37]. As defenses improved over time, the attacks progressed with them [37]. While defenders followed several lines of research when building defenses: control flow integrity [286, 44, 212, 218, 78, 287, 261, 24, 106, 210, 268], binary re-randomization [279], information hiding [13], and code pointer integrity [135], the attacks kept up the pace and got more and more sophisticated [238, 137, 143, 21, 59] by deliberately avoiding the primitives (invariants) of the used defenses. In addition, the attackers try to exploit: (1) existing implementation bugs, (2) new weaknesses or (3) side-channels of systems.

What is the problem? In principle, even with the myriad of defenses, performing exploits is still possible. In practice, the situation is non-trivial, as applied CFI policies make reasoning about security harder, since attacks become highly program-dependent. The attacker is confronted with the challenge of searching (manually or automatically) the protected program's binary or source code for gadgets which remain useful after new defenses have been deployed. Thus, there is a growing need for more advanced defense-aware tools, that assist attackers and security analysts searching for gadgets, in order to craft or defend against *Code Reuse Attacks* (CRAs), respectively.

What are the existing solutions? Unfortunately, most static gadget discovery tools [67, 233, 237] perform relatively simple static analysis on the program binary (*i.e.,* location- based analysis on gadget start and end addresses) in order to find useful gadgets. Furthermore, these tools are not aware of the applied defense and lack a deeper understanding of the protected program. As such, they can find gadgets, but they can neither determine if the gadgets are still usable after a defense was deployed, nor can they categorize a found gadget. Other tools perform dynamic analysis on the protected program in order to find usable gadgets. In particular, Newton [267] models the defense dynamically, but relies on the real execution of the target program. As such, the tool is constrained to program-driven (based on given inputs) executions. These tools, however, only cover a minimal part of the program control flow graph and its dependencies. Therefore, the knowledge of usable gadgets that are discoverable during analysis is useful, but limited.

What are the limitations of the solution? To further raise the bar for the attacker with each applied defense, new tools ideally need to: (1) model the defense as precisely as possible, and (2) use precise semantic knowledge about the protected program code (source code and/or machine code) in order to thoroughly search for available gadgets in the protected program. Unfortunately, to the best of our knowledge, the only dynamic analysis tool available, Newton [267] (which is not open source), is only able to find useful gadgets during runtime,with the number of usable found gadgets being rather low. Furthermore, this tool is restricted by the taint dynamic analysis on which it relies. Moreover, all currently available static tools naively search in the binary of the protected program for gadgets without a prior modeling of the defense. They also do not provide any insight to whether a gadget is usable during an attack or whether the callsites for the found gadgets are illegal.

What is our insight? In this Chapter, we present LLVM-CFI, a static source code analysis tool built upon the Clang/LLVM [152] compiler framework for modeling static state-of-the-art control flow integrity (CFI) defenses to empirically compare them to each other. We focus on static CFI defenses, as they are more common then dynamic defenses and comparing them against each other is easier.

For evaluation convenience, we used a source code based implementation, even though some of the original tools were implemented at binary level. Thus, our goal is to evaluate different CFI tools, regardless of whether they were implemented at the source code or binary level. Note that if we would evaluate all existing work on the same system, the use of different compilers would make the results uncomparable. Further, LLVM-CFI can be also used to find source code gadgets that are still available for attack crafting, even after a restrictive CFI defense was deployed. LLVM-CFI relies on the insight that by carefully modeling a CFI defense into a comprehensive policy, the introduced constraints on callsites and calltargets can be assessed during program compile time. Therefore, there is no need to look at the binary of the protected program since its source code provides more semantic richness and precision w.r.t. the constraints imposed by each CFI defense.

We evaluate LLVM-CFI with common open source programs: NodeJS, Bind, Memcached, Httpd, Lighttpd, Nginx, Apache Traffic Server, Google's Chrome, and Redis. We show that LLVM-CFI can help the assessment of CFI defenses and is effective at finding gadgets, even with highly restrictive state-of-the-art CFI defenses deployed. In addition, we demonstrate how LLVM-CFI can be utilized to craft a code reuse attack. We also show how it can be effectively used to empirically measure the real attack surface reduction after a certain static CFI defense policy was used to harden a program's binary.

What are our contributions? In summary, in this Chapter, we make the following contributions:

- We show that static CFI attacker models are powerful and drastically lower the bar for performing CRAs against state-of-the-art defenses.
- We implement LLVM-CFI, a novel framework usable for generating low-effort CRAs and for empirically assessing CFI defenses against each other, using constraint-driven static analysis of aggregated program meta-data obtained during program compilation.
- We evaluate and compare existing static CFI defenses against each other, showing their strengths and weaknesses by employing LLVM-CFI's CFI defense constraints. We show the target sets and allow reasoning about the sizes of the sets and provide useful metrics.
- We present a NodeJS-based case study with the goal of highlighting how LLVM-CFI can be used to craft CRAs against a state-of-the-art defense, *e.g.,* the secure VTV/IFCC [261] implementation.
- We introduce four new CFI defense assessing metrics that are more effective and precise than existing metrics. We also show the significance of one of our metrics by putting it to work in our evaluation. Compared to other tools in this Section, we take into account object oriented program features.

8.2 Threat Model

We align our threat model with previous work by considering a powerful and realistic adversary [267, 187, 251, 59, 238]. We rely on deployed and complementary mitigation techniques to achieve a comprehensive coverage. The attacker knows which defenses are applied, has access to the source code of his prospective victim application and the target binary is not re-randomized in short intervals. For instance, the attacker can extract knowledge about the environment and apply this knowledge to the attack. The program which the attacker wants to attack contains an exploitable memory corruption (*e.g.,* buffer overflow) that allows the attacker to corrupt memory objects (*e.g.,* inserting fake objects). The attacker relies on an automated gadget discovery tool to craft a CRA with reduced application knowledge. Further, the attacker's goal is to find gadgets and mount a CRA even if state-of-the-art CFI [261, 286] defenses are deployed. We deliberately only focus on CFI-based defenses as this defense class is well explored, evaluated and used in practice [149]. The system on which the vulnerable application is running ensures that the memory can be either writable or executable (W$\oplus$X), but not both at the same time (*e.g.,* DEP [164]). We assume that Address Space Layout Randomization (ASLR) [219] is in place. A perfect (one to one mapping between callee and caller) shadow stack implementation is also deployed, which makes the exploit of CFG returning edges impossible. The attacker can therefore only jump at the beginning of a function and not inside due to a coarse-grained forward-edge CFI [290] defense. The system on which the vulnerable application runs enforces execute-only memory pages. This essentially means that the CPU can fetch instructions, while normal read and write accesses trigger an access violation (*e.g.,* XOM [25]).

8.3 Overview

LLVM-CFI is a static gadget detection framework which can be used to assist an analyst evaluating the attack surface of different types of static CFI defenses. To achieve this, LLVM-CFI applies a static black box strategy in order to statically retrieve code-reuse gadgets through a set of attacker-controllable forward control flow graph (CFG) edges. The forward-vulnerable CFG edges are expressed as a callsite with a variable number of possible target functions. Further, these edges can be reused by an attacker to call arbitrary functions via arbitrary read or write primitives. To call such series of arbitrary functions, an attacker can chain a number of edges together by dispatching fake objects contained in a vector. See, for example, the COOP [238] attack which is based on a dispatcher gadget used to call other gadgets through a single loop iteration. The COOP attack uses gadgets which are represented by whole virtual functions.

LLVM-CFI supports a wide range of code reuse defenses based on user-defined policies, which are composed of constraints about the set of possible calltargets allowed by a particular applied CFI defense. The main idea behind LLVM-CFI is to compile the target program with different types of CFI policies and get the allowed target set per callsite for each constraint configuration. Note that we assume the program was compiled with the same compiler as the one on which LLVM-CFI is based. Moreover, LLVM-CFI's policies are reusable and extensible; they model security invariants of important CRA defenses. Essentially, under these constraints, virtual pointers at callsites can be corrupted to call any function in the program. Thus, in this thesis, we focus on the possibility to bend [36] a pointer to the callsite's legitimate targets. Further, we assume that large programs contain enough gadgets for successfully performing CRAs. Bending assumes that it is possible for an attacker to reuse protected gadgets during an attack making the applied defense of questionable benefit.

Figure 8.1 depicts the main components of LLVM-CFI and the work-flow used to analyze the source code of a potentially vulnerable program in order to determine CRA statistics, as follows. The analyst first provides as input ❶ to LLVM-CFI a program's source code ❷. Based on the

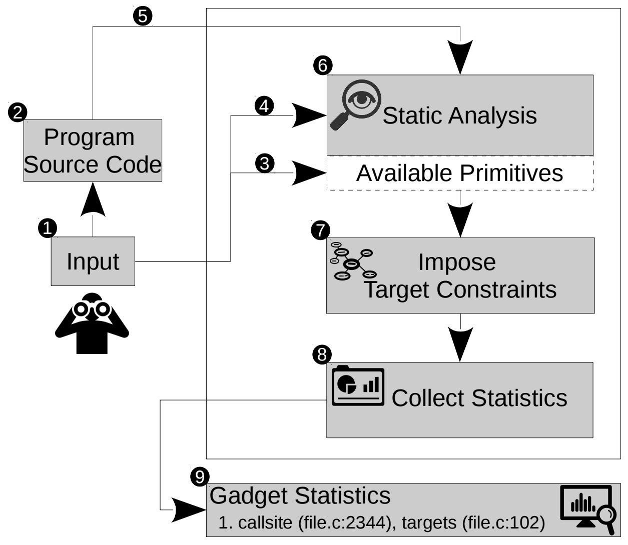


Figure 8.1: Design of LLVM-CFI with the main stages of the analysis pipeline.

previously selected defense, the desired analysis will be performed. The analysis previously mentioned is dependent on the selected defense model and on the available primitives. The analyst does this if he knows that the defense he wants to apply is currently available in the LLVM-CFI tool. The user selects the used defense model from the list of implemented defenses. This is done by switching on a flag inside the LLVM-CFI source code, which can also be implemented as a compiler flag, if desired. In case the defense is not available, the analyst needs to extend the list of primitives ❸, and model his defense as a policy (set of constraints) in the analysis component of LLVM-CFI❹. In order to do this, he needs to know about the analysis internals of LLVM-CFI. After selecting/modeling a defense, the analyst forwards the application's source code ❺ to LLVM-CFI which will analyze it with its static analysis component. During static analysis ❻ the previously selected defense will be applied when compiling the program source code. As the analysis is performed, each callsite is constrained with only the legitimate calltargets. Note that the per callsite, a legitimate calltarget is dependent on the currently selected defense model. The result of the analysis contains information about the residual target set for each individual callsite after a CFI policy was assessed ❼. This list contains a set of gadgets (callsites + calltargets) that can, given a certain defense model, be used to bend the control flow of the application. These target constraints are collected and clustered in the statistics collection component of LLVM-CFI❽. Finally, at the end of the gadget collection phase, a list of calltargets containing potential usable gadgets statistics ❾ based on the currently applied defense(s) will be reported. Note that if needed, the set of available primitives can be extended beyond the extensive set provided by LLVM-CFI.

8.3.1 Available Analysis Primitives

LLVM-CFI provides the following program primitives, which are either collected or constructed during the program analysis phase. These primitives are used by LLVM-CFI to implement static CFI policies and to perform target constraint analysis. Briefly, the currently available primitives are as follows:

Virtual table hierarchy. (see [106] for a more detailed definition) Allows performing virtual table inheritance analysis of only virtual classes as only these have virtual tables. Note that a class is virtual if it defines or inherits at least one virtual function. A function in C++ is virtual if it has the C++ virtual modified.

Vtable set. Is a set of vtables corresponding to a single program class. This set is useful to derive the legitimate set of calltargets for a particular callsite. The set of calltargets is determined by using the class inheritance relations contained inside a program.

Class hierarchy. (see [263, 232] for more details) Can be represented as a CFG. Its main purpose is to model inheritance relations between classes. Note that a program can have multiple class hierarchies (*e.g.,* Google Chrome). The difference between virtual table hierarchy and class hierarchy is that the class hierarchy contains both virtual and non-virtual classes whereas the virtual table hierarchy can only be used to reason about the inheritance relations between virtual classes.

Virtual table entries. Allow LLVM-CFI to analyze the number of entries in each virtual table with the possibility to differentiate between virtual function entries, offsets in vtables, and thunks.

Vtable type. Is determined by the name of the vtable root for a given vtable. A vtable root is the last derived vtable contained in the vtable hierarchy.

Callsites. Are used by LLVM-CFI to distinguish between direct and indirect (object-based dispatch and function-pointer based indirect transfers) callsites.

Indirect callsites. Are based on: (1) object dispatches or (2) function pointer based calls. Based on these primitives, LLVM-CFI can establish different types of relations between callsites and calltargets (*i.e.,* virtual functions). At the same time, we note that it is possible to derive backwards relationships from calltargets to legitimate callsites based on this primitive.

Callsite function types. Allow to precisely determine the number and the type of the provided parameter by a callsite. As such, a precise mapping between callsites and calltargets is possible.

Function types. Allow to precisely determine the number of parameters, their primitive types and return type value for a given function. This way, LLVM-CFI can generate a precise mapping between compatible calltargets and callsites.

These primitives can be used as building blocks during the various analyses that LLVM-CFI can perform in order to derive precise measurements and a thorough assessment of a modeled static CFI policy. We note that in order to model other CFI defenses, other (currently not available) simple or aggregated analysis primitives may need to be added into LLVM-CFI's analysis phase.

8.3.2 Constraints

The basic concept of any CRA is to divert the program intended control flow by using arbitrary memory write and read primitives. As such, the result of such a corruption is to bend [36] (modify) the control flow, such that it no longer points in the intended (legitimate) calltarget set but rather call calltargets which would not be called during benning program execution. This means that the attacker can point/call to any memory address in the program (shared libraries included). While this type of attack is still possible, we want to highlight another type of CRA in which the attacker uses the intended/legitimate per callsite target set. That is, the attacker calls inside this set and performs his malicious behavior by reusing calltargets which are protected, yet usable during an attack. As previously observed by Veen *et al.* [267], CRA defenses try to mitigate this by mainly relying on one or two dimensions at a time, as follows:

Write constraints. limit the attacker's capabilities to corrupt writable memory. If there is no defense in place, the attacker can essentially corrupt: pointers to data, non-pointer values such as strings, and pointers to code (*i.e.,* function pointers). In this thesis, we do not investigate these types of defenses as these were already addressed in detail by Veen *et al.* [267]. Instead, we focus on target constraints as these represent a big class of defenses which in our assessment needs separate and detailed treatment. This obviously does not mean that our analysis results cannot be used in conjunction with dynamic write constraint assessing tools. Rather, our results represent a common ground truth on which runtime assessing tools can build their gadget detection analysis.

Target constraints. restrict the legitimate calltarget set for a callsite which can be controlled by an attacker. With no target constraints in place, the target set for each callsite is represented by all functions located in the program and any linked shared library. The key idea is to reduce the wiggle room for the attacker such that he cannot target random callsites. As most of these defenses impose a one-to-*N* mapping, an attacker being aware of said mapping could corrupt the pointer at the callsite to bend [36] the control flow to legitimate targets in an illegitate order to achieve his malicious goals. This essentially means that all static defenses impose target constraints.

Static Analysis. LLVM-CFI is based on the static analysis of the program which is represented in LLVM's intermediate representation (IR). The analysis is performed during link time optimization (LTO) inside the LLVM [152] compiler framework to detect callsites and legitimate callees under the currently analyzed CFI defense. LLVM-CFI uses the currently available primitives and the implemented defenses to impose target constraints for each callsite individually. Currently, LLVM-CFI supports eight target constraint defenses, but this list can easily be extended since other defenses are based on similar mechanisms which can be performed during a whole program analysis.

Generic Target Constraints. As mentioned above, LLVM-CFI can be used to impose existing generic calltarget constraints (defenses) based on class hierarchy relations and callsites and calltarget type matching with different levels of precision depending on the currently modeled CFI defense. Further, LLVM-CFI allows extending and combining existing policies or applying them concurrently.

8.3.3 Generating Defense Statistics

During the static analysis LLVM-CFI determines the legitimate calltarget set for each callsite. These results are used as raw data for computing more complex statistics such as the average, minimum and maximum count of legitimate calltargets per callsite. LLVM-CFI outputs all collected statistics about the legitimate caller/callee pairs such that these can be later used by an analyst. The results can, for example, be used as parameters for different metrics for computing legitimate callee return sets. Mapping calltargets to previously discovered (manually/automatically) gadgets is also possible. Moreover, other relevant statistics can be built for each callsite w.r.t. how many legitimate calltargets actually contained gadgets which are relevant to a specific type of code reuse attack, *e.g.,* the COOP [238] code reuse attack where a complete virtual function (calltarget) represents a single gadget. Thus, an analyst can further infer which calltargets should be protected by a defense and which are safe to remain unprotected, in case he needs to strike a trade-off between protection (security) and performance which is most likely always the case in reality.

8.4 Mapping Defenses

In this Section, we show how to map CFI defense constraints into LLVM-CFI. Similarly to Newton [267], which was developed in parallel to LLVM-CFI, we model the security provided by CRA defenses along two axes: (1) write constraints imposed by the defense, and (2) imposed target constraints. The main motivation for the usage of this modeling technique is that: (1) the majority of CFI techniques do not impose write constraints, with the exception of CFI runtime tools which adjust their analysis by using hardware features, and (2) it serves as a natural extension (*i.e.*, it eases understanding) of existing work. Next, we will map the defenses according to these constraints. In our work, we focus only on one class of defense at a time and we add more constraint types to the target constraint axis. As opposed to Newton [267], LLVM-CFI helps to derive static constraints imposed on the target program. This mapping allows to define textual descriptions of each CFI policy and reduces the task to a compiler-based counting problem allowing an analyst to determine which callsites and calltargets are protected and which are not.

8.4.1 Deriving Constraints

Table 8.1 highlights the constraints imposed by each defense subclass. In our work, we do not consider runtime-based write constraining defenses (HCFI and CsCFI, see Newton [267] for more details), since these runtime constraints are hard to be assessed statically. Instead, LLVM-CFI models a comprehensive set of eight defense classes with no write constraints, which we discuss in detail in subsubsection 8.4.1.1.

8.4.1.1 Constraint Subclasses

Defense			Write Constraints	Target Constraints	
Class	*Subclass*	*Solutions*	*Details*	*Dynamic*	*Details*
	TypeArmor	[268]	None	✘	Bin types
	IFCC/MCFI	[210, 261]	None	✘	Src types
	Safe IFCC/MCFI	[210, 261]	None	✘	Safe src Types
	ShrinkWrap/IVT	[106]	None	✘	Precise sub-hierar.
CFI	VTV	[261, 24]	None	✘	Sub-hierarchy
	vTint	[287]	None	✘	All vtables
	Marx/VCI	[218, 78]	None	✘	vtable paths
	HCFI	[212]	None	✓	Computed
	CsCFI	[44]	Segr	✓	None
	vTrust	[286]	None	✘	Precise src types

Table 8.1: Mapping of CFI code-reuse defenses into LLVM-CFI constraints.

TypeArmor. Imposes callsite target constraints based on a function type policy. In particular, for each callsite only targets which fulfill the parameter count based policy are allowed during runtime.

IFCC/MCFI. Enforces similar constraints as TypeArmor, with the exception that the function type is computed at the source rather than at the binary level.

Safe IFCC/MCFI. Contains the same defenses as the IFCC/MCFI defense subclass, except that in this case, we distinguish a *safe* mode, where type information is less strict for compatibility reasons with real-world programs, this is discussed thoroughly by Tice *et al.* [261] VTV/IFCC.

VTV. Subclass enforces for each indirect callsite the whole subclass hierarchy *Src sub-hierarchy* which is precise but leaves wiggle room for the attacker and it enforces too many calltargets, as noted by Haller *et al.* [106].

ShrinkWrap/IVT. Subclass can enforce a more precise class sub-hierarchy (*Precise src types*) than IFCC/MCFI. For each indirect callsite (object dispatch) a precise class hierarchy is computed.

vTint. Operates at the binary level in order to find indirect callsites and virtual tables. This defense subclass is enforcing for each callsite all entries contained in all detected virtual tables (*all vtables*).

Marx/VCI. Subclass operates on binary to recuperate a quasi class hierarchy: (1) the class has no root node, and (2) the edges in the class hierarchy are not oriented.

vTrust. Subclass enforces matching function types for each indirect callsite. It follows a precise source code callsite-to-targets mapping (*Precise src types*) based on computing at all calltargets a hash composed of the number, types, and return type of each virtual function.

Next, we highlight the differences between the VTrust CFI policy and the class hierarchy analysis (CHA) based policy of LLVM-CFI/Shrinkwrap and IVT in order to make clear how the calltarget sets differ depending on the used defense policy. ShrinkWrap and IVT use CHA and based on the base type of the object, only some subtypes are allowed, while VTrust falls back to the base objects and is therefore less precise than CHA-based policies. IVT, for example, allows all subtypes. Note that with a larger class hierarchy also as many targets are allowed by vTrust and IVT, whereas ShrinkWrap allows just a sub-part of the class sub-hierarchy (some subtypes).

```
0.class Foo { virtual void bar(); };
1.class Bar:public Foo { virtual void bar(); }
2.class Baz:public Bar { virtual void bar(); }
3.class Bac:public Bar { virtual void bar(); }
4.Bar *b = new Bar();
5.b->bar();
```

Figure 8.2: C++ based class hierarchy with four classes.

Figure 8.2 depicts a simple C++ class hierarchy which will be used to show that based on the used CFI policy different functions are accessible after applying a certain policy. Under vTrust the indirect call, line five in Figure 8.2, can target any of the four functions, and under CHA based policy (as implemented by IVT) `Bar::bar`, `Bac::bar` and `Baz::bar` can be called, while under a different version of CHA based policy (as implemented by LLVM-CFI and Shrinkwrap) only `Bar::bar` and `Baz::bar` can be called. This highlights the fact that the CFI policies have different granularity w.r.t. constraining the calltarget set per callsite.

Note that the first three constraint classes were first presented by Newton [267], while the last five classes are presented for the first time in this thesis.

8.4.1.2 Out of Scope Write Constraints

Corrupting data pointers, *Corrupting non-pointers*, and *Corrupting segregated state* write constraints are out of scope in this work, *i.e.*, all assessed classes in this thesis do not impose any writing constraints. We advise the reader to look at Newton [267] for more details about these write constraints. Note that these constraints are not relevant for our CFI subclasses as they are not dependent on them.

8.4.2 Mapping Defenses

Figure 8.3 depicts the constraints of the defenses, presented in Table 8.1. The constraints are grouped in two categories in ascending order indicated with two arrows on the X-axis according to their precision of determining precise target sets per callsite. The X-axis shows the write constraints imposed by each defense subclass, and the Y-axis shows the target constraints. Defenses that share both the same write and target constraints impose equivalent security restrictions, thus each (X-axis, Y-axis) pair depicted in Figure 8.3 represents an equivalence class.

Note that in this thesis, we are interested in only static CFI defenses which do not impose any write constraints. We are aware that these defenses are thought to be accompanied by some type of write constraint but for the sake of simplicity, we abstract these away for now. As such, run-time CFI (black rectangle), re-randomization (yellow circle), pointer integrity (brown diamond), and information hiding (green triangle) depicted in Figure 8.3 are not taken into consideration

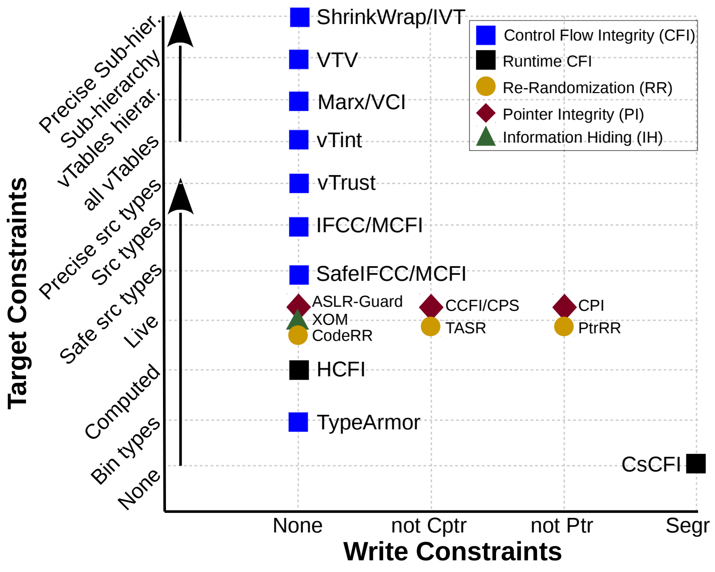


Figure 8.3: Mapping of CFI defense classes to write (X-axis) and target (Y-axis) constraints in LLVM-CFI.

as these were thoroughly analyzed by Newton [267]. Our objective is to focus on the details of the static CFI class in order to reveal novel insights of this widely used class of defense.

LLVM-CFI can compile a given program with DWARF information. In this situation, the function name and location inside the program is back-traceable to the exact location in the source file. The same level of detail is possible for the associated callsite.

Important to observe is that all static CFI defenses impose no write constraints and as such it is recommendable to use these defenses with some type of write constraints which for example would impose that the virtual pointer used during object dispatches cannot be overwritten to point to an illegitimate address inside the program. Further, the value of this pointer is determined during runtime. We further assume that there are no implementation specific vulnerabilities of these defenses and as such the equivalence classes hold. Further, our constraint-based classification abstracts away specific implementation details and ignores implementation specific differences across these defenses. For example, we consider the used primitives (*e.g.,* class hierarchy) for enforcing the CFI policy to be ideal with no precision differences between binary and source code based tools. As such, we are able to generate lower bounds w.r.t. the legitimate calltarget set of these defenses. Further, our approach is applicable to general gadget generation constraints across many different defenses.

Next, we show how to implement constraints for the mapped defenses in LLVM-CFI, by using the primitives specified in Section 8.3.1. Specifically, we present in detail the constraints imposed at any indirect callsite depending on the target constraint.

Corrupting code pointers. The CFI defenses which do not impose any write constraints are allowing any memory to be corrupted, such as code pointers. This is presented in Figure 8.3 by abstracting the CFI constraints to a counting problem of legitimate calltargets per callsite. These defenses are located on the left hand side of Figure 8.3 depicted on the X axis with (*None*).

Note that in general, these type of constraints assume that the callsite is not corruptible since it is located in read-only memory, but since these constraints do not impose any specific code pointer corrupting defense these are regarded in this thesis as not defending code pointer at all.

The particular constraint-based counts are obtained by computing for each constraint the target set counts with our LLVM-CFI tool. Next, LLVM-CFI compiles any given program in order to obtain the target set and performs counts according to the currently used constraint. Note that while in Section 8.4.1.1, we provide the general description of how the constraint classes work, in this Section, we describe how the target constraints work in detail by associating them to the LLVM-CFI primitives.

In the following, we will present eight defense classes modeled inside LLVM-CFI. Note that each CFI defense description is very close to how the original CFI defense policy was implemented in each tool and was thoroughly discussed with the original authors.

Bin Types (TypeArmor). For each indirect callsite (object dispatch and indirect pointer call) (1) count the total number of virtual table entries residing in the whole virtual table hierarchies, and (2) count the total number of non-virtual functions residing in the program, which are consuming at most as many parameters as provided by the call site and up to six parameters. If a program contains multiple distinct virtual table hierarchies (islands) then continue to count them too and take them also into consideration for a particular callsite. Note, an island is a program class hierarchy which is not connected to any other program's class hierarchy.

Safe src types (Safe IFCC). For each indirect callsite count the number of functions (virtual and non-virtual) located in the whole program for which the number and type of parameters required matches the number and type of arguments provided at the callsite. The return type of the matching function is ignored. All types of parameter pointer types are considered interchangeable, *i.e.,* **int*** and **void*** pointers are considered interchangeable.

Src types (IFCC). For each indirect callsite count the number of functions (virtual and non-virtual) located in the whole program for which the number and type of parameters required matches the number and type of arguments provided at the callsite. The return type of the matching function is ignored. Compared to *Safe src types* this policy distinguishes between different pointer types. Neither the return value of the matching function nor the name of the function are taken into consideration.

Precise src types (vTrust). For each indirect callsite compute the number of parameters, their types, and the name of the function (the literal name used in `C/C++` without any class information attached) that is called at that location. Match this function type identifier with all virtual functions contained in all virtual table hierarchies. The name of the function is taken into consideration when building the hash but not the function return type as this can be polymorphic. A match is given when the signature of a callsite matches with one of the signatures of one of the virtual functions.

All vtables (vTint). For each indirect callsite, count the total number of virtual functions pointed to by vtable entries in any vtable present in the program.

vtable hierarchy/island (Marx). For each indirect callsite count the total number of virtual functions pointed to by vtables entries having the same index in the vtable as the index used at the callsite. Do this for certain vtables where the index matches with the index used at the callsite and which are located in the class hierarchy which contains the class of the dispatched object. The abstract classes are not taken in consideration, this can be recognized by vtables having pure virtual function entries.

Sub-hierarchy (VTV). For each virtual callsite build the class sub-hierarchy having as root node the base class (least derived class that the dispatched object can be of) of the dispatched object. From the classes in the sub-hierarchy take all vtables. In these vtables find the function entries located at the offset used by the virtual dispatch mechanism at this particular callsite. Then count the number of virtual functions to which these entries point to.

Precise Sub-hierarchy (ShrinkWrap). For each virtual callsite identify the vtable type used. Take the vtable of said type from the base class of the dispatched object and build the vtable sub-hierarchy having this vtable as root node. From the vtables in this hierarchy find the function entries located at the offset used by the virtual dispatch mechanism at this callsite. Then count the number of virtual functions, that these entries point to.

After LLVM-CFI computes for each callsite the calltarget set, as above described, it will sum up all results for each applied constraint and for each callsite to generate several statistics.

8.5 Assessing CFI Policies

In this Section, we present how LLVM-CFI can be used to perform different types of CFI-related measurements. LLVM-CFI helps to provide precise and reproducible measurement results when performing CFI-related investigations. This Section provides several alternatives to existing limited CFI metrics: AIR [290], fAIR [261], and AIA [90], which provide average result values w.r.t. only forward edges calltarget set reduction per callsite after a certain CFI policy was applied to a program. Further, we will show with which novel CFI-related policies LLVM-CFI can be used in conjunction. Another set of metrics was introduced in a recent survey by Burow *et al.* [32]; we evaluate our metrics on real-world programs while in the survey the authors evaluate their metrics on a per-mechanism basis, changing the point of view. Finally, for space reasons we focus on the most widely used ones.

Figure 8.4 depicts the dependencies between our metrics and program primitives. In Figure 8.4, we use the following abbreviations:(1) *ics:* indirect call site (*i.e.,* x86 `call` instruction); (2) *irs:* indirect return site (*i.e.,* x86 `ret` instruction); (3) *P:* program; (4) *VT:* virtual table; (5) *VTI:* virtual table inheritance; (6) *CH:* class hierarchy; (7) *CFG:* control flow graph; (8) *CG:* code reuse gadget; Metrics: (9) *CTR:* indirect call target reduction; (10) *CSD:* indirect call site damping metric; (11) *RTR:* indirect return target reduction; (12) *RSD:* indirect return site damping metric; (13) *fCGA:* forward-edge based *CG* availability; (14) *bCGA:* backward return-edge based *CG* availability.

Based on the two observations: (1) LLVM-CFI allows to determine precise forward edge mappings as well as backward edge mappings when the forward edge mapping is available due to the caller callee function calling convention, and (2) LLVM-CFI can be used to search for the presence of code reuse gadgets (*i.e.,* COOP gadgets which are complete virtual functions).

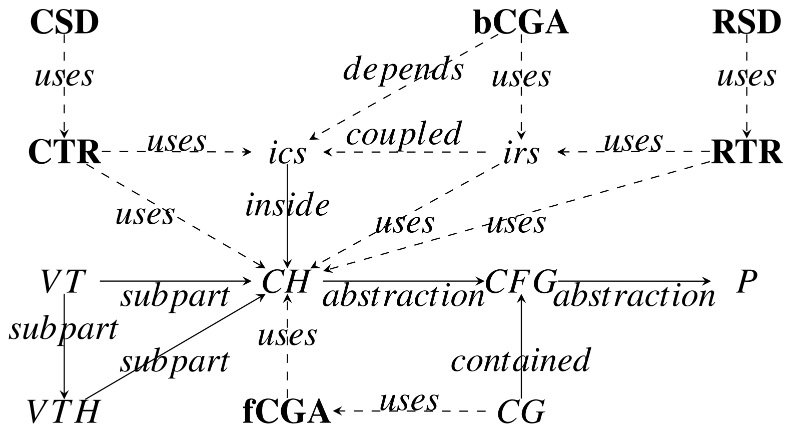


Figure 8.4: Dependencies between our metrics (bold text), and program metadata primitives.

Next, we formulate four advantages of our approach: (1) our metrics do not provide average numbers but rather absolute values, (2) can be used to assess backward edge target set reduction, (3) can be used to assess the forward and backward edge control flow transfer damping due to a variable number of checks inserted before each indirect control flow transfer, and (4) can be used to assess the forward and backward edge control flow transfers w.r.t. gadget availability. Finally, we present our four CFI metrics which can be used in conjunction with LLVM-CFI.

Definition 1 (CTR). Let ics_i be a particular indirect callsite in a program P, ctr_i is the total number of legitimate calltargets for an ics_i after hardening a program with a certain CFI policy.

Then the CTR metric is:

$$CTR = \sum_{i=1}^{n} ctr_i \tag{8.1}$$

Note that the lower the value of CTR is for a given program, the more precise the CFI policy is. The optimal value of this metric is equal to the total number of callsites present in the hardened program. This means that there is a one-to-one mapping. We can also capture the distribution of the numbers of call targets using min, max, and standard deviation functions.

Minimum: $\min_i\{ctr_i\}$; Maximum: $\max_i\{ctr_i\}$; and Standard Deviation (SD):

$$CTR_{SD} = \sqrt{\frac{\sum_{i=1}^{n}(ctr_i - \overline{ctr_i})^2}{n}} \tag{8.2}$$

Definition 2 (RTR). Let irs_i be a particular indirect return site in the program P, rtr_i is the total number of available return targets for each irs_i after hardening the backward edge of a program with a CFI policy.

Then the RTR metric is:

$$RTR = \sum_{i=1}^{n} rtr_i \tag{8.3}$$

Note that as lower the value of RTR is for a given program, the better the CFI policy is. The optimal value of this metric is equal to the total number of indirect return sites present in the hardened program. This means that there is a one-to-one mapping. Other key properties: Minimum: $RTR_{MIN} = \min_i\{rtr_i\}$; Maximum: $RTR_{MAX} = \max_i\{rtr_i\}$; and Standard Deviation (SD):

$$RTR_{SD} = \sqrt{\frac{\sum_{i=1}^{n}(rtr_i - \overline{rtr_i})^2}{n}} \tag{8.4}$$

Definition 3 (CSD). Let ncn_i be the runtime cost for a CFI check located before an indirect callsite ics_i which is present in a given hardened program. Let k be the runtime overhead (damping ratio) imposed by one inserted CFI check at the irs_i. The k value could represent milliseconds needed to perform a CFI check.

Then the CSD metric is:

$$CSD = \sum_{i=1}^{n} ncn_i * k \tag{8.5}$$

Note that as lower the value of CSD is for a given program, the better the CFI policy is. The optimal value of this metric is equal to the total number of indirect callsites present in the hardened program when k equals one. The value of k can take any millisecond value.

Definition 4 (RSD). Let nrn_i be the runtime cost for a CFI check performed before an indirect return site irs_i which is present in a given hardened program. Let k be the runtime overhead (damping ratio) imposed by one inserted CFI check at the irs_i. The k value could represent milliseconds needed to perform a CFI check.

Then the RSD metric is:

$$RSD = \sum_{i=1}^{n} nrn_i * k \tag{8.6}$$

Note that the lower the value of RSD is for a given program, the better the CFI policy is. The optimal value of this metric is equal to the total number of indirect callsites present in the hardened program when k equals one. The value of k can take any millisecond value.

Definition 5 (fCGA). Let cgf_i be the total number of legitimate call targets that are allowed and which contain gadgets according to a gadget finding tool.

Then the forward code reuse gadget availability $fCGA$ metric is:

$$fCGA = \sum_{i=1}^{n} cgf_i \tag{8.7}$$

Note that the lower the value of $fCGA$ is, the better the policy is. This means that every time a calltarget containing a code reuse gadget is protected by a CFI check, this gadget is not reachable. The reverse is true when the calltarget return contains a gadget and there are indirect control flow transfers which can call this indirect return site unconstrained.

Definition 6 (**bCGA**). Let cgr_i be the total number of legitimate callee returns addresses which contain code gadgets according to a gadget finding tool.

Then the backward code reuse gadget availability $bCGA$ metric is:

$$bCGA = \sum_{i=1}^{n} cgr_i \tag{8.8}$$

Note that the lower the value of $bCGA$ is, the better the policy is. This means that every time when a calltarget return site which contains a code reuse gadget is protected by a CFI check then this gadget is not reachable. The reverse is true, when the calltarget return contains a gadget and there are indirect control flow transfers which can call this indirect return site unconstrained.

Above, we depicted these metrics to point out that all these CFI-related measurements are relevant and could be performed depending on the type of CFI policy a certain tool implements. The eight assessed CFI policies can neither constrain the backward edge nor analyze their availability of gadgets and their runtime cost.

The metrics presented above, can be used by LLVM-CFI to assess CFI policies w.r.t. other dimensions (*i.e.,* (1) backward edges, and (2) gadget availability) with which most of the CFI techniques are not addressing. Thus, we did not use the four metrics in conjunction with the eight CFI classes assessed in this thesis since these metrics to not address (1) and (2). Moreover, we are not aware of any CFI policy which is based on these types of four metrics. Finally, by using these metrics experiments become more reproducible and the tools better comparable against each other.

8.6 Design and Implementation

In the following Section, we present LLVM-CFI's design and implementation details.

8.6.1 Data Collection and Aggregation

Collection. LLVM-CFI collects the virtual tables of a program in the Clang front-end and pushes them through the compilation pipeline in order to make them available during link-time optimization (LTO). For each virtual table, LLVM-CFI collects the number of entries. The virtual tables are analyzed and aggregated to virtual table hierarchies in a later step. Other data such as direct/indirect callsites and function signatures are collected during LTO.

Aggregation. Next we present the program primitives which are constructed by LLVM-CFI: (1) virtual table hierarchies based on the previously collected virtual tables inside the Clang front-end. The virtual table hierarchies are used to derive relationships between the classes inside a program (class hierarchies), determine sub-hierarchy relationships and count, for example, how many virtual table entries (virtual functions) a certain virtual table sub-hierarchy has. (2) virtual table sets which are used for mapping callsites to legitimate class hierarchy-based virtual calltargets. (3) callsite function types which are composed of the number of parameters provided by a callsite, their types, and if the callsite is a void or non-void callsite. (4) function types which are composed of the function name, the expected number of parameters and their types and if the function is a void or non-void function.

8.6.2 CFI Defense Modeling

LLVM-CFI implements a set of constraints for each modeled CFI-defense, which are defined as analysis conditions that model the behavior of each analyzed CFI-defense. These constraints are particular for each CFI-defense and operate on different primitives. More specifically, different constraints of a CFI-defense are implemented inside LLVM-CFI. The steps for modeling a CFI defense are as follows: (1) Which LLVM-CFI's primitives are used by the policy? (2) Is there a nesting or subset relation between primitives? (3) Does the policy rely on hierarchical meta-data primitives? (4) What is the callsite/calltarget matching criteria? (5) How to count a callsite/calltarget match?

Note that there is no effort needed to port LLVM-CFI from one policy to another as all policies can operate in parallel during compile time. As such, the measurement results obtained for each policy are written in one pass in an external file for later analysis.

Next, we give a concrete example of how a CFI defense, TypeArmor's *Bin types* policy [268], was modeled inside LLVM-CFI by following the steps mentioned above. For more details, see Section 8.4.2 in Appendix for a description on how this policy works and the constraint sub-classes depicted in Section 8.4.1.1.

(1) The policy uses the: callsite primitive, indirect callsite, callsite function type, and function type primitives provided by LLVM-CFI. (2) Yes, from all functions contained in the program, we analyze only the virtual functions which expect up to six parameters to be passed by the callsite. Next, from all callsites we filter out the ones which are not calling virtual functions and which provide more than six parameters to the calltarget. Check if the callsite is a void or non-void callsite. Check if each analyzed calltarget is a void or a non-void target. (3) The policy does not rely on hierarchical meta-data. (4) A callsite matches a calltarget if it provides less or the same number of parameters as the calltarget expects. (5) In case the matching criteria holds, we count the total count one up for each found match.

Finally, these constraints are implemented as an LLVM compiler module pass performed during LTO. Note that even an analyst with restricted previous knowledge can model the constraints of a CFI policy by observing how other existing policies were implemented inside LLVM-CFI.

8.6.3 CFI Defense Analysis

LLVM-CFI performs for each implemented CFI defense a different analysis. Each defense analysis consists of one or more iterations through the program primitives which are relevant for the CFI defense currently being analyzed. Depending on the particularities of a defense, LLVM-CFI uses different previously collected program primitives. More specifically, class hierarchies, class sub-hierarchies, or function signatures located in the whole program or in certain class sub-hierarchy are individually analyzed. During a CFI-defense analysis, statistics are collected w.r.t. the number of allowed calltargets per callsite taking into account the previously modeled CFI-defense.

For example, for a certain CFI defense (*e.g.,* TypeArmor's CFI policy **Bin types**) it is required to determine a match between the number of provided parameters (up to six parameters) of each indirect callsite and all virtual functions present in the program (object inheritance is not taken into account) which could be the target (may consume up to six parameters) of such a callsite. In order to analyze this CFI defense and collect the statistics, LLVM-CFI visits all indirect callsites it previously detected in the program and all virtual functions located in all previously recuperated class hierarchies. Afterwards, each callsite is matched with potential calltargets (virtual functions). Finally, after all virtual callsites and virtual functions were visited (after all class hierarchies were traversed), the generated information is presented to the analyst by printing it in the console and external files.

8.6.4 Implementation

We implemented LLVM-CFI as one link time optimization (LTO) pass by extending the Clang/LLVM (v.3.7.0) compiler [50] framework infrastructure. The implementation of LLVM-CFI is split between the Clang compiler front-end (part of the metadata is collected here), and one link-time pass, totaling around 2 KLOC. LLVM-CFI supports separate compilation by relying on the LTO mechanism built in LLVM [50]. By using Clang, LLVM-CFI collects front-end virtual tables and makes them available during LTO. Next, virtual table hierarchies are built which are used to model different CFI defenses. Other LLVM-CFI primitives such as function types are constructed during LTO. Finally, each of the analyzed CFI defense is separately modeled inside LLVM-CFI by using the previously collected primitives and aggregated data to impose the required defense constraints.

8.7 Evaluation

In this Section, we show how useful LLVM-CFI is by putting it to work and addressing the following research questions (RQs):

- **RQ1:** What is the residual attack surface of NodeJS (we performed an use case) after several CFI policies are independently applied? (Section 8.7.1)
- **RQ2:** What are the generalized results for all analyzed programs? (Section 8.7.2)
- **RQ3:** How can LLVM-CFI be used to rank CFI policies based on the level of protection they offer? (Section 8.7.3)

- **RQ4:** How can LLVM-CFI be used to construct code reuse attacks? (Section 8.7.4)

Test Programs. We evaluated LLVM-CFI by using the following real-world programs: Nginx [204] (v. 1.13.7, `C` code), NodeJS [213] (v. 8.9.1, `C/C++` code), Lighttpd [144] (v. 1.4.48, `C` code), Httpd [7] (v. 2.4.29, `C` code), Redis [231] (v. 4.0.2, `C` code), Memcached [163] (v. 1.5.3, `C/C++` code), Apache Traffic Server [8] (v. 2.4.29, `C/C++` code), and Google Chrome [97] (v. 33.01750.112, `C/C++` code).

Experimental Setup. The experiments were performed on an Intel i5-3470 CPU with 8GB of RAM running on the Linux Mint 18.3 OS. All experiments were performed ten times to provide reliable values. If not otherwise stated, we modeled each of the eight CFI defenses inside LLVM-CFI according to the policy descriptions provided in Section 8.4 and used Definition 1 to compute the residual attack surface after a CFI policy was applied.

8.7.1 Detailed Analysis of NodeJS

In this Section, we analyze the residual attack surface after each of the eight CFI policies was applied individually to NodeJS. We selected NodeJS as this is a very popular real-world application and it contains both `C` and `C++` code. As such, LLVM-CFI can collect results for the `C` and `C++` related CFI polices.

	Targets Median					**Targets Distribution**					
						NodeJs			**MKSnapshot**		
Static target constraint	*NodeJS*	*MKSnaphot*	*Total*	*Min*	*Max*	*Min*	*90p*	*Max*	*Min*	*90p*	*Max*
Bin types	21,950 (21,950)	15,817 (15,817)	15,817 (20,253)	15,817 (15,817)	21,950 (21,950)	12,545 (885)	30,179 (30,179)	32,478 (32,478)	8,714 (244)	21,785 (21,785)	23,376 (23,376)
Safe src types	2,885 (88)	2,273 (495)	2,273 (139)	2,273 (88)	2,885 (21,950)	0 (0)	5,751 (5,751)	5,751 (5,751)	1 (0)	4,436 (4,436)	4,436 (4,436)
Src types	1,511 (56)	1,232 (355)	1,232 (139)	1,232 (56)	1,511 (355)	0 (0)	5,751 (5,751)	5,751 (5,751)	1 (0)	4,436 (4,436)	4,436 (4,436)
Precise src types	3	2	3	2	3	0	499	730	0	507	756
All vtables	6,128	2,903	6,128	2,903	6,128	6,128	6,128	6,128	2,903	2,903	2,903
vtable hierarchy	2	1	2	1	2	0	54	243	0	16	108
Sub-hierarchy	2	1	1	1	2	0	7	243	0	11	108
Precise sub-hierarchy	2	1	1	1	2	0	6	243	0	9	108

Table 8.2: Policies evaluation for NodeJS. The values not contained in round brackets are obtained for only virtual callsites and all targets (*i.e.,* virtual and non-virtual), while the values in round brackets are obtained for all indirect callsites (*i.e.,* virtual and function pointer based calls) and all targets. For the *Bin types*, *Safe src types*, and *Src types* policies depicted above the targets can be virtual or non-virtual, for the remaining policies the targets inherently can only be virtual functions. Targets: median (minimum and maximum) number of legal function targets per callsite. Target distribution: minimum/90th percentile/maximum number of targets per callsite.

Table 8.2 depicts the static target constraints for the NodeJS program under different static CFI calltarget constraining policies. Table 8.2 provides the minimal and maximum values of virtual calltargets which are available for a virtual callsite after one of the eight CFI policies is applied. MKSnapShot contains the Chrome V8 engine and is used as a shared library by NodeJS after compilation. We decided to add MKSnapshot in Table 8.2 as this component is strongly used by NodeJS and represents a source of potential calltargets. The NodeJS results were obtained after static linking of MKSnaphot.

The targets median entries in Table 8.2 (left hand side) indicate the median values obtained for independently applying one of the eight CFI policies to NodeJS. For both NodeJS and MKSnaphot, the best median number of residual targets is obtained using the following policies: (1) *vtable hierarchy*, (2) *Sub-hierarchy*, and (3) *Precise sub-hierarchy*. These results indicate that these three CFI policies provide the lowest attack surface while the highest attack surface is obtained for the *Bin types* policy, which allows the highest number of virtual and non-virtual targets.

The targets distribution in Table 8.2 (right hand side) shows the minimum, maximum and 90 percentile results for the same eight policies as before. While the minimum value is 0 the highest values for both NodeJS and MKSnapshot are obtained for the *Bin types* policy, while the lowest values are obtained for the following policies: (1) *vtable hierarchy*, (2) *Sub-hierarchy*, and (3) *Precise sub-hierarchy*. Further, the 90p results show that on the tail end of the distribution, a noticeable difference between the three previously mentioned policies exists. We can observe that for these critical callsites the *Precise sub-hierarchy* policy provide the least amount of residual targets and therefore the best protection against CRAs. Meanwhile, the 90p results for the *Precise src type* and *vtable hieachy* policies indicate that the residual attack surface might still be sufficiently large for the attacker.

8.7.2 Generalized Results

Table 8.3 depicts results for the three policies which can provide protection for both `C` and `C++` programs. In contrast to Table 8.4, all indirect calls are taken into account (including virtual calls). Therefore, the targets can be virtual or non-virtual. Intuitively, the residual attack surface grows with the size of the program. This can be observed by comparing the results for large (*e.g.,* Chromium) with smaller (*e.g.,* Memcached) programs.

Table 8.4 depicts the overall results obtained after applying the eight assessed CFI policies to virtual callsites only. The first four policies (italic font) cannot differentiate between virtual and non-virtual calltargets. Therefore, for these policies the baseline of possible calltargets includes all functions (both virtual and non-virtual). This is denoted with Baseline all func. Since the remaining four policies can only be applied to virtual callsites, they restrict the possible calltargets to only virtual functions. Thus, the baseline for these policies includes only virtual functions (Baseline virtual function). For a better comparison between the first and second categories of policies, we also calculated the target set when restricting the first four policies to only allow virtual callsites. For *Bin types*, *Safe src types*, *Src types*, and *All vtables* the results indicate that there is *no* protection offered. The three-class hierarchy-based policies perform best when considering the median and average results. In addition, the *Precise src type* policy performs surprisingly well, especially after restricting the target set to only virtual functions.

8.7.3 Ranking of CFI Policies

In this Section, we normalize the results presented in RQ2 using the *Baseline* values, (*i.e.,* the number of possible target functions), in order to be able to compare the assessed CFI policies

		Callsites		**Targets (Non-)VFunctions**		
Program	*Value*	*Write con-straints*	Baseline all func.	*Bin types*	*Safe src types*	*Src types*
	Min			885	0	0
	90p			30,179	5,751	5,751
NodeJS	Max	none	32,478	32,478	5,751	5,751
	Median			21,950	88	56
	Avg			20,787	1,242	1,099
	Min			357	0	0
Apache	90p			5,450	1,315	1,315
Traffic	Max	none	6,201	6,201	1,315	1,315
Server	Median			2,608	1,315	1,315
	Avg			3,350	840	835
	Min			3,612	0	0
	90p			201,477	64,315	64,315
Chromium	Max	none	232,593	232,593	64,315	64,315
	Median			97,041	8,672	7,394
	Avg			132,182	27,238	27,074
	Min			99	0	0
	90p			1,793	160	160
Httpd	Max	none	1,949	1,915	160	160
	Median			1,070	18	16
	Avg			1,017	53	48
	Min			37	0	0
	90p			582	44	44
Lighttpd	Max	none	594	582	44	44
	Median			395	6	6
	Avg			388	17	17
	Min			92	0	0
	90p			155	2	2
Memcached	Max	none	225	221	17	17
	Median			155	2	2
	Avg			157	2	2
	Min			422	1	1
	90p			1,172	149	149
Nginx	Max	none	1,270	1,259	149	149
	Median			719	75	75
	Avg			697	81	81
	Min			1,266	1	1
	90p			2,437	54	54
Redis	Max	none	2,880	2,635	391	391
	Median			1,994	16	14
	Avg			1,877	36	35

Table 8.3: Virtual and pointer based callsites results.

against each other w.r.t. calltarget reduction. This allows for a better comparison of programs with different sizes and complexities.

Table 8.5 depicts the average, standard deviation and 90th percentile results obtained after analyzing only virtual callsites. Unless stated otherwise, we use the CTR_{SD} metric depicted in Equation Equation 8.2 to compute the standard deviation (SD) for the assessed programs. For these callsites, all eight CFI policies can be assessed.

We calculated the average over the three `C++` programs after normalization. By considering these aggregate average values, the eight policies can be ranked (from best (smallest aggregate

		Callsites	Targets Baseline		Virtual Function Targets							
Program	*Value*	*Write constraints*	Baseline all func.	Baseline virt. func.	*Bin types*	*Safe src types*	*Src types*	*Precise src types*	*all vtables*	*vtable hierarchy*	*Sub-hierarchy*	*Precise sub-hierarchy*
	Min				12,545 (1,956)	0 (0)	0 (0)	0 (0)	6,128	0	0	0
	90p				30,179 (4,078)	5,751 (810)	5,751 (810)	499 (10)	6,128	54	7	6
NodeJS	Max	none	32,478	6,300	32,478 (4,455)	5,751 (810)	5,751 (810)	730 (243)	6,128	243	243	243
	Median				21,950 (3,106)	2,885 (426)	1,511 (121)	3 (3)	6,128	2	2	2
	Avg				19,395 (2,793)	2,406 (414)	2,113 (354)	86 (12)	6,128	14	8	8
	Min				2,608 (232)	1 (0)	1 (0)	0 (0)	788	0	0	0
	90p				4,085 (546)	1,315 (97)	1,315 (97)	17 (13)	788	34	7	7
Traffic Server	Max	none	6,201	796	6,201 (710)	1,315 (159)	1,315 (159)	18 (16)	788	42	18	18
	Median				2,608 (232)	1,315 (97)	1,315 (97)	17 (13)	788	7	1	1
	Avg				3,122 (321)	928 (76)	923 (74)	11 (9)	788	10	3	3
	Min				97,041 (37,873)	0 (0)	0 (0)	0 (0)	68,560	0	0	0
	90p				201477 (63,816)	64,315 (24,661)	64,315 (24,661)	48 (30)	68,560	192	25	15
Chromium	Max	none	232,593	78,992	232,593 (71,000)	64,315 (24,661)	64,315 (24,661)	3,029 (509)	68,560	4,486	4,486	4,486
	Median				97,041 (37,873)	8,672 (4,593)	7,633 (4,593)	3 (2)	68,560	6	2	2
	Avg				128,452 (45,731)	29,315 (11,119)	29,127 (11,013)	57 (19)	68,560	78	37	32

Table 8.4: Evaluation of virtual callsites for only C++ programs. Baseline all func. represents the total number of functions, while Baseline virtual func. represents the number of virtual functions. The first four policies (from left to right in italic font) allow virtual or non-virtual targets, while the remaining policies inherently allow only virtual targets. The values in round brackets show the theoretical results after adapting the first four policies to only allow virtual targets. Each table entry contains five aggregate values: minimal, 90p: minimum/90th percentile/maximum, maximal, median and average (Avg) number of targets per callsite.

	Baseline	Bin types			Safe src Types			Src types			Precise src types			all vtables			vtable hierarchy			Sub-hierarchy			Precise sub-hierarchy		
Program		*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*
NodeJS	6,300	59.72	21.0	92.92	7.41	6.32	17.71	6.51	6.44	17.71	0.26	0.54	1.54	97.27	0.0	97.27	0.23	0.63	0.86	0.13	0.46	0.11	0.13	0.46	0.1
Tr. Server	796	50.35	15.79	65.88	14.97	8.89	21.21	14.89	9.01	21.21	0.18	0.12	0.27	98.99	0.0	98.99	1.26	1.27	4.27	0.34	0.51	0.88	0.34	0.51	0.88
Chromium	78,992	55.23	19.08	86.62	12.6	12.16	27.65	12.52	12.22	27.65	0.02	0.11	0.02	86.79	0.0	86.79	0.1	0.43	0.24	0.05	0.41	0.03	0.04	0.41	0.02
average	28,696	55.1	18.62	81.8	11.66	9.12	22.19	11.3	9.22	22.19	0.15	0.25	0.61	94.35	0.0	94.35	0.53	0.77	1.79	0.17	0.46	0.34	0.17	0.46	0.33

Table 8.5: Normalized results using virtual callsites only. All results are normalized using Baseline. Baseline: Total number of possible virtual targets. Each entry contains three aggregate values: average-, standard deviation (SD) and 90p-number of targets per callsite.

average) to worst (highest aggregate average)) as follows: (1) *Precise src types* (0.15), (2) *Precise sub-hierarchy* (0.17), (3) *Sub-hierarchy* (0.17), (4) *vtable hierarchy* (0.53), (5) *Src types* (11.3), (6) *Safe src types* (11.66), (7) *Bin types* (55.1), and (8) *All vtables* (94.35).

From the class hierarchy-based policies *Precise sub-hierarchy* perform best in all three aggregate results (Avg, SD and 90p). In comparison, *Precise sub-hierarchy* performs better w.r.t. average and standard deviation but worse w.r.t. 90p. These results indicate that these two policies are the most restrictive, but a clear winner in all evaluated criteria cannot be determined.

Table 8.6 depicts (similarly as Table 8.5) normalized results with the difference that all indirect callsites (both virtual and pointer based) are analyzed. Thus, the *Baseline* values used for normalization include virtual and non-virtual targets. By taking into account the aggregate averages and the standard deviation of the three policies in Table 8.6, we can rank the policies as follows (from best to worse): (1) *Src types* (Avg 5.3 and SD 5.17), (2) *Safe src types* (Avg 5.4 and SD 5.18), and (3) *Bin types* (Avg 60.3 and SD 34.39).

Further, by considering the 90p values we conclude that for the most vulnerable 10% of callsites, *Bin types* only restricts the target set to 87.9% of the unprotected target set. These callsites essentially remain unprotected. Meanwhile, the *Safe src type* and *Src type* policies restrict to only around 12% of the unprotected target set.

		Bin types			**Safe src types**			**Src types**		
Program	*Baseline*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*	*Avg*	*SD*	*90p*
NodeJS	32,478	64.0	20.43	92.92	3.82	5.83	17.71	3.38	5.64	17.71
Tr. Server	6,201	54.03	18.76	87.89	13.54	9.27	21.21	13.46	9.36	21.21
Chromium	232,593	56.83	19.84	86.62	11.71	12.11	27.65	11.64	12.16	27.65
Httpd	1,949	52.18	26.5	92.0	2.7	3.01	8.21	2.46	3.01	8.21
Lighttpd	594	65.25	27.81	97.98	2.94	3.18	7.41	2.93	3.19	7.41
Memcached	225	69.75	7.11	68.89	1.0	0.97	0.89	1.0	0.97	0.89
Nginx	1,270	54.91	24.85	92.28	6.38	4.56	11.73	6.36	4.57	11.73
Redis	2,880	65.19	16.51	84.62	1.25	2.52	1.88	1.2	2.52	1.88
average	34,773	60.3	34.39	87.9	5.4	5.18	12.09	5.3	5.17	12.08

Table 8.6: Normalized results using all indirect callsites.

8.7.4 Constructing Code Reuse Attacks

In this Section, we show how LLVM-CFI is used to build an attack which bypasses a state-of-the-art CFI policy-based defense, namely VTV's *Sub-hierarchy Policy*. This case study is architecture independent, since LLVM-CFI's analysis is performed at the IR level during LTO time in LLVM. Note that LLVM IR code represents a higher level representation of machine code (metadata), thus our results can be applied to other architectures (*e.g.,* ARM) as well. Our case study assumes an ideal implementation of VTV/IFCC. Breaking the ideal instrumentation shows that the defense can be bypassed in any implementation.

In this case study, we present the required components for a COOP attack by studying the original COOP attack against Mozilla's Firefox web browser and demonstrate that such an attack is easier to perform when using LLVM-CFI. Thus, we discuss the importance of the CRA construction with LLVM-CFI at hand.

The original COOP attack presented by Schuster *et al.* [238] used: (1) a buffer overflow filled with six fake counterfeit objects by the attacker, (2) precise knowledge of the Firefox `libxul.so` layout, (3) where an COOP dispatcher gadget (`ML-G`) resides, and (4) several other useful gadgets in order to open an Unix shell.

In general terms, to pursue a CRA the attacker needs: (1) an exploitable memory corruption, (2) attack starting point (*i.e.,* callsite) becomes corruptible due to (1), (3) program binary mtemory layout leak, (4) appropriate (usable and available) gadgets, (5) the possibility to read and write into memory, (6) the possibility to link gadgets and pass information from one to each other, and (7) the possibility to perform calls into the system `stdlib` or any other reach functionality library.

As demonstrated, the attacker first has to find an exploitable memory corruption (*e.g.,* buffer overflow, *etc.*) and fill it with fake objects. Next, the attacker calls to different gadgets (virtual functions) located in the program binary. As such, we assume that NodeJS contains an exploitable memory vulnerability (*i.e.,* buffer overflow), and that the attacker is aware of the layout of the program binary. The attacker would then want to bend the control flow to only per callsite legitimate calltargets since he does not know if other defenses are in place. He would also want to avoid calling into other program class hierarchies. Therefore, he needs to know which calltargets are legitimate for all callsites in the main NodeJS binary.

Ten controllable				**Target Policies**							
Callsites	#	**Baseline only vFunc.**	**Baseline all Func.**	*Bin types*	*Safe src types*	*Src types*	*Precise src types*	*All vtables*	*vtable hierarchy*	*Sub-hierarchy*	*Precise Sub-hierarchy*
debugger.cpp:1329:33	5	6,300	32,478	31,305	4	4	1	6,128	1	1	1
protocol.cpp:839:60	2	6,300	32,478	21,950	719	719	49	6,128	57	53	49
schema.cpp:133:33	3	6,300	32,478	27,823	136	136	1	6,128	1	1	1
handle_wrap.cc:127:3	1	6,300	32,478	12,545	810	810	1	6,128	72	12	12
cares_wrap.cc:642:5	1	6,300	32,478	1,956	810	810	1	6,128	72	13	13
node_platform.cc:25:5	1	6,300	32,478	1,956	810	810	6	6,128	20	19	19
node_http2_core.h:417:5	3	6,300	32,478	1,956	810	810	6	6,128	20	19	19
tls_wrap.cc:771:10	2	6,300	32,478	3,106	35	35	8	6,128	48	13	5
protocol.cpp:839:60	2	6,300	32,478	3,106	2,984	2,984	49	6,128	53	53	49
protocol.cpp:836:60	2	6,300	32,478	3,106	719	719	49	6,128	53	53	19

Table 8.7: Ten controllable callsites and legitimate targets.

Table 8.7 depicts ten controllable callsites (in total LLVM-CFI found hundreds of controllable callsies) for which the legitimate target set, depending on the used CFI policy, ranges from one to 31,305 calltargets. # depicts the number of passed parameters.

For each calltarget, LLVM-CFI provides: file name, function name, start address and source code line number such that it can be easily traced back in the source code file. The calltargets (right hand side in Table 8.7 in italic font) represent available calltargets for each of the eight assessed policies.

For our case study, we decided not to use the most restrictive CFI policy (*i.e., Precise sub-hierarchy*) as it enforces the same function in the virtual table sub-hierarchy. As such, only through inheritance, chances are that the implementation of the function may vary from the initial implementation. Thus, the number of useful gadgets in this sub-hierarchy is low.

Consequently, we assume that the attacker knows that the NodeJS binary is protected with the *Sub-hierarchy* policy. The attacker ranks all callsites depending on the number of usable calltargets. By ranking the callsites w.r.t. their residual target set, the attacker wants to know the precise functions (calltargets) which are allowed for each callsite. Further, in order to perform the attack, he has to access the source code of the application. After searching through the source code, he finds several callsites where vector-based object dispatches are performed. The attacker next finds out that the detected calltarget set contains several usable gadgets. Note that for the COOP attack to be performable, the only hard constraint is that at least one usable `ML-G` gadget must exist in the program. This constraint was addressed by the attacker at the beginning of their source code search since they decided to only look for callsites which are part of an `ML-G` COOP gadget. As such, the attacker knows exactly which gadgets are available for each controllable callsite. As none of the static CFI policies impose write constraints by default (*i.e.,* none) the attacker can overflow a buffer at a certain callsite with fake objects. Further, they start to call their gadgets one by one and passing information over the stack through scratch registers from one gadget to the next one. Note that their attack does not violate the CFI check in place at the selected object dispatch location since this is contained in a `ML-G` gadget. Further, he can deliberately avoid violating the original program control flow by calling into other program class hierarchies or other illegitimate calltargets as the original COOP attack does. Eventually, the attacker succeeds in opening an Unix shell, similarly to the COOP.

This CRA construction shows the usefulness of LLVM-CFI for an attacker, which can discover gadgets that are legitimate for each callsite after a certain CFI policy was applied. This is in order to better tailor his attack w.r.t. a deployed CFI-based defense. Finally, this demonstrates the usefulness of LLVM-CFI for an analysis.

8.8 Discussion

8.8.0.1 Limitations

In this Section, we present some limitations of LLVM-CFI and discuss other important consideration w.r.t. our metrics, assessing of CRAs defenses, gadget linkalibility, and successful exploitation.

Evaluation Results. We are aware that our evaluation results reflect the measurements obtained by applying LLVM-CFI on several real-world open source programs which may not be generalizable. We therefore believe that more programs should be evaluated with the help of LLVM-CFI (the first CFI-based consistently reasoning framework for static CFI defenses). Thus, we urge other researchers to use LLVM-CFI's CFI-based metrics when assessing their defense and comparing it with other existing tools. Finally, we envisage that LLVM-CFI could become the de-facto-standard benchmarking framework for assessing static CFI-based defenses.

Automated Gadget Chain Construction. We are aware of the urgent need for building tools which can generate complete gadget chains. Thus, we provide within LLVM-CFI the first step towards building such a chain by first filtering gadgets (calltargets) which are still accessible through bending even in the presence of state-of-the-art CFI defenses. While we simplify this task by modeling a CFI policy and by helping to detect CFI-resistant gadgets, the last step is still manual. Also, to the best of our knowledge, there is no CFI policy aware open source tool which can build gadget chains fully automatically. Thus, an approach to automatically build a gadget chain is to implement a data flow analysis used for checking how data flows from one gadget to the next. This can be implemented at the IR or source code level in LLVM-CFI. As LLVM-CFI operates at LLVM's IR level, this data flow analysis could be used for example to check if gadgets are compatible with each other. This can be done by studying the possibility of transferring data and forming a gadget chain between them.

8.8.0.2 Other Considerations

In this Section, we present some limitations of LLVM-CFI and discuss several other key considerations.

Our Metrics vs. other Metrics. Existing CFI-defense assessing metrics (*e.g.,* AIR, fAIR and AIA) are designed to reason about CFI based defenses by providing average values obtained mostly by dividing the total number of control flow transfers to the total number of callistes for example. Also, when computing these values the ground truth numbers w.r.t. available calltargets, total number of calltargets, total number of unprotected calltargets are in most cases not provided. Thus, it is hard for any researcher which looks for reproducibility of these results to compare his results with other research results. Further, these metrics do not reason about other aspects of CFI-based defenses such as the forward check runtime overhead, return site target reduction, return site target runtime overhead and availability of gadgets at the end of the forward or backward edges. Thus the only benefit of these metrics are some average numbers which are hard to reproduce because of the above mentioned reasons. Finally, similar to LLVM-CFI these three metrics do not reason about gadget link-ability which would help to shed light in areas of optimal CFI-based protection schemes which then could be effectively used to protect applications without the need to enforce a strict CFI-policy on the whole program binary or libraries. More specifically, think as if the defender could harden a C++ based application such that all main loop gadgets (and alike) are made unusable for any attacker. This would remove a crucial building block for a COOP attack and thus the attacker would not have an important building block in his arsenal. Thus, the COOP attack would not be possible.

LLVM-CFI's metrics are superior compared to AIR, fAIR and AIA metrics since these allow to quantify static CFI-based defenses w.r.t. to more dimensions. Further, LLVM-CFI allows to precisely reason about the forward edge based calltarget reduction by allowing to better compare the obtained results with ground truth numbers, by for example not averaging the results and providing the first framework which allows to comprehensively reason about a CFI defense. Finally, LLVM-CFI can be used to reason quantitatively and consistently about object-oriented programing (OOP) concepts which represent the building blocks for many CFI defenses.

Assess other Code Reuse Attack Defenses. LLVM-CFI can be extended to assess other types of attacks by first identifying the set of gadgets the given specific attack operates on. Next, these gadgets have to be classified w.r.t to their start and end format, (*i.e.,* how do this gadgets look like). Further, the gadgets should be classified as benign- or malicious w.r.t. the particular attack being assessed. For example, most of the previous CRAs were using the return instructions to link gadgets together. This clearly violates the caller/callee calling convention which is modeled and consistently integrated into COOP. Next, from all metrics available to LLVM-CFI, one has to determine which of the metrics makes sense for the particular attack. For example, some attacks just use forward edges while others exclusively use backward edges.

Assess Gadget Linkability. LLVM-CFI can be extended to measure gadget linkability, by determining gadget availability. For example, this concept is used to check if a certain gadget is present in the program and usable (*i.e.,* which assures that a gadget is usable for a certain type of attack). In order to assess gadget linkability, a policy-based language can be used. This can be used to assess if any of two available and usable gadgets can be linked together to create a gadget chain. A first step in this direction is to analyze the data flow between gadgets by using this policy language. Any gadget linkability policy should assess wether—through a chain of consecutive gadget interactions—the program stack or the heap can be sequentially modified such that it can be used in the next step of the attack that uses the previous results computed or prepared on the stack or heap or both. Note that a first step towards this direction was made by

Johnson *et al.* [128] and Ispoglou *et al.* [121] which assesses gadget linkability with the help of a policy determining if any two gadgets can be successfully linked together by analyzing the data flow between them.

Assess Successful Exploitation. As far as we know, there is no research on how to assess whether a code reuse attack was successful or not. Further, as noted by [36], successful exploitation primarily depends on the goals of the attack which can only be specified by the attacker. We note that any final or intermediate goal of an attack can be determined by taking into account gadget (1) availability, (2) usability, (3) linkability and (4) attacker goals which are basically final or intermediate steps in the linkability deciding process.

By first implementing the capability of gadget linkability into LLVM-CFI, it can be determined through the help of the same or another policy language if intermediate or final attacker goals are reached. Note that an attack is considered successful if all goals stated by the attacker are reached during the attack. This basically depends on (1)-(4) mentioned above and the possibility to define sub-linkability gadget chains for any given chain. Note that sub-linkability chains represent attack sub-goals while the successful traversal of all sub-goals represents the reachability of the final attack goal, which decides if an attack was successful or not.

8.9 Summary

Static source code analysis tools used for assisting an analyst during the search for code reuse gadgets inside the source code of real-world programs can effectively be used by even a low-effort attacker to find defense-aware gadgets in order to craft practical attacks. This stands in contrast to previous beliefs that such analysis is limited.

In this Chapter, we presented LLVM-CFI, a CFI defense aware assessment and gadget search framework which can be used for both defensive and offensive purposes. We implemented LLVM-CFI, our CFI defense policy assessment framework based on static compiler which offers the possibility to effectively analyze real-world programs. By using LLVM-CFI, an analyst can drastically cut down the time needed to search for gadgets which are compatible with state-of-the-art CFI defenses contained in many real-world programs. Our experiment results indicate that most of the CFI defenses are too permissive. Further, if an attacker does not only rely on the program binary when searching for gadgets and has a tool such as LLVM-CFI at hand to analyze the source code of the vulnerable application, then many CFI defenses can be easily bypassed.

Chapter 9

ρFEM: Backward-edge Protection Using Reversed Forward-edge Mappings

In this Chapter, which belongs to the second part of this thesis, we designed and implemented a compiler based tool, called ρFEM, which is based on the lessons learned in Chapter 8. ρFEM protects program CFG backward edges stemming from indirect and direct forward-edge program control flow transfers. In this way, we were able to address **RQ6** and propose a novel technique for protecting program CFG backward-edges with the help of a fine grained CFI policy which provides an optimal set of return targets for each protected callee. In this way the likelihood of successfully performing CRAs which exploit backward edges is greatly reduced and at the same time we provide in this Chapter a solution which is an alternative for shadow stack based techniques. In contrast to shadow stack techniques, ρFEM does not rely on entropy and information hiding. Thus, the according protection disclosing vectors do not apply. Finally, note that parts of this Chapter have been published by Muntean *et al.* [199].

9.1 Introduction

What is the overall view? Code reuse attacks (CRAs), such as return oriented programming (ROP) [31] attacks, have threatened software systems for more than a decade. CRAs are possible due to the fact that statically building a precise program control flow graph (CFG) is theoretically not possible. Building the CFG requires precise program alias analysis (*e.g.,* source code, binary, intermediate representation (IR)), which is undecidable [229]. Consequently, the obtained CFG is an over or under approximation of the real program's CFG. In addition to control flow forward-edge (*i.e.,* `jump`, `call`) violations, backward-edge (*i.e.,* function returns, `ret`) violations play a crucial role in facilitating CRAs.

What is the problem? For example, an attacker can *bend*/modify [36] the intended control flow of a program during runtime in such a way that he calls consecutively assembly code sequences (gadgets) contained in the original program using CFG edges belonging to the program's CFG in order to achieve his malicious objective. As such, one of the main building blocks (along forward edges) of chaining gadgets together are CFG backward edges. This gadget chaining process violates the caller-callee function calling convention enforced by any widely used application binary interface (ABI), *i.e.,* Itanium ABI [122], Microsoft ABI [101], and ARM ABI [9].

What are the existing solutions? In general, there are two approaches to protect backward-edges integrity: (1) check-based approaches which check if a `ret` goes to legitimate targets, and (2) integrity-based approaches which ensure the integrity of the return addresses in the stack. Compared to `call` instructions, `ret` instructions are executed more frequently, thus the check-based approach tends to introduce high performance overhead. Since `ret`'s target (*i.e.,* return address) is determined by its call instruction, if we can efficiently ensure the integrity of return addresses, then we can ensure the required program protection. We can not only guarantee an unique target for a return instruction, but also avoid checks. For a long time, shadow stack techniques [63] (integrity-based approach) along with call-preceded return enforcement (*e.g.,* BinCFI [290]), control-flow locking [19], pointer encryption, and double stacks [63] were the only available solutions against backward-edge violating attacks. Shadow stack techniques are based on building a second shadow stack for every program stack on which function return addresses are pushed before entering a certain function. They are then checked before leaving the function and finally are being popped after the check was performed. This ensures that each backward edge respects the used function calling convention by partially simulating the original stack in parallel to the program execution.

However, with the growing demand for fully-precise control flow integrity (CFI) solutions, it is commonly understood that any CFI-based protection mechanism not having a shadow stack-based policy to protect the backward edges is broken [36]. For this reason, a myriad of tools, which implement shadow stack techniques, were proposed over the last years: (1) binary based [1, 156, 216, 289, 290, 187, 266, 221], (2) source code based [274, 19, 209, 224, 61, 211, 160, 90, 210, 212, 49, 20], and (3) other types [44, 66, 284]. Shadow stack's integrity can be protected by separate mechanisms such as: (1) information hiding (randomization, mostly used) or (2) software-based fault isolation (SFI), *etc.* SFI can help to protect shadow stacks by instrumenting memory writes. Essentially, the system can set up a memory region for storing the shadow stack and then instrument a memory write in the program to ensure it does not write to that memory region. Note that in general, all shadow stack/safe stack techniques that do not enforce integrity but rely on information hiding are weak. At the same time, integrity is generally more costly than information hiding. To the best of our knowledge, only Intel has released its hardware (HW) shadow stack support, Intel CET [118]. This HW feature allows to build shadow stack more efficiently. The common belief with this approach is that integrity-based approaches will outperform the check-based approaches, in terms of both security and performance. Microsoft's RFG [282] (*i.e.,* OS-dependent implementation, and is as of 2018 removed from the Windows bounty program [168]), which is closed source, and no performance data was published. Further, Clang's SafeStack [49] (*i.e.,* compiler-based code instrumentation) has been widely used. In addition, the original binary-based CFI implementation of Abadi *et al.* [1] is not public and cannot be evaluated. Also, target sets are not discussed and it was built using the closed source binary rewriting framework Vulcan. As such, a compiler-based approach is likely better but cannot be compared with the original approach, since this is not public.

In general, shadow stack techniques (*i.e.,* source code and binary based) are precise due to the one-to-one mapping between caller-callee pairs, have the following limitations: (1) increased binary size (shadow stack at most double the amount of memory used for stacks), especially in certain embedded scenarios, when not used in conjunction with hardware features such as the Intel's CET and the like, (2) source code based approaches provide no protection to closed source software (note that runtime based tools *e.g.,* ROPdefender [67] and binary based tools can protect backward edges in binaries as these can still monitor and instrument the binary), (3) compatibility issues with common software practices such as longjumps/setjumps, exceptions, and continuations, that perform unconventional control flows, (4) relatively high runtime overhead (up to 10% for a traditional shadow stack [63], mostly for binary based tools) and considerably lower runtime overhead (around 1%) when using hardware features such as Intel's CET or are based on compiler support, see Clang's SafeStack) due to implementation-specific reasons. Further, it is interesting to note that in real programs there is no correlation between the number of callsites a function may return to and how often it is called. Also, the performance of shadow stack techniques is dependent on function call frequency, as functions, which are more frequently called introduce a higher performance overhead.

What are the limitations of the solution? Unfortunately, information hiding was shown to be generally easy to bypass (and any other information hiding [94] technique with it) and thus, the information hiding component of Clang's SafeStack [49] or GCC shadow stack [88] can also be bypassed. This bypass demonstrated that at least four attack vectors exist for disclosing the two-separated-stacks principle on which Clang's SafeStack is based. Namely, guessing oracles, neglected pointers, thread spraying, and allocation oracles can all independently be used to bypass this protection technique. These attack vectors [94] do not reveal a particular limitation of Clang's SafeStack, but rather, that information hiding, and thus SafeStack, which relies on this technique, can be bypassed. Similar to other CFI solutions, there is no data that needs to be integrity protected when using ρFEM since all control metadata used in ρFEM is static. Finally, shadow stack techniques have some readable/writable data that either needs to be integrity protected (with additional overhead) or may be accessible by an attacker. Compared to shadow stack techniques, ρFEM's metadata is read-only, similar to other CFI techniques.

What is our insight? In this thesis, we present ρFEM, a context-insensitive, compile-time software instrumentation tool used to enforce a fine-grained CFI-based policy. ρFEM protects backward edges by checking them before transferring control flow. Note that the precision of ρFEM's backward-edge mappings relies primarily on the forward edge mappings. As ρFEM computes the forward edge mappings for indirect transfers, we could easily implement a CFI policy, which protects forward-edges as well. In this work, we decided to only protect backward edges, due to the fact that this research area is considerably less researched, than forward-edge defenses. ρFEM builds return sets for calltargets by: (1) computing a return set for each non-virtual calltarget based on reversing a precise function signature forward-edge mapping and (2) calculating a return set for each virtual calltarget based on reversing a precise class virtual table hierarchy forward-edge mapping. Both of these forward-edge mappings are used to compute the legitimate callee return address set for each calltarget by enforcing the caller/callee function calling convention. ρFEM protects backward edges: (1) without relying on information hiding, and (2) with low runtime overhead and binary blow-up while providing a minimal callee return address set, and (3) by not relying on OS-specific support. The runtime overhead is small (around 1%) due to our novel backwards reflected class hierarchy technique, which helps to build efficient `ID` sets for each backward edge (virtual functions). The precise forward edge mapping considerably reduces the legal callee's return target set from arbitrary targets inside the program binary to minimal `ID` sets. This allows us to perform our checks efficiently and precisely. We evaluate ρFEM with real-world programs in addition to all pure C/C++ programs contained in the SPEC CPU2017 benchmark and report a low runtime overhead, while maintaining high target address precision. ρFEM provides a trade-off, giving great performance at a slight security loss.

What are our contributions? In summary, in this Chapter, we make the following contributions:

- We design a novel fine-grained backward-edge protection technique by not relying on information hiding.
- We implement our technique based on the Clang/LLVM compiler framework inside a prototype called ρFEM.
- We evaluate ρFEM thoroughly with Google Chrome, NodeJS, Memcached, Nginx, Lighttpd, and the SPEC CPU2017 benchmarks. We achieve a low runtime overhead of 2.72% in geomean for the Google Chrome web browser, as well as of less than 1% in geomean for the SPEC CPU2017 benchmarks. ρFEM enforces less than 3.54 return targets per callee in geomean.

9.2 Threat Model

We assume a powerful attacker which can exploit any backward-edge based indirect program transfer and has the capability to make arbitrary memory writes. Note that ρFEM focuses exclusively on protecting indirect backward-edge based control flow transfers. We assume that other types of protection mechanism are in place for protecting against forward-edge based attacks. These defense mechanisms are orthogonal to our protection policy. Our approach does not rely on information hiding from the attacker and as such we can tolerate arbitrary reads. More precisely, we consider a powerful, yet realistic adversary model that is consistent with previous work on code-reuse attacks and mitigations [135]. We rely on several existing and complementary mitigations for comprehensive coverage.

Adversarial Capabilities.

- *System Configuration.* The adversary is aware of the applied defenses and has access to the source and non-randomized binary of the target application.

- *Vulnerability.* The target application suffers from a memory corruption vulnerability that allows the adversary to corrupt memory objects. We assume that the attacker can exploit this flaw to read from and write to arbitrary memory addresses.
- *Scripting Environment.* The attacker can exploit a scripting environment to process memory disclosure information at run time, adjust the attack payload, and subsequently launch a code-reuse attack.

Defensive Requirements.

- *Writable (xor) Executable Memory.* The target system ensures that memory can be either writable or executable, but not both. This prevents an attacker from injecting new code or modifying existing executable code.
- *Execute-only Memory.* We build on previous systems which enforce execute-only memory pages, *i.e.,* the CPU can fetch instructions but normal read or write accesses trigger an access violation.
- *JIT Protection.* We assume mitigations are in place to prevent code injection into the just in time (JIT) code cache and prevent reuse of JIT compiled code [211]. These protections are orthogonal to ρFEM.
- *Brute-forcing Mitigation.* We require that the protected software does not automatically restart after hitting a booby trap which terminates execution. In the browser context, this may be accomplished by displaying a warning message to the user and closing the offending process.

9.3 Design and Implementation

A `NOP`, no-op, or NOOP is an assembly language instruction available in all important used CPU instructions set architectures (*e.g.,* `x86`, `x86-64`, `SPARC`, `MIPS`, *etc.*) that does nothing. The approach of ρFEM is based on inserting new `NOP`s after the caller address and filling them with `ID`s before the callee returns to help constructing a mapping between legal callees and callers. This mapping is used to enforce a fine-grained CFI policy between legal callees and callers. To achieve this, ρFEM builds two separate caller-callee mappings: (1) for virtual functions (callees), and (2) non-virtual functions (callees). Next, we present the general approach of how these two mappings (sets) are constructed for a single callee.

9.3.0.1 Building Backward Edge Sets

Figure 9.1 depicts how backward edge return sets are built based on the program class hierarchy in (a) and (b) for virtual callsites, and based on a precise forward-edge function signature type mapping in (c) and (d) for function pointer-based callsites.

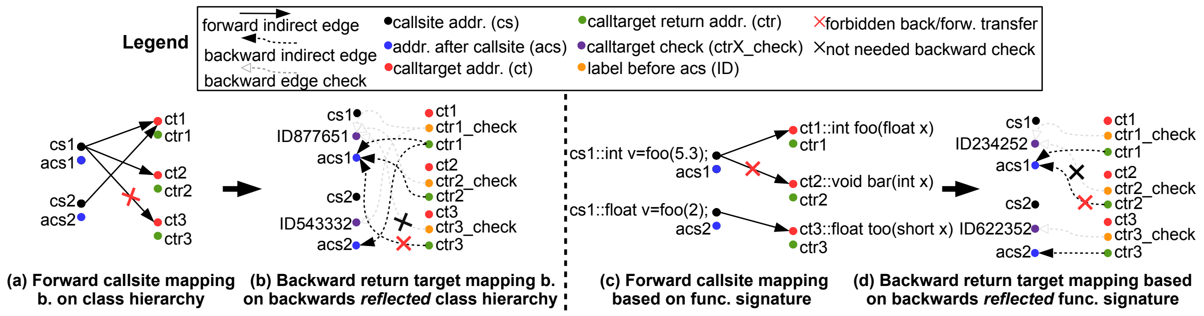

Legend
forward indirect edge
backward indirect edge
backward edge check
callsite addr. (cs)
addr. after callsite (acs)
calltarget addr. (ct)
calltarget return addr. (ctr)
calltarget check (ctrX_check)
label before acs (ID)
forbidden back/forw. transfer
not needed backward check
cs1 acs1 cs2 acs2 ct1 ctr1 ct2 ctr2 ct3 ctr3
ID877651 ID543332 ctr1_check ctr2_check ctr3_check
cs1::int v=foo(5.3);
cs1::float v=foo(2);
ct1::int foo(float x)
ct2::void bar(int x)
ct3::float too(short x)
ID234252 ID622352
(a) Forward callsite mapping b. on class hierarchy
(b) Backward return target mapping b. on backwards reflected class hierarchy
(c) Forward callsite mapping based on func. signature
(d) Backward return target mapping based on backwards reflected func. signature

Figure 9.1: Identifier (ID) based backward edge mapping of virtual functions called through object dispatches, (a) & (b) and ID based backward-edge mapping of non-virtual functions called indirectly through function pointers, (c) & (d).

Class Hierarchy. Forward edge information, provided by the class hierarchy, is used to map backward edges (see Figure 9.1(a) forward-edge mapping and (b) backward-edge mapping for more details). Essentially, all legitimate forward edges are *reflected* back. For each callee, a set containing all the calltarget's return sites is built. Depending on the base class of the dispatched object at the callsite and the location in the class hierarchy, the number of calltarget return sites differs. Note that our technique is aware of class inheritance and as such, inherited members along the class sub-hierarchy are included in the relevant backward-edge target set.

Function Signature. Mapping backward edges using function signatures (see Figure 9.1(c) forward-edge mappings and (d) backward-edge mapping) uses a precise forward-edge analysis based on determining a set of legitimate calltargets for each forward edge (callsite based on function pointer). The forward mapping is built by matching callsites and calltargets as follows. For each callsite the number of provided parameters, their type, and the return type is used to build a per function signature. This callsite information is matched with all calltargets in the program during compile time by comparing the number of parameters consumed, their type, and the return type with the data collected at the callsites (not object dispatches). For each matching function signature (calltargets) and callsite signature pair, a set is built. Finally, for each calltarget contained in the previously determined calltarget set, another per-calltarget return set is built by following the caller-callee function calling convention. This essentially means that for each calltarget-callsite mapping, the legitimate calltarget set contains the return address located after the legitimate callsite.

9.3.1 ρFEM Design

Figure 9.2 depicts the main components of ρFEM and where these reside in different parts of the Clang/LLVM compiler framework. The IDs and callee-side checks are generated as follows. The virtual class metadata collected in Clang ❶ is used to reconstruct the class hierarchy during LLVM-LTO ❷. Afterwards, metadata about the function entries in the virtual tables (vtables) is utilized together with the class hierarchy to generate IDs for virtual functions and corresponding ranges for virtual function callsites ❸.

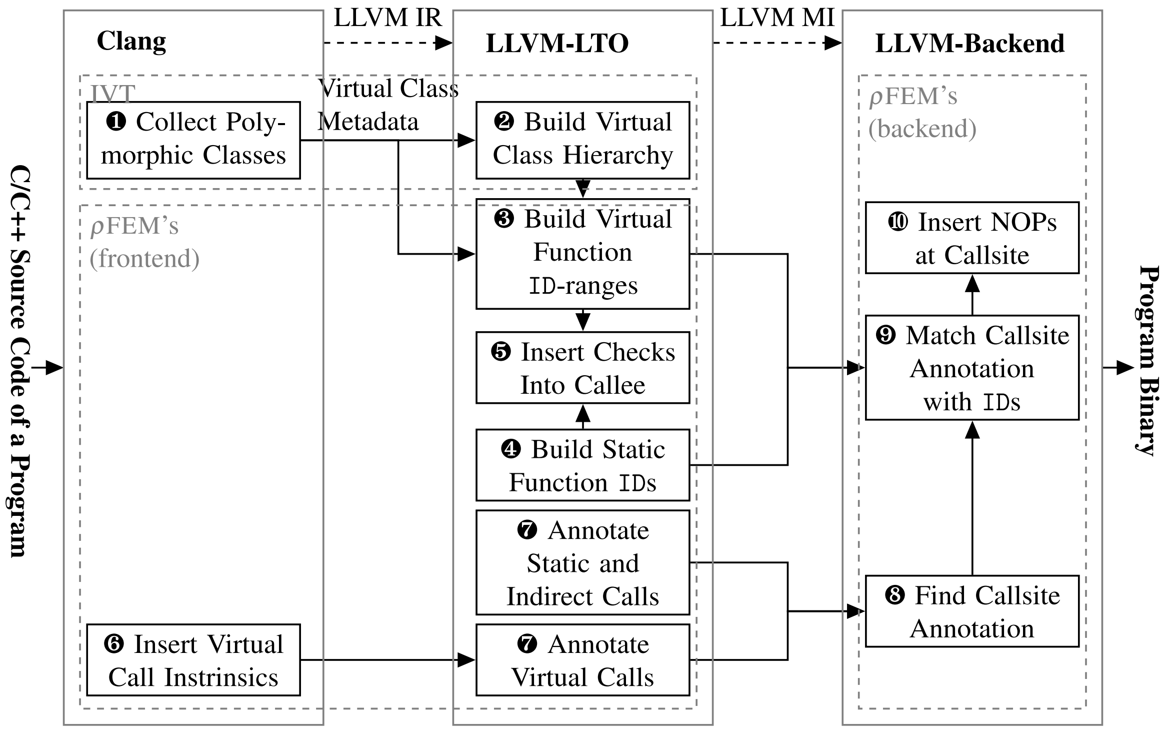

Clang
LLVM IR
LLVM-LTO
LLVM MI
LLVM-Backend
IVT
Virtual Class Metadata
❶ Collect Polymorphic Classes
❷ Build Virtual Class Hierarchy
ρFEM's (frontend)
❸ Build Virtual Function ID-ranges
❺ Insert Checks Into Callee
❹ Build Static Function IDs
❼ Annotate Static and Indirect Calls
❻ Insert Virtual Call Instrinsics
❼ Annotate Virtual Calls
ρFEM's (backend)
❿ Insert NOPs at Callsite
❾ Match Callsite Annotation with IDs
❽ Find Callsite Annotation
C/C++ Source Code of a Program
Program Binary

Figure 9.2: Design of ρFEM.

Once all virtual functions have been assigned an `ID`, ρFEM continues to assign `ID`s to the remaining non-virtual functions ❹. Finally, checks are inserted before the return instructions in each function (callee) using the virtual and non-virtual function `ID`s calculated beforehand ❺.

Further, for callsites `ID` assignment ρFEM performs the following steps. In the Clang front-end, each virtual function call is annotated using an LLVM intrinsic ❻. An intrinsic is a particular type of Clang program annotation, that allows marking of certain program parts. This enables inspection at a later phase along the compilation pipeline. During link-time optimization (LTO), these intrinsics are detected and the corresponding call instructions are further annotated for later back-end analysis ❼. All remaining calls, *i.e.,* the ones which were not annotated in the front-end, are marked as either direct calls or function pointer-based indirect calls ❼. In the LLVM back-end, all annotated calls are relocated ❽ and then matched with the correct `ID` or an `ID`-range depending on the annotation type ❾. That is, virtual callsites are assigned ranges of IDs, direct callsites are assigned the unique `ID` of the function called directly and callsites which use function pointers are assigned their respective function signature `ID`. Next, `NOP` instructions are inserted directly after the callsite, carrying the `ID` data as a *payload* ❿. Inserting the `NOP`s this late in the compiler analysis pipeline (*i.e.,* machine instruction (MI) generation stage) ensures that no instructions are placed between the callsites and our `NOP`s by a different LLVM pass.

9.3.2 Direct Call Analysis

Forward edges stemming from direct calls do not need to be protected, as the address to which the program control flow is transfered is fixed. This address is write protected and cannot be overwritten during runtime. The situation is different for callees which were called through forward direct calls. Since the attacker may manipulate the callee return address on the stack, this backward-edge needs to be protected. ρFEM handles backward edges returning from direct calls in the following way. Each direct function gets its own `ID`. Direct callsites only have a single target, for which a single valid `ID` can be determined. Therefore, it is sufficient to attach a single valid `ID` to the callsite.

9.3.3 Virtual Call Analysis

In this Section, we explain the virtual function analysis algorithm in detail by providing its invariant. ρFEM uses a modified version of the interleaved virtual tables (IVT) metadata as presented by Bounov *et al.* [24]. ρFEM constructs backward-edge mappings of legitimate return address sets for virtual functions (callee) by first building the virtual table hierarchy and enforcing the respective legitimate virtual table sub-hierarchy for each callee. This metadata consists of virtual table hierarchies for all class hierarchies of a program compiled with the Clang/LLVM compiler. Additionally, we modify this metadata layout to have it contain: (1) virtual table entry information, and (2) virtual table offsets which are used during our analysis. ρFEM uses this modified metadata to determine legitimate callsites and to infer forward-edge information. In the evaluation Section, we show that our modified metadata allows ρFEM to be backwards compatible, as it can run alongside other tools.

Invariant. In order to construct the forward-edge target set for a virtual callsite, we need to be able to represent the `ID`s of the functions contained in this set as a compact range. Note, that we do not need a total ordering of all function `ID`s. Instead, we only have to be able to build compact (no gaps) ranges.

Using the previously established terminology (see Section 2.7.4 for more details), we note that the object at a particular callsite can either be of precise class type or of any subclass type thereof. Furthermore, if the object has dynamic type of precise class, then, per definition, the callsite uses the implementation provided by the base class. Therefore, any subclass of a precise class can at most call the implementation found in the base class or an override of this implementation. Thus, the sub-graph of the vtable hierarchy rooted at the base class contains all possible function implementations (*i.e.,* all callees for this callsite). Hence, this sub-graph provides the set of all functions which can be called at this particular callsite, *i.e.,* the target set of the callsite.

Next, we describe how ρFEM uses this sub-graph to assign `ID`s in order to ensure the invariant previously described. There are four steps, which are explained in their order of execution. Note that the algorithm's steps one, two, and three run at link-time optimization (LTO), *i.e.,* at compile-time after static linking, while step four of the analysis runs in the LLVM back-end *i.e.,* after LLVM-IR was lowered to machine instructions, and right before the program binary is emitted.

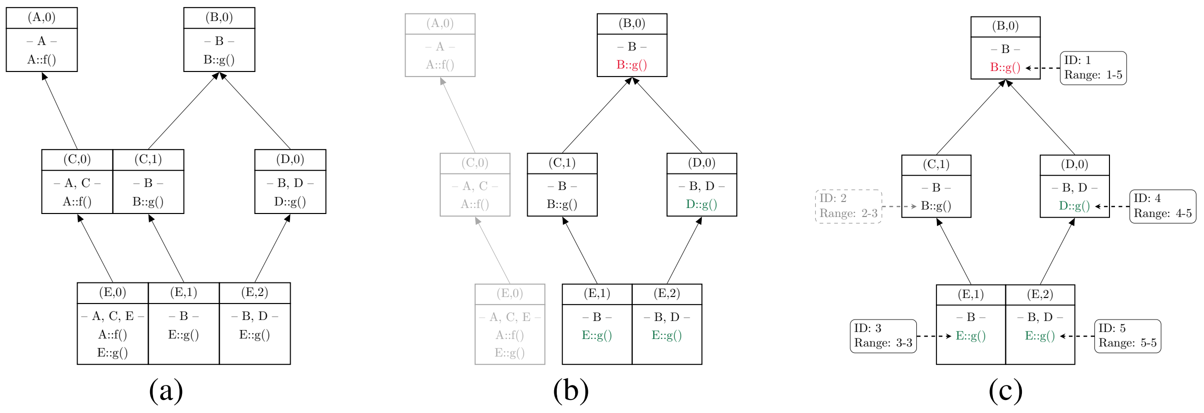


Figure 9.3: Steps used to determine IDs and the range for function *g()*. (a) Step one: Building class hierarchy from the virtual table hierarchies. (b) Step two: Collecting root class information of functions (shaded red) and overrides (shaded green). (c) Step three: Calculating ranges and IDs for function *g()*.

9.3.3.1 Step 1: Collecting function hierarchies

Using the modified virtual table hierarchy, a regular class hierarchy is reconstructed, *i.e.,* if a class has multiple vtables, they are merged into a single node in the class hierarchy. Figure 9.3(a) shows such a class hierarchy reconstruction. Each box depicted in Figure 9.3 represents a vtable and the first line in each vtable states the class name from which the vtable was inherited.

This class hierarchy allows ρFEM to collect information about each virtual function implementation. Firstly, the class in which the implementation was defined, and secondly, whether this class is the root class of the given function. If it is not the root class, then the implementation has been overridden by a sub-class implementation, *i.e.,*ρFEM is able to analyze all function overrides by using the vtable metadata.

This information is extracted from the class hierarchy as follows. ρFEM starts by topologically sorting the class hierarchy, which ensures that each parent is visited before any of its children. Then, the topologically-sorted list of classes is traversed and each function entry in the vtables of the class is inspected. In case a class contains a function implementation that was not previously encountered, the function is regarded as it would be implemented by this class. This happens, due to the fact that parents are visited before their children during pre-order traversal.

Finally, to differentiate between root classes and overrides (inheriting classes), we inspect the primary vtable of the direct parent. In case such a parent exists and the parents' primary vtable contains an entry at the same offset as the function in the child, this child overrides the entry in the parent with a new function implementation. Otherwise, the child defines a new function and becomes the root class for this function. Figure 9.3(b) shows the root class and the override information inferred by this step.

9.3.3.2 Step 2: Function-wise traversal

The root class information from step one allows ρFEM to inspect the sub-graph of the vtable hierarchy rooted at the root class. Note, that if we disregarded virtual inheritance this sub-graph would be a tree.

Using this information, ρFEM can iterate over all non-overriding functions and then in an inner loop iterate through the sub-graph rooted at the root class of the particular function. Note that the sub-graph contains individual vtables and not classes.

ρFEM assigns unique `ID`s to each vtable and function combination, while ensuring the following invariant. For a particular function, each parent vtable has to have a lower `ID` than all its children (`ID` value is smaller). This is similar to the well-established heap invariant and can be achieved using a pre-order traversal of the corresponding sub-graph. Thus, ρFEM starts with the first function, iterates through the sub-graph rooted at the root class of this function, and assigns `ID`s to each explored vtable. This is repeated for the sub-graph of each non-overriding function.

At the same time, the virtual call ranges are constructed as follows. Every vtable is assigned the range of `ID`s, containing its own `ID` and the `ID`s of all of its children. Because of the pre-order traversal, this results in a single closed (compact) range with its own `ID` having minimum value. Note that these ranges are assigned for each individual function/vtable combination (pair). In case that the sub-graph is a tree, then no virtual inheritance is involved. Consequently, each vtable gets at most one `ID` and one range per function contained in the vtable. Otherwise, the vtable will get multiple `ID`s and ranges, since with virtual inheritance a vtable can have multiple parents and therefore can be explored from multiple paths in the same sub-graph.

Figure 9.3(c) shows the `ID`s and ranges assigned for function *g()* with the sub-graph rooted at the vtable (B,0). Since in this example all classes use non-virtual inheritance, each vtable gets at most one `ID`. Note that the three vtables shaded in gray in the left part of Figure 9.3(b) are not part of the sub-graph and therefore have no `ID` assigned.

9.3.3.3 Step 3: Callee backward-edge checks

In this step, the callee-side checks are inserted. For each virtual function in the program, ρFEM uses the information from step one to determine the class it belongs to. Then, it takes each vtable of this class and looks up the `ID`s assigned to the vtable/function combinations. Note that ρFEM might find multiple `ID`s, either, because there can be multiple vtables for a class or because with virtual inheritance there can be multiple `ID`s for a single vtable. It is interesting to note that in the end the total number of `ID`s is independent of whether virtual or non-virtual inheritance was used, since the number of edges in the vtable hierarchy is independent of the inheritance type. All `ID`s are unique `ID`s for this function, because the `ID`s were assigned for function/vtable combinations, and a function cannot be defined twice in a vtable.

Next, the actual check is generated during LTO. The check works as follows: it takes the callee's return address from the stack and tries to load the range data from this address. The `NOP` instructions containing this data are inserted in step four (next step). ρFEM then checks whether or not one of the callee `ID`s is inside of the fetched range. If this is the case, then the check passes. Otherwise, the return address is not an address after a valid callsite for this particular calltarget and program execution is interrupted.

9.3.3.4 Step 4: Attaching callsite metadata

In the Clang front-end each callsite was annotated with its base class and the function implementation of the base class which can be called by the callsite. As explained previously, a callsite can only call functions in the sub-graph rooted at the callsite's base class. This principle is reflected by the range, which is obtained through the base class/function combination. This holds, because the range contains only the `ID`s assigned to the function implementations in the base class or in any of its children classes, which were explored by the pre-order traversal in step two.

Therefore, annotating each callsite with the range assigned to the base class enforces a strict set of valid function targets. Any function implementation dispatched at this callsite has at least one `ID` inside this range, and therefore the callsite passes the check. Any other function implementation cannot have an `ID` inside the range, since the ranges are closed (compact) ranges (as explained in step two).

Finally, one last optimization is performed in Steps three and four. Rather than storing the start and end `ID` of a range, the start `ID` and the width of the range are stored. This reduces the amount of operations required for the check and thus the runtime overhead.

9.3.4 Function Pointer Based Call Analysis

Each callee, of which the address escapes to memory (address taken (AT)) can potentially be called by a function pointer. Since precise control flow analysis is generally impossible due to the fact that alias analysis is undecidable [229], we assume, that it is valid, for each function pointer based callsite to call a callee, as long as the function signatures match. As such, ρFEM implements a function signature encoding, allowing it to encode function signature data in `ID`s. The function signature consists of: (1) the name of the caller and of the callee, (2) the number of parameters the caller provides and the number of parameters the callee consumes (as for now, the first eight function parameters are taken into consideration), (3) their parameter types (as for now, 26 LLVM IR parameter types, *i.e.,* HalfTyID, FloatTyID, VoidTyID, *etc.* are taken into consideration), and (4) the callee return type.

For an AT function, ρFEM generates a function-signature-`ID` by using the previously mentioned function signature encoding algorithm. Using the same encoding, ρFEM annotates each function pointer based callsite with such an `ID`. The callee accepts both the `ID`(s) generated in Section 9.3.2 or Section 9.3.3 alongside with the function-signature-`ID` in case it was called indirectly, but only if its own function signature was used at the callsite. Note that the regular `ID` is the `ID` which was assigned through the pre-order traversal. Finally, during runtime, in case the caller signature matches the callee signature, the control flow is allowed to return. Otherwise, the control flow transfer is stopped.

9.3.5 Backward-Edge Checks

```
1 ...
2 X *x=new W();
3 int t = x->foo();
4 ...
```

(a)

```
1 int foo(){
2 ...
3 return x;
4 }
```

(b)

```
1 ...
2 0x400d60 add $0x10,%rax
3 0x400d64 mov (%rax),%rcx
4 0x400d67 mov %rax,%rdi
5 0x400d6a callq *0x8(%rcx)
6 ...
```

(c)

```
1 ...
2 0x400cef pop %rbp
3 0x400cf0 retq
4 ...
```

(d)

```
1 ...
2 0x400d60 add  $0x10,%rax
3 0x400d64 mov (%rax),%rcx
4 0x400d67 mov %rax,%rdi
5 0x400d6a callq *0x8(%rcx)
6 0x400d6d nopl 0x8000a(%rax)
7 0x400d74 nopl 0x80001(%rax)
8 ...
```

(e)

```
1 ...
2 0x400cce mov     0x8(%rbp),%rcx
3 0x400cd2 mov     0x3(%rcx),%eax
4 0x400cd5 mov     $0x8000a,%edx
5 0x400cda sub     %eax,%edx
6 0x400cdc cmp     0xa(%rcx),%edx
7 0x400cdf jbe     0x400cf1
8 0x400ce1 cmp     $0x2000000,%rcx
9 0x400ce8 ja      0x400cf1
10 0x400cea cmp    $0x7fffe,%eax
11 0x400cef jne    0x400cf3
12 0x400cf1 pop    %rbp
13 0x400cf2 retq
14 0x400cf3 ud2
```

(f)

Figure 9.4: Caller (a) and callee (b) source code and caller (c) without instrumentation. Callee assembly (d) and caller (e) and callee (f) assembly instrumentation.

Figure 9.4 depicts the instrumentation added by ρFEM to a caller and its corresponding callee in order to protect against backward-edge control flow violations.

Range based checks. The code listings depicted in Figure 9.4(a) and Figure 9.4(b) show the original source code, while the code listings depicted in Figure 9.4(c) and Figure 9.4(d) show the resulting assembly instructions without applying any backward-edge checks. Lines 2 – 4 in Figure 9.4(c) execute the virtual dispatch using an object *X* stored in the `rax` register. Before the callee returns, as depicted in Figure 9.4(d), the stack has to be popped once to clean up the stack frame.

Next, we analyze the actual checks shown in the last row contained in Figure 9.3(e) The newly-added `NOP` instructions are present in lines 6 and 7 in Figure 9.4(e) and contain a range starting at `0x0a` (`StartID`) with a width of `0x01` (`WidthOfRange`), *i.e.*, the only callees valid at this callsite have `ID 0x0a` or `0x0b`. When looking at the callee's instructions depicted in Figure 9.4(f), we observe that its `ID` is located on line four. As expected, it has one of the `IDs` inside the range, namely `0x0a`. We can also see the range in the two instructions before (line two and three): the start `ID` is loaded from the first `NOP` and then the callee `ID` is substracted from it (`StartID`−`0x0a`). In case everything went through up to this point, the result of the subtraction should now be in the range from 0 to `WidthOfRange`, which is checked with the help of the `cmp` and `jbe` instructions located on line numbers 6 and 7 in Figure 9.4(f).

Signature-based checks. Similarly, indirect calls (*i.e.*, function pointer based), which have a matching function signature encoding (*e.g.*, the `cmp` with the address `0x7fffe` on line 10 in Figure 9.4(f)), also pass the check and execution continues.

Error handling. In case none of the checks succeed, the program executes the `ud2` instruction depicted on line number 14 in Figure 9.4(f), causing the program to terminate. While this type of mitigation is sufficient for our purposes, in real-world applications more sophisticated error handling might be used. Instead of abruptly terminating the program another possible approach is to log each legal and illegal backward-edge transfer.

External calls. If a protected function is called by an unlabeled callsite (*i.e.,* external library call), then this call causes the protected function to return at the next address after the call instruction with the help of the instructions located on line 7 and 8 in Figure 9.4(f). In our example, we used the dummy value of `0x2000000` to differentiate between external and internal calls. Note that ρFEM is capable of differentiating between these types of calls by determining during compile time the address range of the protected application. As such, external calls have a memory address not contained in the range of the protected application and the inserted check can differentiate between internal and external calls.

9.3.6 Implementation

We implemented ρFEM as three link-time optimization (LTO) passes and a machine instruction-level pass by extending the Clang/LLVM (v.3.7.0) compiler [50] framework infrastructure. As three of the four ρFEM passes are performed during link time, our system requires LLVM's LTO. The implementation of ρFEM is split between the Clang compiler front-end (metadata collection), three new link-time passes and one machine-level pass used for analysis and generating backward-edge constraints, totaling around 3 KLOC. ρFEM supports separate compilation by relying on the LTO mechanism built in LLVM [50]. ρFEM generates unique `IDs` by keeping track of already assigned ones and continuously incrementing a counter variable for generating new `IDs`. Further, by carefully traversing each class hierarchy in a pre-order traversal, unique `ID` assignment is guaranteed.

9.4 Evaluation

In this Section, we focus on showing the relevance and security benefits of ρFEM. Thus, we address the following research questions (RQs).

- **RQ1:** How **effective** is ρFEM in protecting backward edges? (Section 9.4.1)
- **RQ2:** Which types of backward-edge based **attacks** can be defended when using ρFEM? (Section 9.4.2)
- **RQ3:** What **security benefit** does ρFEM offer when it makes backward edge ROP gadgets not usable anymore? (Section 9.4.3)
- **RQ4:** What is the **runtime overhead** of ρFEM? (Section 9.4.4)

Test Programs. In our evaluation, we used the following real-world C/C++ programs: (1) Memcached [163] (v. 1.5.3), (2) Nginx [204] (v. 1.13.7), (3) Lighttpd [144] (v. 1.4.48), (4) Redis [231] (v. 4.0.2), (5) Apache Httpd Server (Httpd) [7] (2.4.29) and the following C++

programs: (6) NodeJS [213] (v. 8.9.1), (7) Apache Traffic Server [8] (v. 2.4.29), and (8) Google Chrome [97] (v. 33.01750.112). We selected these programs because of their security relevance and real-world usage.

Experimental Setup. The benchmarks were performed on an Intel i5-3470 CPU with 8GB of RAM running on the Linux Mint 18.3 Sylvia OS. We compiled each program ten times in order to provide reliable mean values. If not otherwise stated, all programs were compiled with the `-O2` compiler optimization flag.

9.4.1 Protection Effectiveness

In this Section, we first evaluate the effectiveness of calltarget set reduction of ρFEM with first the NodeJS program and then we present general results for multiple real-world programs.

9.4.1.1 Case Study: NodeJS

We assessed the effectiveness of return target reduction compared to allowing all possible addresses in the binary by presenting the results obtained in a NodeJS based case study. We selected NodeJS because it is a `C/C++` program that is frequently used in practice. To achieve better coverage of the NodeJS code base, the binary was statically linked. Therefore it includes many functions from other libraries such as: OpenSSL, and Google's V8 engine.

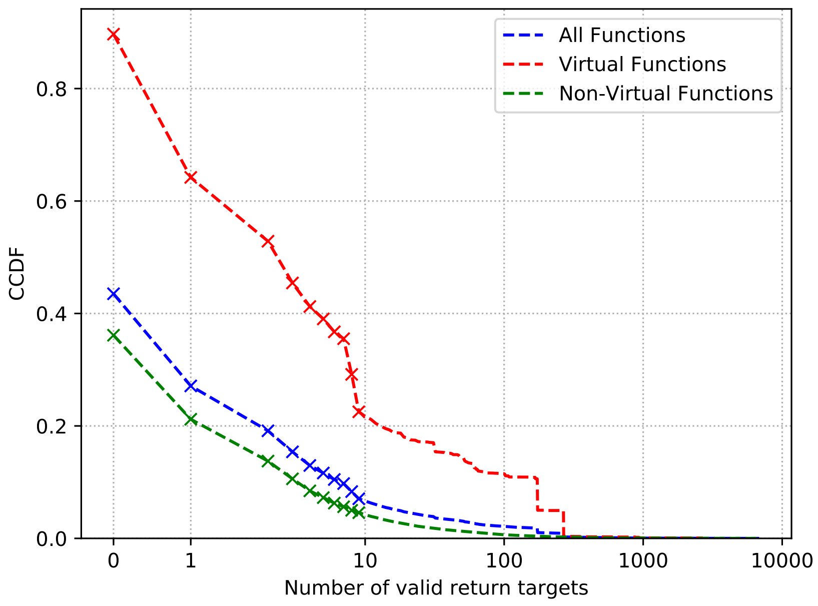


Figure 9.5: CCDF between the number of return targets and the percentage of legitimate callees.

Figure 9.5 depicts the complementary cumulative distribution function (CCDF, we used a *greater than* comparison). The X-axis represents the number of return targets (by number), that a callee can target, while the Y-axis represents the percentage of callees, that allow more than the given number of return targets. Note that for this plot, we assume that no pointer based function calls are allowed to call virtual functions. We consider a callee to be no longer sufficiently protected when it is allowed to return to more than ten targets, this value was arbitrarily picked and can be adjusted.

The blue line shows the distribution when taking all functions into account. We can observe that around 7% of the functions allow to target more than ten return addresses. The red line only takes virtual callees into account. Around 20% of these callees allow more than ten return targets. This is due to the range-based approach for protecting virtual callsites. Since most virtual functions have their address taken, many of the return targets stem from function pointer based callsites. The green line shows the results for non-virtual functions, which are the majority of functions contained in NodeJS (more than 85%). Therefore, the results of the green line are similar to those of the blue line, with around 5% of the functions allowing more than ten return targets.

9.4.1.2 Generalized Results

We assessed the precision of ρFEM by counting the number of allowed return targets per function (callee). We consider the size of this return target set an indicator for the precision of ρFEM's backwards-edge protection.

				Statistics				
Program	**# Callees**	**Min**	**90p**	**Max**	**Geo.**	**Median**	**Avg**	**SD**
Httpd	1,086	0	20	187	3.34	3	8.92	18.18
Lighttpd	451	0	6	317	1.97	1	4.78	19.46
Memcached	106	0	8.5	136	2.81	2	6.07	15.15
Nginx	1,132	0	29	1,630	3.29	2	11.57	58.23
Redis	2,644	0	7	3,796	1.97	1	7.89	81.74
NodeJS	30,330	0	231	6,837	3.34	1	34.70	114.11
Tr. Server	6,115	0	14	2,673	3.13	2	14.80	64.04
average	5,980.57	0	45.07	2,225.14	2.83	1.71	12.68	52.99
geomean	1,616.32	0	18.23	986.87	2.77	1.57	10.28	40.74

Table 9.1: Number of allowed return addresses per callee. All virtual and non-virtual functions are considered as callees.

Table 9.1 depicts the total number of functions hardened by ρFEM and the size of their legitimate return address set enforced by ρFEM. Table 9.1 contains the following entries: # Callees (total number of callees), minimum, 90th percentile, maximum, average, geomean, median, and standard deviation. The geometric mean values for all assessed programs are less than 3.5 return addresses per callee. This considerably decreases the chances of a successful attack. The average value obtained for NodeJS (most complex program in Table 9.1) represents an outlier. This value originates from the high number of small helper functions which are not in-lined and a large number of function pointer based indirect callsites which are pulling down the metric. We note that these worst case functions with over a few hundred return targets are not sufficiently protected. We investigated the worst cases in NodeJS and observed many indirect callsites calling template functions, which were generated for a myriad of JavaScript types. These functions all have the same signature and can therefore be targeted by many indirect callsites.

				Statistics				
Program	**# Callees**	**Min**	**90p**	**Max**	**Geo.**	**Median**	**Avg**	**SD**
NodeJS	4,177	0	239	2,792	23.12	19	92.31	143.31
Tr. Server	948	0	25	992	7.54	11	20.52	59.8
Chrome	66,032	0	1,150	15,014	144.21	155	1,399.97	3,677.36
average	23,719	0	471	6,266	58.29	61.66	504.26	1,293.49
geomean	6,394.53	0	190.11	3,464.50	29.29	31.87	138.41	315.86

Table 9.2: Return addresses allowed by ρFEM for several C/C++ programs. Note that only virtual functions are considered here as callees.

Table 9.2 depicts the sizes of the legitimate return targets for only virtual functions. By comparing the results in Table 9.2 and in Table 9.1 against each other we note that in general ρFEM performs not as good for virtual functions than for non-virtual functions. This is due to the fact that ρFEM: (1) uses ranges for virtual functions instead of single `IDs` (ranges contain more than one element), (2) can not precisely determine when a virtual function is called through a function pointer based call (due to the currently used address taken analysis).

				Statistics				
Program	**Baseline**	**Min ‰**	**90p ‰**	**Max ‰**	**Geo. ‰**	**Med. ‰**	**Avg ‰**	**SD ‰**
Lighttpd	52,060	0	0.38	3.59	0.06	0.06	0.17	0.35
Memcached	24,672	0	0.34	5.51	0.11	0.08	0.25	0.61
Nginx	173,273	0	0.17	9.41	0.02	0.01	0.07	0.34
Redis	333,835	0	0.02	11.37	0.01	0.00	0.02	0.24
NodeJS	2,479,736	0	0.09	2.76	0.00	0.00	0.01	0.05
average	612,715.2	0	0.2	6.53	0.04	0.03	0.10	0.32
geomean	179,091.65	0	0.13	5.67	0.02	0.01	0.06	0.24

Table 9.3: Fraction of instructions allowed to return to.

Table 9.3 depicts the fractions of instructions a callee can return to. The results denote the fraction (in ‰per thousand) of return targets allowed per callee. The *Baseline* entry denotes the number of assembly instructions (addresses) in the program binary code section(s). Note that without any backward edge protection a return instruction can freely transfer control flow to any of the *Baseline* addresses. The results depicted in Table 9.3 are important since these show the fraction of legitimate addresses which are allowed to be called after we hardened the binary with ρFEM. The results in second column up to the last (from left to right) were determined by dividing the results from Table 9.1 with the total number of *Baseline* instructions depicted in column 2 (*Baseline*) of Table 9.3. The results indicate that the fraction of addresses that are targetable after applying ρFEM for every analyzed program, is less than one in a thousand addresses on average targetable.

9.4.2 Exploit Coverage

We created a suite of C/C++ programs demonstrating various possible scenarios of callee return target address overwrites.

Exploit	Stopped	Remark
Active-Set Attacks [260, 258]:		
Type 1 (**T1**)	✓	Return to any stack func.
Type 2 (**T2**)	✓	Return to a child process
Type 3 (**T3**)	✓	Return to earlier callsites
Type 4 (**T4**)	✓	Return to future callsites
Type 5 (**T5**)	✓	Return to program begin
CALL-ret violating [242]:		
Innocent flesh on the bone	✓	Caller-callee function calling conv. violation
CALL-ret non-violating [238]:		
COOP (forward edge)	✗	Out of scope
ret equivalent:		
`pop jump` instr. combination	✗	Out of scope

Table 9.4: Stopped backward-edge attacks.

Table 9.4 presents a summary of several types of backward-edge based attacks and the primitives on which these rely. For each of Types one to five (**T1** - **T5**), our suite contains at least one program reflecting this behavior. Next, we present the backward-edge primitives on which the attacks depicted in Table 10.5 rely. Note that **T1** up to **T5** can all be independently used to bypass [260, 258] the HAFIX [65] backward-edge protection technique. **T1.** Return to any active function on the stack (not just the last function put on the call stack). **T2.** Return to parent code in a child process after a fork. **T3.** Return to earlier callsites in functions on the stack. **T4.** Return directly to future callsites in functions on the stack. **T5.** Return directly to the beginning of a program (typically the second callsite in main).

Type	Full CFL	ρFEM
T1	✗	✓
T2	✗	✓
T3	✗	✓
T4	✗	✓
T5	✗	✓

Table 9.5: Exploits caught by callee return target protection techniques.

Table 9.5 depicts the results of running the programs from our suite with ρFEM. We could not evaluate these with the CFL tool as this is not open source. Based on the analysis performed by the CFL [19] tool, we expect that CFL cannot detect any of these five types of backward-edge violations as it allows a callee to return to any address following an indirect or direct function call.

Next, we explain how ρFEM can mitigate these attacks. **T1.** None of the addresses enforced by ρFEM is a function start address, only legitimate function return addresses are enforced. **T2.** In case the return address is not in the legal return target set of the particular return, then this is forbidden. **T3.** Callsite addresses are rejected by ρFEM completely, allowed addresses are only these which are following a callsite. **T4.** Future callsite addresses are not included by ρFEM in the target address set and as such these are forbidden. **T5.** Callsite addresses are completely forbidden by ρFEM. Finally, note, that ρFEM can stop the Galileo ROP attack [242] due to the fact that the callee can return to any program address and this is forbidden when ρFEM is used.

9.4.3 Security Analysis

In this Section, we evaluate the availability of gadgets after using ρFEM. Assuming that the initial backward edge is protected by ρFEM, three conditions have to be met to make a gadget usable in a ROP chain: (1) the gadget has to start with a NOP instruction (in order to be targetable from a secured backward edge), (2) the payload of the NOP instruction has to pass the backward-edge check of the incoming backward edge, and (3) the return instruction at the end of the gadget has to be either unprotected or its target has to be contained in the return target set of the function the gadget is part of. Note that condition (2) has already been extensively discussed in RQ1 in a generalized form. Assuming the return target set of the gadget is not sufficient to chain to the next gadget (as shown in RQ1), then condition (3) only holds if the backward edge is unprotected.

	Gadget Statistics					
Program	**# LTO**	**# TR**	**#no-NOP**	**#ret check**	**#not pro-tect**	**%not pro-tect**
Httpd	12,664	19,723	19,430	11,033	77	0.39%
Lighttpd	5,855	7,309	7,154	1,100	132	1.81%
Nginx	15,789	20,392	20,212	8,475	128	0.63%
Memcached	1,805	2,056	2,007	184	43	2.09%
NodeJS	375,032	490,570	485,396	99,853	3,222	0.66%
Tr. Server	75,187	106,766	104,935	23,486	1,275	1.19%
average						1.13%
geomean						0.95%

Table 9.6: `ret` gadgets available before/after hardening.

Table 9.6 depicts the results obtained after analyzing the hardened and unhardened program binaries using the open source ROPgadget [233] tool (ROP gadget finding tool). We passed the following arguments to ROPgadget: `-depth=30 -nosys -nojop`. In this example, we consider the Table 9.6 columns numbered from left to right, in total we have seven columns.

The second column of (# gadgets with only LTO) contained in Table 9.6 shows the number of unique gadgets found in the unhardened binary with LTO enabled. Here, a gadget is considered unique if it consists of a unique sequence of instructions.

The third column (# gadgets ρFEM) contained in Table 9.6 presents the number of unique gadgets in the hardened binaries. In this case, for a gadget to be unique, we also consider the existence of a `NOP` instruction at the beginning of the gadget. This means that two gadgets with the same sequence of instructions, one with `NOP` before these instructions and one without, are different gadgets. This differentiation is important since for an attack the existence of the leading `NOP` is relevant. Due to this change the number of gadgets found by ROPgadget [233] increases compared to the unhardened binary.

The fourth column (# gadgets no-`NOP` miss) contained in Table 9.6 shows the number of gadgets in the hardened binary that fail condition (1), *i.e.,* these gadgets do not start with a `NOP` instruction and are therefore not targetable by a secured backward edge. We can observe that most gadgets do not start with a `NOP` instruction as expected, since the number of callsites (and therefore of `NOP` instructions) is small compared to the total number of instructions.

The gadgets depicted in column five (# gadgets ret-check) contained in Table 9.6 end with a return check generated by ρFEM and therefore can only target gadgets that match the return target set of the function the gadget is contained in. Note, that the return instructions not protected by ρFEM are mostly either boilerplate functions (*e.g.,* global setup and init functions) or small stack adjustment functions generated by LLVM. These functions rarely contain any calculating instructions (*e.g.,* `ADD`, `SUB`) with non-imminent operands.

The sixth column (# unprotected gadgets) contained in Table 9.6 shows the number of gadgets for which both condition (1) and (2) hold. These gadgets are not counted in column two or three and therefore have a `NOP` instruction at the beginning, but no check of the backward edge. These are the remaining gadgets that can be used for a ROP chain attack without exploitation of the residual return target set evaluated in RQ1.

Finally, the seventh column (% unprotected gadgets) contained in Table 9.6 depicts the percentage of all gadgets in the hardened binary which are unprotected after hardening the program binary with ρFEM. We can observe that on average only 1.13% of the gadgets found are still useful after hardening a binary with ρFEM. Thus, ρFEM protects on average 98.87% of the identified gadgets.

9.4.4 Runtime Overhead

Figure 9.6 shows the runtime overhead of SafeStack (shaded light gray), ρFEM (shaded black), and ρFEM +IVT (shaded gray) on popular Chrome web browser benchmarks. Thus, ρFEM can be applied incrementally when used together with IVT, offering forward- and backward-edge protection. The geomean overhead, when using only ρFEM and ρFEM +IVT, is 2.72% and 2.78%, respectively.

Table 9.7 depicts the overall runtime overhead when running ρFEM on all pure C/C++ programs contained in the SPEC CPU2017 (rate) benchmark. Note that, we could not compile `502.gcc_r` with SafeStack, whereas it was possible with ρFEM. For this reason, we did not include it in Table 9.7. Finally, note that the average performance overhead of ρFEM is 0.63%, which we consider a good outcome.

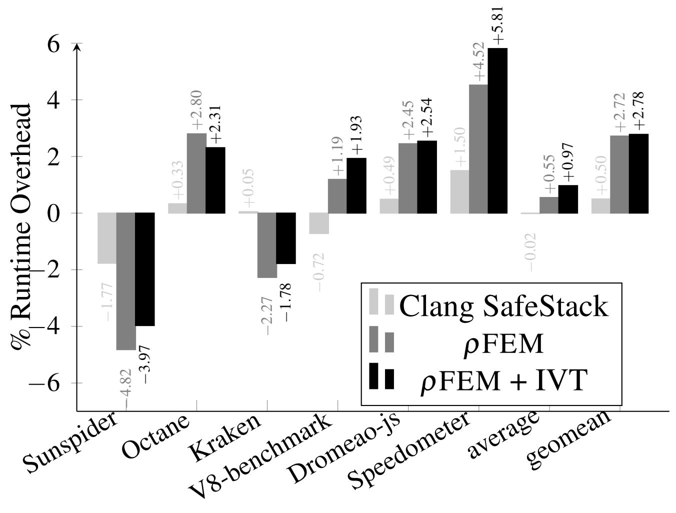


Figure 9.6: ρFEM vs. SafeStack vs. IVT overhead

Benchmark	LTO	SafeStack + LTO	SafeStack + LTO %	ρFEM + LTO	ρFEM + LTO %
500.perlbench_r	11.0	11.1	-0.91	11.1	-0.91%
505.mcf_r	11.7	11.4	2.56%	11.7	0.00%
520.omnetpp_r	8.05	8.12	-0.87%	8.02	0.37%
523.xalancbmk_r	8.79	8.79	0.00%	8.61	2.05%
525.x264_r	18.6	19.4	-4.30%	18.5	0.54%
531.deepsjeng_r	13.6	14.0	-2.94%	13.4	1.47%
541.leela_r	12.7	12.8	-0.79%	12.6	0.79%
557.xz_r	8.73	9.13	-4.58%	8.74	-0.11%
508.namd_r	12.8	13.0	-1.56%	12.7	0.78%
511.povray_r	17.7	17.6	0.56%	16.9	4.52%
519.lbm_r	4.82	4.82	0.00%	4.82	0.00%
526.blender_r	17.1	17.0	0.58%	17.1	0.00%
538.imagick_r	17.6	19.1	-8.52%	17.7	-0.57%
average	-	-	-1.33%	-	0.63%
geomean	-	-	0.31%	-	0.03%

Table 9.7: Overhead on the SPEC CPU2017 (rate) benchmark.

Table 9.8 captures the results of running ρFEM on several programs contained in the SPEC CPU2017 speed benchmark and comparing the results with SafeStack. Again, note that we could not compile `602.gcc_s` with SafeStack, whereas it was possible with ρFEM. We removed it from Table 9.8. The average overhead of ρFEM is 0.23% and compares favorably to that of SafeStack with an average of 0.70%.

Benchmark	LTO	SafeStack + LTO	SafeStack + LTO %	ρFEM + LTO	ρFEM + LTO %
600.perlbench_s	3.51	3.46	1.42 %	3.55	-1.14 %
605.mcf_s	6.69	6.31	5.68 %	6.67	0.30 %
620.omnetpp_s	3.81	3.7	2.89 %	3.66	3.94 %
623.xalancbmk_s	3.73	3.65	2.14 %	3.55	4.83 %
625.x264_s	5.15	5.2	-0.97 %	5.09	1.17 %
631.deepsjeng_s	4.06	4.02	0.99 %	4	1.48 %
641.leela_s	3.62	3.54	2.21 %	3.62	0.00 %
657.xz_s	2.06	2.24	-8.74 %	2.24	-8.74 %
average	-	-	0.70%	-	0.23%
geomean	-	-	2.34%	-	0.42%

Table 9.8: Overhead on the SPEC CPU2017 (speed) benchmark.

9.5 Discussion

In this Section, we discuss attack trade-offs and limitations of ρFEM.

9.5.0.1 Attack Trade-offs

Size of return sets trade-offs. Compared to Clang's SafeStack (based on LLVM v.3.7.0 stable) the sizes of the return sets of the assessed programs of ρFEM do not ensure a one-to-one mapping. As it can be observed in Table 9.1, the sizes of the return sets are 2.83 in geomean. The security of the program depends on the available gadgets, which are located at the addresses directly after the addresses contained in the per function target set. In our opinion, the availability of these gadgets has to be further evaluated, in order to tell if an attack is still possible under these conditions.

Attacker accessibility trade-offs. As the sizes of the return sets are larger than one, in some situations (even through the return site), these could still be used by an attacker to return to a legitimate address, allowing access to useful gadgets. The accessibility of the gadgets depends on the following. The larger the return set per callee is, the higher are the chances, in general, for an attacker to be able to perform an attack.

Shadow stacks trade-offs. We note that the legitimate return set for a callee protected by a shadow stack technique is always 1 and as such the chances to successfully perform an attack are lower compared to using ρFEM. Further, we cannot rule out that an attacker may still be able to perform an attack when the binary is protected by a shadow stack technique, in case the attacker knows: (1) the legitimate return address of a callee, and (2) the return address which can be used to access a useful gadget.

9.5.0.2 Limitations

Labeling of legitimate return sites. ρFEM inserts labels with IDs that correspond to each legitimate return site for each function return (callee). This means that the legitimate addresses of each function return are marked with the `NOP` instructions containing the same ID. This could provide an attacker an useful hint at which addresses it is legitimate to return. In case these return addresses contain useful gadgets than the attacker may return at this addresses, thus bypassing the ρFEM CFI checks. This limitation can be addresses by adding an additional level of indirection at the location when the `NOPs` are inserted. In this way it should be made hard for the attacker to determine which IDs are valid at that particular return site. Thus, the labeling of the legitimate return sites is avoided and the attacker would not be able to determine the legitimate return set for a particular function return and as such the bypassing of the backward edge CFI checks would be made much harder for the attacker.

Control flow bending. Control-Flow Bending (CFB) [36] has shown that, even with fully-precise static CFI, powerful CRAs are still possible. CFB demonstrates that without shadow stack, perfect static CFI can be bypassed through the usage of *dispatcher functions*, which have a large number of legitimate callers, such as the `printf()` function. Taking the dispatcher function as a pivot, attackers can return to any legitimate callers. The common way to mitigate CFB attacks is to use a shadow stack, where only one return target is allowed. ρFEM cannot handle CFB attacks with the same precision as shadow stack techniques. Note that all other techniques (excluding shadow stacks) cannot protect against CFB as well. On one hand, only shadow stack can mitigate this type of attack but these shadow stacks can be bypassed [94]. On the other hand, the goal of ρFEM is not to be able mitigate CFB attacks. Further, note that against the common believe that only shadow stacks can handle CFB attacks, we think that in case the return target set per callee is small (say under five targets as in our case in geomean) the likelyhood of finding usable gadgets in this set when performing CFB are low. As such ρFEM can protect against CFB only to a certain degree.

No inter-modular support. ρFEM can only secure single binaries. This means that for dynamic libraries each library hast to be compiled in order to be protected. In the current ρFEM implementation, the `IDs` for different modules may overlap which increases the return target set. In addition, inter-modular backward edges are not protected. This situation can be addressed by synchronizing IDs between modules. The program modules are compiled after compiling dynamic libraries. This allows for forward sharing of `ID` information in the modules which use the dynamic libraries. This is an engineering limitation which is easily solvable.

Imprecise function pointer callsite analysis. In the Evaluation (Section 10.4) we show that function pointer based callsites account for a significant amount of return targets. This is especially problematic for virtual callees since these usually are not targets of function pointer based callsites. This issue can be addressed by developing a better address taken (AT) function detection which helps to reduce the number of functions that can be targeted by any function pointer based callsite. In addition, the function signature encoding can be improved by taking into account more types and the (*e.g.,* `this`) pointer.

9.6 Summary

Return based program indirect control flow transfers play an important role when performing nowadays software-based code reuse attacks (CRAs) as presented in recent attacks against Google's Chrome and Mozilla Firefox Web browsers. The current solutions which protect against backward-edge based return violations enforce the caller/callee calling convention mostly based on shadow stack techniques. Unfortunately, shadow stack techniques are highly vulnerable when attackers are using one of the existing weaknesses: (1) neglected pointers, (2) thread spraying, (3) allocation oracles, and (4) entropy based attacks.

In this Chapter, we presented ρFEM, an Clang/LLVM based backward edge runtime protection tool that leverages static forward edge information to harden backward edges in C/C++ programs. For forward edges belonging to a virtual call this information is obtained using (1) class hierarchy analysis, while for forward edges corresponding to a function pointer based call (2) a function signature matching approach is used. The information from (1) and (2) is used to create backward edge mappings which are then utilized to build backward edge runtime constraints. We conducted an evaluation of ρFEM with several real-world programs: Google Chrome, NodeJS, Nginx, *etc.* Our evaluation shows that only a low median number of return targets per callee return site are allowed. The median number of return addresses per callee is small enough (up to 3 return addresses) while the geomean is 2,83. This considerably reduces the chances of successfully performing control flow bending attacks by violating backward-edge transfers. The max value of return targets is quite high, as such these locations can be flagged during the analysis and require some orthogonal protection, *i.e.,* push towards future work. The largest class of targets for a calle is 15,014 measured for the Chrome Web browser depicted in Table 9.2. Further, we want to address these cases in our future work. As a consequence the programmer has a clear indication where he can improve the approach.

Our evaluation results show that ρFEM can be used as an always-on runtime mitigation technique, thus proving its practicability in real scenarios. Our experiments with the Google's Chrome web browser suggest that ρFEM imposes a low average runtime overhead of 2.77% geomean. Consequently, ρFEM is compatible with current large C/C++ applications such as Chromium, readily deployable, and advances the state-of-the-art for the protection of callee return targets. Finally note that, ρFEM does not suffer from the attack vectors used for bypassing information hiding in general.

Chapter 10

τCFI: Runtime Protection of Program Binaries Against Code Reuse Attacks

In this Chapter, which belongs to the second part of this thesis, we developed a framework, called τCFI, for protecting of legacy program binaries with new CFI policies designed by taking into account the lessons learned in Chapter 8. These learned lessons helped us to thoroughly address **RQ7** in order to provide a tool which can effectively protect forward and backward program CFG edges in stripped binaries (*i.e.,* most of the semantic information has vanished through the compilation process as this type of information is usually not required in production-ready binaries). CRAs which rely on corrupting forward and/or backward CFG edges due to indirect control flow transfer violations are mitigated by greatly reducing the likelihood of successfully performing such an attack when hardening the program binary with τCFI. The successful CRA likelihood is reduced by allowing indirect backward edges transfers to return to only legitimate targets. This greatly reduced the attacker wiggle room. Finally, note that parts of this Chapter have already been published by Muntean *et al.* [195].

10.1 Introduction

What is the overall view? The C++ programming language has been extensively used to build many large, complex, and efficient software systems over the last decades. A key concept of the C++ language is polymorphism. This concept is based on C++ virtual functions. Virtual functions enable late binding and allow programmers to overwrite a virtual function of the base-class with their own implementation. In order to implement virtual functions, the compiler needs to generate virtual table meta-data structures for all virtual functions and provide to each instance (object) of such a class a (virtual) pointer (the value of which is computed during runtime) to the aforementioned table. Unfortunately, this approach represents a main source for exploitable program indirection (*i.e.,* forward edges) along function returns (*i.e.,* backward edges), as the C/C++ language provides no intrinsic security guarantees (*i.e.,* we consider Clang-CFI [149] and Clang's SafeStack [49] optional).

What is the problem? In this Chapter, we present a new control flow integrity (CFI) tool called τCFI used to secure `C++` binaries by considering the type information from application binaries. Our work targets applications, whose source code is unavailable and that contain at least one exploitable memory corruption bug (*e.g.,* a buffer overflow bug). We assume such bugs can be used to enable the execution of sophisticated Code-Reuse Attacks (CRAs) such as the COOP attack [238] and its extensions [59, 143, 22, 137], violating the program's intended control flow graph (CFG) through forward edges in the CFG and/or through attacks, that violate backward edges such as Control Jujutsu [80]. A potential prerequisite for violating forward-edge control flow transfers is the corruption of an object's virtual pointer. In contrast, backward edges can be corrupted by loading fake return addresses on the stack.

What are the existing solutions? To address such object dispatch corruptions, and in general any type of indirect program control flow transfer violations, CFI [1, 2] was originally developed to secure indirect control flow transfers, by adding runtime checks before forward-edge and backward-edge control transfers. CFI-based techniques that rely on the construction of a precise CFG are effective [32] if CFGs are carefully constructed and sound [257]. However, these techniques still allow CRAs that do not violate the enforced CFG. For example, the COOP family of CRAs bypasses most deployed CFI-based enforcement policies, since these attacks do not exploit indirect backward edges (*i.e.,* function returns), but rather imprecision in forward edges (*i.e.,* object dispatches, indirect control flow transfers), which in general cannot be statically (before runtime) and precisely determined as alias analysis in program binaries is undecidable [229]. Source code based tools such as: SafeDispatch [124], MCFI [210, 211], ShrinkWrap [106], VTI [24], and IFCC/VTV [261] can protect against forward-edge violations. However, they rely on available source code, limiting their applicability (*e.g.,* proprietary libraries cannot be recompiled). In contrast, binary-based forward-edge protection tools, including binCFI [289], vfGuard [226], vTint [287], VCI [78], Marx [218] and TypeArmor [268], typically protect only forward edges through a CFI-based policy, and most of the tools assume that a shadow stack [135] technique is used to protect backward edges.

What are the limitations of the solution? Unfortunately, the currently most precise binary-based forward-edge protection tools w.r.t. calltarget reduction, VCI and Marx, suffer from forward-edge imprecision, since both are based on an approximated program class hierarchy obtained through the usage of heuristics and assumptions. TypeArmor enforces a forward-edge policy, which only takes into account the number of parameters of caller-callee pairs without imposing any constraint on the parameters' types. Thus, these forward-edge protection tools are generally too permissive. CFI-based forward-edge protection techniques without backward-edge protection are broken [36], thus these tools assume that a shadow stack protection policy is in place. Unfortunately, shadow stack based techniques (backward-edge protection) were recently bypassed [94] and add, on average, up to 10% runtime overhead [63]. Goktas *et al.* [94] demonstrate that at least four independently usable attack vectors (namely: guessing oracles, neglected pointers, thread spraying, and allocation oracles) exist for thwarting shadow stack techniques (binary and source code based) targeting their entropy based hiding principle and making their usage questionable. τCFI defeats these attack vectors by not relying on program features that are used by these attack vectors and by using a deterministic approach rather then one based on information hiding.

What is our insight? In this Chapter, we present τCFI, which is a fine-grained forward-edge and backward-edge binary-level CFI protection mechanism, that neither relies on shadow stack based techniques to protect backward edges, nor any runtime-type information (RTTI) (*i.e.,* metadata emitted by the compiler, which is most of the time stripped from production binaries). Note that, in general, variable type reconstruction on production binaries is a difficult task, as the required program semantics are mostly removed through compilation.

At a high level, there is a number of analyses τCFI performs in order to achieve its protection objective. In particular, it (1) uses its register width (ABI dependent) as the type of the parameter for each function parameter, (2) when determining whether an indirect call can target a function, it checks whether the call and the target function use the same number of parameters and whether the types (register width) match, (3) based on the provided forward-edge caller-callee mapping, it builds a mapping back from each callee to the legitimate addresses, located next to each caller (*i.e.,* the instruction following the callsite). τCFI's backward-edge policy is based on the observation that backward edges of a program can be efficiently protected, if there is a precise forward-edge mapping available between callers and callees. This is later used during the classification of matching callsites and calltargets, in order to distinguish between valid and invalid function calls, and results in a more precise callee target set for each caller than other state-of-the-art approaches like, for instance, TypeArmor.

Analysis Details. τCFI's analysis is based on an basic block based use-def callees analysis to derive the function prototypes, and a liveness analysis at indirect callsites to approximate callsite signatures. This efficiently leads to a more precise CFG of the binary program in question, which can also be used by other systems in order to gain a more precise CFG on which to enforce other types of CFI-related policies. These analysis results are used to determine a mapping between all callsites and legitimate calltarget sets. Further, this mapping is used in a backward analysis for determining the set of legitimate return addresses for each function return determined by each calltarget. Note that we consider each calltarget to be the start address of a function.

τCFI incorporates an improved forward-edge protection policy, which is based on the insight that if the binary adheres to the standard calling convention (*i.e.,* Itanium ABI) for indirect calls, undefined arguments at the callsite are not used by any callee by design and that based on the passed function parameter types can be approximated by their corresponding register width.

τCFI uses a forward-edge based propagation analysis to determine a minimal set of possible return addresses for calltargets (*i.e.,* function returns), which helps to impose the caller-callee function calling convention with high precision. This policy is based on the observation that if a fine-grained forward-edge policy can be precisely determined between callers and callees, then this mapping can be used backwards from the calltarget to the callsite in order to construct a fine-grained CFI policy, which helps to impose the caller-callee calling convention backwards. Our backward-edge policy represents a fine-grained Safe Stack [135] (recently bypassed [94]) alternative. This attack shows that in general the protection offered by shadow stacks is questionable (at least 4 attack vectors), since it is relatively easy for a motivated attacker to disclose the shadow stack and bypass it.

We have implemented τCFI on top of DynInst [18], which is a binary rewriting framework, that allows program binary instrumentation during loading or runtime. Note that τCFI preserves the original code copy of an executable by instrumenting all code of an executable shadow copy, which is later mapped to the original binary after it was loaded and τCFI's analysis finished. τCFI works with legacy programs and can be used to protect both executables and libraries. τCFI performs per-file analysis; as such each file is protected individually. We have evaluated τCFI with several real-world open source programs (*i.e.,* NodeJS, Lighttpd, MySql, *etc.*), as well as the SPEC CPU2006 benchmarks and demonstrated that our forward-edge policy is more precise than state-of-the-art tools. τCFI is applicable to program binaries for which we assume source code is not available. τCFI significantly reduces the number of valid forward edges compared to previous work and thus, we are able to build a precise backward-edge policy, which represents an efficient alternative to shadow stack based techniques.

What are our contributions? In summary, in this Chapter, we make the following contributions:

- We present τCFI, a new CFI system that improves the state-of-the-art CFI with more precise forward-edge identification by using type information reverse-engineered from stripped x86-64 binaries.
- We implement τCFI as a binary instrumentation framework to enforce a fine-grained forward-edge and backward-edge protection.
- We conduct a thorough evaluation, through which we show that τCFI is more precise and effective than other state-of-the-art techniques.

10.2 Threat Model

We follow the same basic assumptions stated in [268] w.r.t. forward edges protection. More precisely, we assume a resourceful attacker that has read and write access to the data Sections of the attacked program binary. We assume that the protected binary does not contain self-modifying code or any kind of obfuscation. We also consider pages to be either writable or executable, but not both at the same time. Further, we assume that the attacker has the ability to exploit an existing memory corruption in order to hijack the program control flow. As such, we consider a powerful yet realistic adversary model that is consistent with previous work on CRAs and their mitigations [135]. The adversary is aware of the applied defenses and has access to the source and non-hardened binary of the target application. She can exploit (bend) any backward-edge based indirect program transfer and has the capability to make arbitrary memory writes. Further, our technique is orthogonal to other forward-edge and backward-edge protection techniques. As such, we assume that our techniques can be used alongside other existing techniques. Our approach does not rely on information hiding from the attacker and as such we can tolerate arbitrary reads. Finally, the analyzed program binary is not hand-crafted and the compiler which was used to generate the program binary adheres to one of the following most used caller-callee function calling conventions [9, 101, 122].

10.3 Design and Implementation

In this Section, we present the overview of τCFI followed by its design and implementation.

10.3.1 Approach Overview

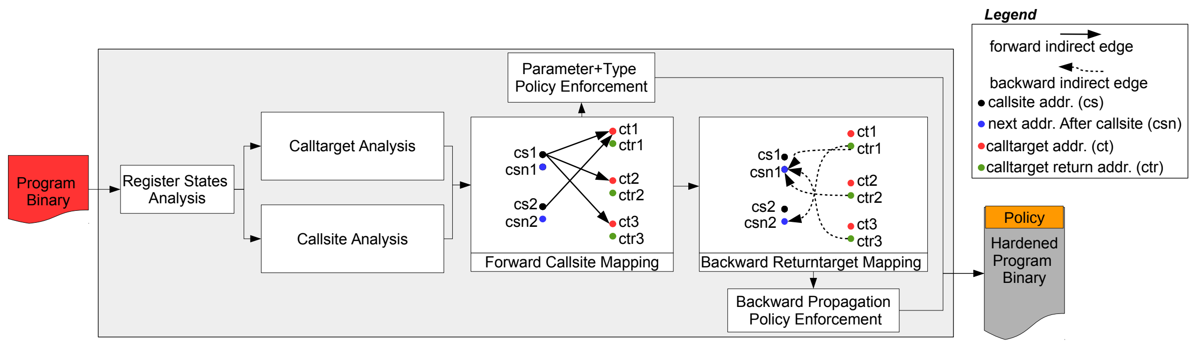


Figure 10.1: Main steps (gray shaded box) done by τCFI when hardening a program binary.

Figure 10.1 depicts an overview of our approach. From left to right, the program binary is analyzed by τCFI and the calltargets and callsite analysis are performed for determining how many parameters are provided, how many are consumed, and their register width. After this step, labels are inserted at each previously identified callsite and at each calltarget. The enforced policy is schematically represented by the black highlighted dots (addresses, *e.g., cs1*) in Figure 10.1 that are allowed to call only legitimate red highlighted dots (addresses, *e.g., ct1*). Next, to compute the set of addresses which a return instruction can target, the address set determined by each address located after each legitimate callsite is computed. This information is obtained by using the previously determined callsite forward-edge mapping to derive a function return backward map that uses return instructions as keys and return targets as values. This way, τCFI has a set of addresses for each function to which the function return site is allowed to transfer control. Finally, range or compare checks are inserted before each function return site. These checks are used during runtime to check if the address, where the function return wants to jump to, is contained in the legitimate set for each particular return site. This is represented in Figure 10.1 by green highlighted dots (addresses *e.g., ctr2*) that are allowed to call only legitimate blue highlighted dots (addresses *e.g., csn1*). Finally, the result is a hardened program binary (see right-hand side in Figure 10.1 fore mode details).

10.3.2 Parameter Count and Type Policy

Parameters can be passed through registers or the stack. In the Itanium `C++` ABI, the first six parameters are passed through registers (*i.e.,* `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`). Even when a 64-bit register is used to pass a parameter, the actual number of bits used in the register might be smaller. Therefore, we treat the used widths of parameter-storing registers as the types of the parameters. There are four types of reading and writing access on registers. Therefore, our set of possible types for parameters is $\{64, 32, 16, 8, 0\}$, where zero models the absence of a parameter. For the Itanium ABI, our analysis tracks the 6 registers used in parameter passing and classifies callsites and calltargets according to how these registers are used.

Our analysis overapproximates at callsites and underapproximates at calltargets the parameter count and types, which is due to the general difficulty of statically determining the exact number of arguments provided by a callsite and the number of parameters required by a calltarget and w.r.t. the widths of registers used in parameter passing. Specifically, at a callsite, the analysis calculates an *upper bound* for the number of arguments and for the widths of those registers that store arguments. For instance, for a function call that passes one argument with a width of 32-bit, the analysis may estimate that there are two arguments passed and the first one's width is 64-bit. Furthermore, the analysis on a calltarget (a callee function) calculates a *lower bound* for the number of needed parameters and for the widths of those registers that store parameters.

Because of the approximations in our analysis, our policy for matching callsites and calltargets allows a callsite to transfer to a calltarget if (1) the number of estimated arguments at a callsite is greater than the number of estimated parameters at a calltarget; and (2) for each argument at the callsite and its corresponding parameter in the calltarget, the estimated width of the argument is greater than the estimated width of the parameter. Part (1) is about the parameter count and is the same as the parameter-count policy in TypeArmor [268]; part (2) is about the parameter types and enables τCFI to provide a finer-grained policy than just considering the parameter count.

10.3.3 Instruction Read-Write Effect

We first introduce some definitions and notation. The set $\mathscr{I}$ describes the set of possible instructions; in our case, this is based on the instruction set for x86-64 processors. An instruction $i \in \mathscr{I}$ can perform two kinds of operations on registers: (1) Read n-bit from a register with $n \in \{64, 32, 16, 8\}$ and (2) Write n-bit to a register with $n \in \{64, 32, 16, 8\}$. Note that there are instructions that can directly access the higher 8 bits of the lower 16 bits of 64-bit registers. For our purpose, we treat this access as a 16-bit access.

Next, the possible effect of an instruction on one register is described as $\delta \in \Delta$ with $\Delta = \{w64, w32, w16, w8, 0\} \times \{r64, r32, r16, r8, 0\}$. Note that 0 represents the absence of either a write or read access and $(0, 0)$ represents the absence of both. Meanwhile, wn with $n \in \{64, 32, 16, 8\}$ implies all wm with $m \in \{64, 32, 16, 8\}$ and $m < n$ (*e.g.,* $w64$ implies $w32$); the same property holds for rn. The Itanium `C++` ABI specifies 16 general purpose integer registers. Therefore, the read-write effect of an instruction on the set of registers can be described as $\delta_p \in \Delta^{16}$. Our analysis performs calculations based on the effect of each instruction $i \in \mathscr{I}$ via the function regEffect : $\mathscr{I} \mapsto \Delta^{16}$. Note that this function can be purely defined based on the semantics of instructions.

10.3.4 Calltarget Analysis

Our calltarget static analysis classifies calltargets according to the parameters they expect by taking into account the parameters' count and types. Given a set of address-taken functions[1], the static analysis performs an interprocedural analysis to determine the register states for the 6 argument registers.

Next, we present τCFI's analysis, followed by a discussion of optimizations and interprocedural analysis. The basic analysis determines, for each register and at a particular program location, that it is in one of the following states:

- rn, where $n \in \{64, 32, 16, 8\}$ represents that the lower n bits of the register are *read before written* along all control flow paths starting from the location.
- $*$ represents that, along some control flow path, the register is either written before read or there are no reads/writes on the register.

The basic analysis described above can be implemented as a classic backward-liveness analysis, except that it needs to track widths in read operations. For instance, for an instruction i, if the regEffect function shows that i reads the lower 16-bits of `rax`, then the state of `rax` immediately before the instruction is $r16$. For an instruction with multiple successors, the register states after the instruction are calculated based on the states at the beginnings of the successors. For instance, if an instruction has two successors, and the state of `rax` is $r64$ before the first successor and the state of `rax` is $r32$ before the second, then the state of `rax` after the instruction is $r32$, essentially indicating that all paths starting from the end of the instruction have a $r32$ read before write for `rax`. Recall that the calltarget analysis performs an underapproximation; so using $r32$ is safe even though one of the paths performs a $r64$ read.

The backward-liveness analysis, however, is inefficient. Our implementation actually follows TypeArmor [268] to perform a forward interprocedural analysis (with some modification to consider widths of read operations). We refer readers to the TypeArmor for details and give only a brief overview in this thesis.

First, note that τCFI's analysis operates at the basic block level instead of the instruction level. For clarity, a basic block is a linear sequence of machine code instructions which does not contain any branch instructions. Second, the analysis further refines the $*$ state to be either w or c, where w (write before read) refers to a register being written to before read from along some control flow path and c (clear/untouched) represents that the register is untouched along some control flow path. The reason for such a refinement is that during forward analysis,

[1]Since an indirect call can target a function only if the function's address is taken, there is no need to analyze functions whose addresses are not taken; this is similar to TypeArmor.

if the states of all argument registers before a basic block *b* are either *rn* or *w* (*e.g.,* when *b* reads or writes all argument registers), then there is no need to keep analyzing the successor basic blocks since their operations would not change the state before *b*; this enables an early termination of the forward analysis and is thus more efficient. On the other hand, if the state of one of the argument registers is *c*, then the forward analysis has to continue. This is because *c* indicates the register is untouched so far, but it can be read or written in a future basic block. Further, the analysis is inter-procedural and maintains a stack to match direct function calls and returns during analysis. Finally, for indirect calls, however, it does not follow to the targets, but performs an under-approximation instead.

10.3.4.1 Parameter count and types

Once the analysis finishes, we can calculate a function's parameter count and parameter types based on the state before the entry basic block of the function. The argument count is determined using the highest argument register that is marked *rn*. The type of an argument register is directly given by the *rn* state of the register.

10.3.5 Callsite Analysis

Our callsite analysis classifies callsites according to the arguments they provide by considering the argument *count* and their *types*. For callsite analysis, overestimations are allowed: the callsite analysis overestimates the number of arguments and the widths of arguments. As such a callsite is allowed to target a calltarget that requires a smaller or equal number of parameters and that requires a smaller or equal width for each parameter.

For callsite analysis, we employ a customized reaching-definition analysis, see Khedker *et al.* [130] for more details. The analysis determines the states of registers. At a particular program location, it determines whether or not a register is in one of the following states:

- *sn*, where $n \in \{8, 16, 32, 64\}$: this represents a state in which the register's lower *n* bit is set in a control-flow path ending at the program location.
- *t* (trashed): this represents a state in which the register is not set on all control flow paths ending at the program location.

τCFI's reaching-definition analysis is implemented as an interprocedural backward analysis similar to TypeArmor [268], the difference being that τCFI also tracks the widths in write operations to infer *sn* states. Once the analysis is finished, it uses the register state just before an indirect callsite to determine its argument count and types: If an argument register is in state *sn*, then it is considered an argument that uses *n* bits; the argument count is determined by the highest argument register whose state is *sn*.

10.3.6 Return Values

Knowing more information about return values of functions increases CFI precision. For instance, an indirect callsite that expects a return value should not call a function that does not return a value; similarly, an indirect callsite that expects a 64-bit return value should not call a function that returns only a 32-bit value. For calltarget analysis, τCFI traverses backwards from the return instruction of a function and searches for uses of the `RAX` register to determine if a function has a void or a non-void return type. In case there is a write operation on the `RAX` register, τCFI infers that the function's return type is non-void; furthermore, it tracks the widths of write operations to infer the width of the return type. For calltarget return-value type estimation, overapproximations are allowed.

At a callsite, τCFI traverses forward from the callsite to search for reads before writes on the `RAX` register to determine if a callsite expects a return value or not. In case there is such a read on the `RAX` register, τCFI infers that the callsite expects a return value; furthermore, it tracks the widths of read operations to infer the width of the expected return value. For callsite return-value type estimation, underapproximation is allowed.

10.3.7 Backward-Edge Analysis

In order to protect backward edges, we have designed an analysis that can determine possible legitimate return target addresses for each callee function. Our algorithm used for computing the legitimate set of addresses for each callee works as follows. First, a map is obtained after running the callsite and calltarget analysis (see Section 10.3.4 and Section 10.3.5 for more details); it maps a callsite to the set of legal calltargets where forward-edge indirect control-flow transfer is allowed to jump. This map is then reversed to build a second map that maps from the return instruction of a function (callee) to a set of addresses where the return can transfer to.

The return target address set for a function return is determined by getting the next address after each callsite address that is allowed to make the forward-edge control flow transfer. The map is obtained by visiting a return instruction address in a function and assigning to it the addresses next to callsites that can call the function. At the end of the analysis, all callsites and all function returns have been visited and a set of backward-edge addresses for each function return address is obtained. Note that the function boundary address (*i.e.,* `ret`) is detected by a linear basic block search from the beginning of the function (calltarget) until the first return instruction is encountered. We are aware that other promising approaches for recovering function boundaries (*e.g.,* [6]) exist, and plan to experiment with them in future work.

10.3.8 Binary Instrumentation

10.3.8.1 Forward-Edge Policy Enforcement

The result of the callsite and calltarget analysis is a mapping that maps a callsite to its allowed calltargets. In order to enforce this mapping during runtime, callsites and calltargets are instrumented inside the binary program with two labels. Additionally, each callsite is instrumented with CFI checks. At a callsite, the number of provided arguments is encoded as a series of six bits. At a calltarget, the label contains six bits encoding how many parameters the calltarget expects. Additionally, at a callsite 12 bits encode the register-width types of the provided arguments (two bits for each parameter), while at the calltarget another 12 bits are used to encode the types of the parameters expected. Further, at a callsite, several bits are used to encode if the function is expecting a `void` return type or not, and the width of the return type if it is nonvoid (similarly for a calltarget). All this information is written in labels before callsites and calltargets. During runtime before a callsite, these labels are compared by performing an `XOR` operation. In case the `XOR` operation returns false (a zero value), the transfer is allowed; otherwise, the program execution is terminated.

10.3.8.2 Backward-Edge Policy Enforcement

Based on the previously determined reverse map, before each function return a randomly generated label value is inserted. We decided to use these kinds of values as our main requirement is to map a return to a potentially large number of return sites. The same label is inserted before each legitimate target address (the next address after a legitimate callsite). In this way, a function return is allowed to jump only to the instruction that follows next to the address of a callsite.

For callsites that target a calltarget that is also allowed by another callsite, τCFI performs a search in order to detect if the callsite already has a label attached to the address after the callsite. If so, a new label is generated and multiple labels are stored for the address following the callsite. In this way, calltarget return labels are grouped together based on the reverse map. This design allows the same number of function return sites as the forward-edge policy enforces for each callsite. Finally, in case the comparison returns true, the execution continues; otherwise, it is terminated.

10.3.9 Implementation

We have implemented τCFI using the DynInst [18] (v.9.2.0) instrumentation framework with a total of 5,501 lines of C++ code. We currently restricted our analysis and instrumentation to x86-64 executables in the ELF format using the Itanium C++ ABI calling convention. τCFI can deal with the level of executable obfuscation with which DynInst can deal. As such, we fully delegate this responsibility to the used instrumentation framework underneath. We focused on the Itanium C++ ABI convention as most C/C++ compilers on Linux implement this ABI. However, the implementation separated the ABI-dependent code, so we expect it to be possible to support other ABIs as well. We developed the main part of our binary analysis pass in an instruction analyzer, which relies on the DynamoRIO [26] library (v.6.6.1) to decode single instructions and provide access to its information. The analyzer is then used to implement our version of the reaching-definition and liveness analysis. Further, we implemented a Clang/LLVM (v.4.0.0, trunk 283889) backend (machine instruction) pass (416 LOC) used for collecting ground truth data in order to evaluate the effectiveness and performance of our tool. The ground truth data is then used to verify the output of our tool for several test targets. This is accomplished with the help of our Python-based evaluation and test environment implemented in 3,239 lines of Python code.

10.4 Evaluation

We have evaluated τCFI by instrumenting various open source applications and conducting a thorough analysis in order to show its effectiveness and usefulness. Our test applications include the following real-world programs: FTP servers *Vsftpd* (v.1.1.0, `C` code), Pure-ftpd (v.1.0.36, `C` code) and *Proftpd* (v.1.3.3, `C` code); Lighttpd web server (v.1.4.28, `C` code); two database server applications *Postgresql* (v.9.0.10, `C` code) and *Mysql* (v.5.1.65, `C++` code); the memory cache application *Memcached* (v.1.4.20, `C` code); and the *Node.js* application server (v.0.12.5, `C++` code). We selected these applications to allow for a fair comparison with other similar tools. In our evaluation, we addressed the following research questions (RQs):

- **RQ1:** How **effective** is τCFI? (Section 10.4.1)
- **RQ2:** What **security protection** is offered by τCFI? (Section 10.4.2)
- **RQ3:** Which **attacks** are mitigated by τCFI? (Section 10.4.3)
- **RQ4:** Are other forward-edge tools **better** than τCFI? (Section 10.4.4)
- **RQ5:** Is τCFI **effective** against COOP? (Section 10.4.5)
- **RQ6:** How does τCFI **compare** with Clang's Shadow Stack? (Section 10.4.6)
- **RQ7:** What **runtime overhead** does τCFI impose? (Section 10.4.7)

Comparison Method. We used τCFI to analyze each program binary individually. Next, τCFI was used to harden each binary with forward and backward checks. The data generated during analysis and binary hardening was written into external files for later processing. Finally, the previously obtained data was extracted with our Python-based framework and inserted into spreedsheet files in order to be able to better compare the obtained results with other existing tools.

Setup. Our setup is based on Kubuntu 16.04 LTS (k.v.4.4.0) using 3GB RAM and four hardware threads running on an i7-4170HQ CPU at 2.50 GHz.

10.4.1 Effectiveness

Table 10.1 depicts the average number of calltargets per callsite, the standard deviation σ, and the median. In Table 10.1, the abbreviation CS refers to the callsites, while CT means calltargets. Note that the restriction to address-taken functions (see column AT) is present. The label *count** denotes the best possible reduction using the parameter *count* policy based on the ground truth collected by our Clang/LLVM pass, while *count* denotes the results of

O2	**CS**	**CT**	**AT**	***count****				***count***				***type****				***type***			
Target	total	total	total	limit (mean ± σ)			median	limit (mean ± σ)			median	limit (mean ± σ)			median	limit (mean ± σ)			median
ProFTPD	157.0	1,011.0	396.0	349.3	±	52.8	369.0	370.1	±	43.3	382.0	338.2	±	64.8	361.0	354.4	±	85.1	390.0
Pure-FTPD	8.0	127.0	13.0	8.6	±	4.6	8.0	10.1	±	4.8	13.0	8.1	±	4.0	5.0	10.0	±	4.0	10.0
Vsftpd	2.0	391.0	10.0	8.0	±	2.0	8.0	10.0	±	0.0	10.0	6.0	±	2.0	6.0	7.0	±	3.0	7.0
Lighttpd	66.0	289.0	63.0	34.3	±	15.1	21.0	43.7	±	14.5	51.0	34.5	±	14.7	23.0	45.4	±	12.1	50.0
Nginx	270.0	914.0	1,111.0	316.8	±	146.9	266.0	447.6	±	124.0	528.0	317.5	±	146.4	267.0	450.9	±	110.2	528.0
MySQL	7,893.0	9,928.0	5,896.0	338.5	±	189.5	179.0	490.6	±	203.3	574.0	307.9	±	163.6	186.0	519.6	±	147.6	540.0
PostgreSQL	687.0	6,885.0	2,304.0	423.4	±	176.7	471.0	497.0	±	151.8	515.0	416.2	±	188.1	541.0	476.9	±	162.4	562.0
Memcached	48.0	134.0	14.0	12.3	±	2.3	14.0	13.0	±	1.4	14.0	12.7	±	1.0	12.0	12.8	±	1.0	12.0
NodeJS	10,215.0	20,196.0	7,230.0	763.1	±	329.3	806.0	1,051.2	±	293.2	1,169.0	683.2	±	332.9	459.0	939.8	±	314.0	1,022.0
geomean	170.1	1,104.8	259.8	89.0	±	31.2	79.4	110.4	±	27.0	123.1	83.7	±	28.1	69.3	104.7	±	27.8	111.6

Table 10.1: Allowed callsites per calltarget for τCFI's count and type policies.

our implementation of the parameter *count* policy derived from binaries. The same applies to *type*∗ and *type* regarding the parameter *type* policy. A lower number of calltargets per callsite indicates better results. Note that our parameter *type* policy is superior to the parameter *count* policy, as it allows for a stronger reduction of allowed calltargets. We consider this an important result, which further improves the state-of-the-art. Finally, we provide the median and the pair of mean and standard deviation to allow for a better comparison with other state-of-the-art tools.

10.4.1.1 Theoretical Limits

We explored the theoretical limits regarding the effectiveness of the *count* and *type* policies by relying on the collected ground truth data; essentially assuming perfect classification. Based on the type information collected by our Clang/LLVM pass, we derived the available number of calltargets for each callsite by applying the count and type policies. From the results, (1) the theoretical limit of the *count** policy has a geomean of 89 possible calltargets, which is around 8% of the geomean of the total available calltargets (1,104), and (2) the theoretical limit of the *type** policy has a geomean of 83 possible calltargets, which is 7.5% of the geomean of the total available calltargets (1104). In comparison, the theoretical limit of the *type** policy allows about 13% less available calltargets in geomean than the limit of the *count** policy (*i.e.,* 69.3 vs. 79.4).

10.4.1.2 Calltarget per Callsite Reduction

(1) The *count* policy has a geomean of 104 calltargets, which is around 9.4% of the geomean of all available calltargets (1104). This is around 24% more than the theoretical limit of available calltargets per callsite (see *count** 89 vs. 110.4). (2) The *type* policy has a geomean of 104.7 calltargets, which is 9.48% of the geomean of total available calltargets (1,104). This is approximatively 25% more than the theoretical limit of available calltargets per callsite (see *type** 83.7 vs. 104.7). τCFI's *type* policy allows around 9.4% less available calltargets in the geomean than our implementation of the *count* policy (104.7 vs. 110.4), and a total reduction of more than 94% (104.7 vs. 1,104) w.r.t. the total number of calltargets (CT) available once the *count* and *type* policies are applied.

10.4.2 Forward-Edge Policy vs. Other Tools

Target	**IFCC**	**TypeArmor (CFI+CFC)**	**AT**	**τCFI (count)**	**τCFI (type)**
ProFTPD	3.0	376.0	396.0	382.0	390.0
Pure-FTPD	0.0	4.0	13.0	13.0	10.0
Vsftpd	1.0	12.0	10.0	10.0	7.0
Lighttpd	6.0	47.0	63.0	51.0	50.0
Nginx	25.0	254.0	1,111.0	528.0	528.0
MySQL	150.0	3,698.0	5,896.0	574.0	540.0
PostgreSQL	12.0	2,304.0	2,504.0	515.0	562.0
Memcached	1.0	14.0	14.0	14.0	12.0
NodeJS	341.0	4,714.0	7,230.0	1,169.0	1,022.0
geomean	8.7	170.4	259.8	123.1	111.6

Table 10.2: Legitimate calltargets and callsite pair sets for five tools.

Table 10.2 provides a comparison between τCFI, TypeArmor and IFCC w.r.t. the median count of calltargets per callsite. The values for TypeArmor [268] and IFCC [261] depicted in Table 10.2 have been adopted from the corresponding papers in order to ensure a fair comparison. Further, Table 10.2 conveys the limitations of binary-based type analysis, as the median of the possible target set size for τCFI is several times larger than the corresponding set sizes for system using source-level analysis. Note that the smaller the geomean values are, the better the technique is. AT is a technique that allows a callsite to target any address-taken functions. IFCC is a compiler-based solution and is included here as a reference to show what is possible when the program's source code is available. TypeArmor and τCFI on the other hand are binary-based tools. τCFI reduces the number of calltargets by up to 42.9% (geomean) when compared to the AT technique, by more than seven times (7,230 vs. 1,022) for a single test program w.r.t. AT, and by 65.49% (170.4 vs. 111.6) in geomean when compared with TypeArmor, respectively. As such, τCFI represents a stronger improvement w.r.t. calltarget per callsite reduction in binary programs compared to other approaches.

10.4.3 Effectiveness Against COOP

We investigated the effectiveness of τCFI against the COOP attack by looking at the number of register arguments, which can be used to enable data flows between gadgets. In order to determine how many arguments remain unprotected after we apply the forward-edge policy of

τCFI, we determined the number of parameter overestimations and compared it with the ground truth obtained during an LLVM compiler pass. Next, we used some heuristics to determine how many ML-G and REC-G callsites exist in the C++ server applications. Finally, we compared these results with the one obtained by TypeArmor.

		Overestimation					
Program	#cs	0	+1	+2	+3	+4	+5
MySQL (ML-G)	192	184	3	1	0	1	3
Node.js (ML-G)	134	131	1	0	1	0	1
geomean	160	155	1	1	1	1	1
MySql (REC-G)	289	273	10	2	3	0	1
Node.js (REC-G)	72	69	2	0	0	0	1
geomean	144	137	4	1	1	1	1

Table 10.3: Parameter overestimation for the ML-G and REC-G gadgets.

Table 10.3 presents the results obtained after counting the number of perfectly estimated and overestimated protected ML-G and REC-G gadgets. As it can be observed, τCFI obtained a 96% (184 out of 192) accuracy of perfectly protected ML-G callsites for MySQL, while TypeArmor obtained a 94% accuracy for the same program. Further, τCFI obtained a 97% (131 out of 134) accuracy for Node.js, while TypeArmor obtained 95% accuracy on the same program. Further, for the REC-G case, τCFI obtained an 94% (273 out of 289) exact-parameter accuracy for MySQL, while TypeArmor had 86%. For Node.js, τCFI obtained an accuracy of 95% (69 out of 72), while TypeArmor had 96%. Overall τCFI's forward-edge policy obtained a perfect accuracy of 95%, while TypeArmor obtained 92%. While this is not a large difference, we want to point out that the remaining overestimated parameters represent only 5% and thus do not leave much wiggle room for the attacker.

10.4.4 Comparison with the Shadow-Stack

The shadow stack implementation of Abadi *et al.* [1] provides a strong security protection [32] w.r.t. backward-edge protection. However, it: (1) has a high runtime overhead ($\geq$ 21%), (2) is not open source, (3) uses a proprietary binary analysis framework (*i.e.,* Vulcan), and (4) loses precision due to equivalent class merging. Hence, we propose an alternative backward-edge protection solution. In order to show the precision of τCFI's backward-edge protection, we provide the average number of legitimate return addresses for return instructions and compare it to the total number of available addresses without any protection.

Table 10.4 presents the statistics w.r.t. the backward-edge policy legitimate return targets. More specifically, in Table 10.4, we use the following abbreviations: total number of return addresses (Total #RA), total (median) number of return address targets (Total #RATs), total (median) number of return address targets per return instruction (Total. # RATs/RA), percentage of legitimate return address targets per return addresses w.r.t. the total number of addresses

Program	Total #RA	Total #RATs	Total #RATs/RA	%RATs/RA prog. binary
MySQL	5,896.0	3,792.0	0.6	0.014%
Node.js	7,230.0	3,864.0	0.53	0.011%
geomean	6,529.0	3,827.0	0.58	0.012%

Table 10.4: Backward-edge policy statistics.

in the program binary (% RATs/RA w.r.t. program binary). By applying τCFI's backward-edge policy, we obtain a reduction of 0.42 ($1-0.58$) ratio (geomean) of the total number of return address targets per return address over the total number of return addresses. This means that only 43% of the total number of return addresses are actual targets for the function returns. The results indicate a percentage of 0.012% (geomean) of the total addresses in the program binaries are legitimate targets for the function returns. This means that our policy can eliminate 99.98% (100% - 0.012%) of the addresses, which an attacker can use for his attack inside the program binary. To put it differently, only 0.012% of the addresses inside the binary can be used as return addresses by the attacker. Further, we assume that the attacker cannot easily determine which addresses are still available for any given program binary, as it is stripped from debug information. Note that each function q1 return (callee) is allowed to return in geomean to around 111 legitimate addresses (MySQL 519 and NodeJS 939) in all analyzed programs. Finally, we assume that it is hard for the attacker to find out the exact set of legitimate addresses per return site once the policy was applied.

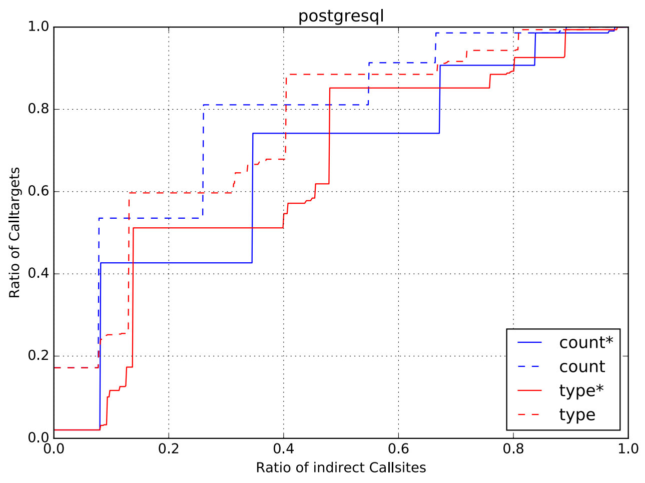


Figure 10.2: CDF for the PostgreSQL program.

10.4.5 Security Analysis

Figure 10.2 depicts the cumulative distribution function (CDF) for the PostgreSQL program compiled with the Clang `-O2` flag. We selected this program randomly from our test programs. The CDF depicts the relation between the ratio of indirect callsites and the ratio of calltargets, for the type and the count policies. While the CDFs for the count policies have only a few changes, the amount of changes for the CDFs of the type policies is vastly higher. The reason for this is fairly straightforward: the number of buckets (*i.e.,* the number of equivalence classes) that are used to classify the callsites and calltargets is simply higher for the type policies. Finally, note that the results depend on the internal structure of the particular program and may for this reason vary for other programs.

10.4.6 Mitigation of Advanced CRAs

Table 10.5 presents several attacks that can be successfully stopped by τCFI by deploying only the forward-edge or the backward-edge policy.

Exploit	**Stopped**	**Remark**
COOP ML-G [238]		
IE 32 bit	×	Out of scope
IE 1 64-bit	✓(FP)	Arg. count mismatch
IE 2 64-bit	✓(FP)	Arg. count mismatch
Firefox	✓(FP)	Arg. count mismatch
COOP ML-REC [59]		
Chrome	✓(FP)	Void target where non-void was expected
Control Jujutsu [80]		
Apache	✓(FP)	Target function not AT
Nginx	✓(FP)	Void target where non-void was expected
All Backward edge violating attacks	✓(BP)	(1)[a] or (2)[b] or (3)[c]

[a] Jump to address $\notin$ in the $max - min$ address range.
[b] Jump to address $\neq$ then a legitimate address.
[c] Jump to address label $\neq$ the calltarget return label.

Table 10.5: Stopped CRAs, forward-edge policy (FP) & backward-edge policy (BP).

For checking if the COOP attack can be prevented, we instrumented the Firefox library (`libxul.so`), which was used to perform the original COOP attack as presented in the original paper. We observed that due to the forward-edge policy this attack was no longer possible. For testing if backward-edge attacks are possible after applying τCFI, we used several open source ROP attacks that are explicitly violating the control flow of a C++ program through backward-edge violations. Next, we instrumented the binaries of these programs. Each attack that was using one of the protected function returns was successfully stopped.

In summary, many forward-edge and backward-edge attacks can be successfully mitigated by τCFI as long as these attacks are not aware of the policy in place and thus cannot selectively use gadgets that have their start address in the allowed set for the legitimate forward-edge and backward-edge transfers, respectively.

10.4.7 Runtime Overhead

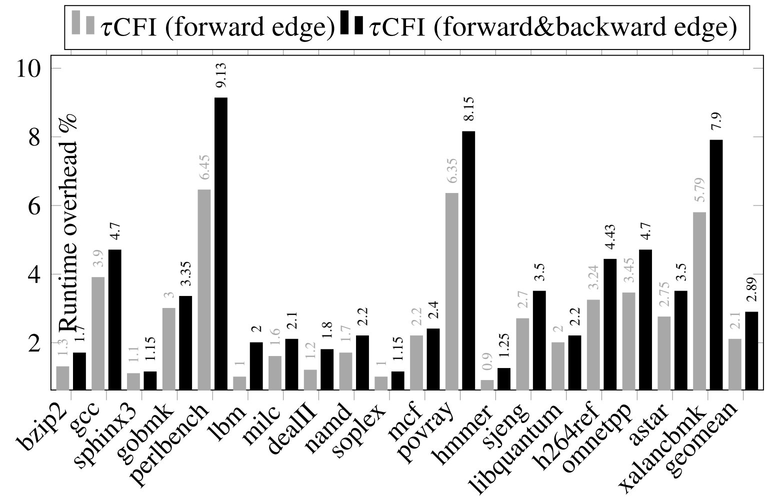


Figure 10.3: Runtime overhead.

Figure 10.3 presents the runtime overhead obtained by applying τCFI's forward-edge policy (register type; parameter count) and backward-edge policy on all `C/C++` programs contained in SPEC CPU2006. Out of the evaluated programs: `xalancbmk`, `namd`, `omnetpp`, `dealII`, `astar`, `soplex`, and `povray` are `C++` programs, while the rest are pure `C` programs. After the programs were instrumented, we measured the runtime overhead. The geomean of the instrumented programs is around 2.89% runtime overhead. One reason for the performance drop is cache misses introduced by jumping between the old and the new executable section of the binary generated by duplicating and patching. This is necessary, because when outside of the compiler, it is difficult to relocate indirect control flow. Therefore, every time an indirect control flow occurs, jumps into the old executable section and from there back to the new executable section occur. Moreover, this is also dependent on the actual structure of the target as the overhead depends on the frequency of indirect control flow operations. Another reason for the slightly higher (yet acceptable) performance overhead is our runtime policy, which is more complex than that of other state-of-the-art tools.

10.5 Discussion

In this Section, we present some limitations of τCFI and indicate several attacker policy discovery trade-offs.

10.5.0.1 Limitations

First, τCFI is limited by the capabilities of the DynInst instrumentation environment, where non-returning functions like exit are not detected reliably in some cases. As a result, we cannot test the Pure-FTP server, as it heavily relies on these functions. The problem is that those non-returning functions usually appear as a second branch within a function that occurs after the normal control flow, causing basic blocks from the following function to be attributed to the current function. This results in a malformed control flow graph and erroneous attribution of callsites and problematic misclassifications for both calltargets and callsites.

Second, parameter passing through floating point registers is currently not supported by τCFI, similar to other state-of-the-art tools. Tail calls are also not supported for now as they lose the one-to-one matching between callers and callees. Further, τCFI does not support self-modifying code as code pages become writable at run-time. We plan to address this limitation in future work.

Third, τCFI is not intended to be more precise than source code based tools such as IFC-C/VTV [261]. However, τCFI is highly useful in situations when the source code is typically not available (*e.g.,* off-the-shelf binaries), where programs rely on third-party libraries, and where the recompilation of all shared libraries is not possible.

Finally, while a major step forward, τCFI cannot thwart all possible attacks, as even solutions with access to source code are unable to protect against all possible attacks [36]. In contrast, τCFI, our binary-based tool can stop all COOP attacks published to date and significantly raises the bar for an adversary when compared to other state-of-the-art tools. Moreover, τCFI provides a strong mitigation for other types of code-reuse attacks as well as for attacks that violate the caller-callee function calling convention.

10.5.0.2 Attacker Policy Discovery Trade-offs

In general, with usage of CFI techniques, it is relatively unchallenging for an attacker to figure out where an indirect program control flow may transfer during runtime. This is because the indirect transfer targets (backward and forward) are labeled with IDs that have to satisfy certain conditions, *e.g.,* a bitwise `XOR` operation between the bits of the start and target address of indirect control flow transfer should return a one or zero in case the transfer is legal or illegal, respectively.

Thus, we note that in general it is not difficult for a resourceful attacker to figure out which callees match to which calltargets or vice versa when these are labeled with `IDs` for example. τCFI is not exempted from this. In general, if the attacker knows where an indirect transfer is allowed to jump to, he may use this wiggle space to craft his attack with the available (reachable) gadgets. The main assumption on which CFI and τCFI are built upon is that the wiggle room is sufficiently reduced for an attacker such that the likelihood for a successful attack is greatly diminished.

10.6 Summary

In this Chapter, we presented τCFI, a tool based on novel control flow integrity (CFI) defense policies, which can be used to protect program control flow graph (CFG) forward-edges and backward-edges in program binaries during runtime. For the protected stripped (*i.e.,* no runtime type information (RTTI) available) x86-64 binaries, we do not need to make any assumptions on the presence of an auxiliary technique for protecting backward edges (*i.e.,* shadow stacks, *etc.*) as τCFI protects these transfers, too. We have evaluated τCFI with real world open source programs and have shown that τCFI is practical and effective when protecting program binaries. Finally, our evaluation results reveal that τCFI can considerably reduce the forward-edge legal calltarget set, provide high backward-edge precision, while maintaining low runtime overhead.

Chapter 11 Conclusion and Future Work

11.1 Conclusion

In this thesis, we discussed the fact that mitigation of code reuse attacks (CRAs) is of high importance for academia and industry. Even through there are many out-of-the-box solutions out there which amongst other research areas also focus on the main research areas covered in this thesis these have certain limitations. As such, in this thesis we want to prpose: (1) tools which can be used for early (*i.e.,* during development) and late (*i.e.,* during deployment) detection and repairing of faults which can lead to potentially exploitable memory corruptions that are in almost all cases the starting point for CRAs, (2) compiler based sanitizers which gained considerable attention in the last few years as these were, for example, successfully integrated in fuzzing environments (*e.g.,* Google's OSS-Fuzz [100]), and (3) tools for hardening of legacy program binaries; none of these industry provided approaches solve the problem of fully avoiding CRAs and at most can claim that they can combat a certain part of an attack prerequisite. For this reason, we nowadays see an arms race in full development. In this race, multiple protection techniques (new hurdles) are constantly being deployed with the goal to considerably or fully reduce the likelihood of such attacks.

Further, in this thesis, we thoroughly reviewed these three main research areas mentioned above and proposed effective solutions which considerably improve the state-of-the-art in all of these three areas. Finally, we conclude this thesis by briefly reiterating through our contributions in Section 11.1.1, presenting some future research avenues in Section 11.2 and providing some final remarks in Section 11.3.

11.1.1 Contributions

In this Section, we present our contributions which cover three main interconnected research areas with which advanced CRAs can be mitigated. We started our thesis by looking into static source code approaches to mitigate memory corruptions by searching and repairing them, in this avoiding potential CRAs. The common belief with this type of mitigation is that a system with less memory corruptions are less CRAs prone. Next, we investigated runtime based mitigation of memory corruptions which could lead to CRAs. The intuition behind these decisions was that since static analysis techniques have their limitations also thoroughly mentioned in the thesis it is a natural decision to cut trough these limitations and use the full spectrum of the program runtime information which is only available during runtime. Within this research area we investigated how to assess existing state-of-the-art static CFI defenses and provided a solution for protecting backward edges. Finally, the third research area covered was binary hardening against CRAs. The intuition behind perusing this interconnected research area is that in some situations source code is not available, code is shared only in binary form by some vendors in order to guard market advantages, and finally it is a well established research area which gained a lot of attention in the last years.

We now briefly visit the contribution details w.r.t all these three main research areas investigated within this thesis.

11.1.1.1 Static source code analysis for automated memory corruption localization and repair generation.

In the first part of this thesis, more specifically, in Chapter 4, we presented a framework inside which, we first designed and implemented and second, integrated our integer overflow detection tool, called IntRepair. This tool is capable of efficiently detecting integer overflows in `C` source code programs. With this approach, we answer **RQ1** by providing a tool which can reliably detect integer overflows and does not suffer from false negatives. Further, the tool is integrated in a well established and widely used IDE (*i.e.,* Eclipse IDE).

Next, in Chapter 5, we extended our static `C` source code analysis framework by presenting an integer overflow detection and repair tool which can provide source code repairs that help a programmer to automatically repair an integer overflow. Within this Chapter, we addressed **RQ2** by providing an integer overflow repair tool, named IntRepair, which is capable of efficiently removing a fault, can automatically validate a repair, and does not introduce unwanted program behavior.

Further, in Chapter 6, we extended our static source code analysis framework in order to be able to detect and repair buffer overflows which are in many situations a main prerequisite for CRAs. In this Chapter, we provided a tool, called BuffRepair, which can automatically generate buffer overflow repairs, does not suffer from false negatives, and can validate a repair. At the same time, we fully addressed **RQ3** and integrated our tool into a well used and established IDE as we did with both IntRepair and IntRepair.

11.1.1.2 Dynamic source and machine code based analysis and instrumentation for runtime memory corruption detection, CRAs mitigation and static CFI policies assessment.

In the second part of this thesis, specifically, in Chapter 7, we investigated the detection of dynamic memory corruptions detections which can most commonly lead to (or are a prerequisite for) CRAs. We were motivated to follow this path due to the intrinsic limitations of static analysis tools. More specifically, we developed a compiler based sanitizer tool, called CastSan, which can detect object type confusions during runtime and which is fully integrated into a well established compiler framework (Clang/LLVM). Within this Chapter, we address **RQ4** by providing a fully functional object type confusion tool which is based on a novel and efficient technique and in case of consistent use is able to considerably reduce the likelihood of CRAs.

Next, in Chapter 8, we designed a static compiler based tool, called LLVM-CFI, which is usable for assessing static state of the art CFI policies. The intuition behind this decision is motivated twofold. Firstly, due to the fact that it is most likely that memory corruptions can not be fully eliminated we wanted to provide an active defense by hardening the program. Secondly, we wanted to come up with novel CFI based techniques for protecting indirect program control flow transfers. In order to effectively address this task we wanted to learn how effective the existing CFI defenses are and what level of security they offer. As such, we addressed **RQ5** by providing a framework for assessing static CFI policies w.r.t. calltarget set reduction after a certain CFI defense was applied. Further, in this thesis we learned important lessons which helped us prepare next design decisions.

Further, in Chapter 9, we designed and implemented a compiler based tool, called ρFEM, which is based on the lessons learned in Chapter 8. ρFEM protects program CFG backward edges stemming from indirect and direct forward-edge program control flow transfers. In this way, we were able to address **RQ6** and propose a novel technique for protecting program CFG backward-edges with the help of a fine grained CFI policy which provides an optimal set of return targets for each protected callee. In this way the likelihood of successfully performing CRAs which exploit backward edges is greatly reduced. As such, we provide in this Chapter a solution which is an alternative for shadow stack based techniques. In contrast to shadow stack techniques, ρFEM does not rely on entropy and information hiding. Thus, the according protection disclosing vectors do not apply.

Finally, in Chapter 10, we developed a framework, called τCFI, for protection of legacy program binaries with new CFI policies designed by taking into account the lessons learned in Chapter 8. These learned lessons helped us to thoroughly address **RQ7** in order to provide a tool which can effectively protect forward and backward program CFG edges in stripped binaries (*i.e.,* most of the semantic information has vanished through the compilation process) as this type of information is usually not required in production-ready binaries. In this way CRAs which rely on corrupting forward and/or backward CFG edges due to indirect control flow transfer violations are mitigated by greatly reducing the likelihood of successfully performing such an attack when hardening the program binary with τCFI.

11.2 Future Work

In this Section, we briefly present some future work and research avenues for each of the tools we presented within this thesis.

11.2.1 Next Steps related to IntDetect

In future work, we want to extend the CFG building algorithm in order to support C++ control flow related semantics—object instantiation, dispatching, templates, *etc.* Additionally, we want to implement other symbolic function models which are relevant for detection of other types of integer related vulnerabilities. Further, we want to extend our tool such that it is possible to detect integer underflows, and other inter overflow related problems such: integer coercion errors, off-by one errors, unexpected sign extension, and signed to unsigned conversion errors. Finally, we want to implement mechanisms for checking if the integer overflow was intended or unintended and also to differentiate between benign (*i.e.,* not exploitable) and malicious (*i.e.,* exploitable) integer overflows.

11.2.2 Next Steps related to IntRepair

In future work, we aim to extend INTREPAIR to repair other integer related errors such as signedness errors [27, 253] caused by type conversions, and handle the full C/C++ language features as described in Section 11.2.1. More specifically, we want to extend the set of repair patterns for different types of integer related problems as well as for the currently studied integer overflows. Further, we want to implement an automated approach which ranks multiple patterns which may fit to repair a certain integer overflow by letting the programmer to choose from a list of possible features (*e.g.,* minimal code size, usage of mathematical functions, in-place repairing, not in-place repairing, *etc.*). We think that these repair prioritization features will allow the programmer to better decide which repair fits best to his particular needs.

11.2.3 Next Steps related to BuffRepair

In future work, we want to extend BUFFREPAIR such that it can detect other types of buffer overflow related problems (*e.g.,* stack and heap based buffer attacks, arithmetic attacks and format string attacks) and types (*i.e.,* heap based buffer overflow). Further, we want to differentiate between benign (*i.e.,* not exploitable) and malicious (*i.e.,* exploitable) buffer overflows as well as programmer intended and unintended buffer overflows. As a natural consequence of extending the types of buffer overflow which BUFFREPAIR could handle we want to extend our set of buffer overflow repair patterns which need to be adapted for these new scenarios. Thus, as we currently support *in-place* and *non-in-place* buffer overflows we also want to research other optimal approaches for finding *non-in-place* repairs which are sound and potentially can remove the fault on many program CFG execution paths. Finally, once the fault is fixed then it will not manifest in all CFG nodes contained on all CFG program paths located under the *non-in-place* repair location.

11.2.4 Next Steps related to CastSan

In this Section, we present several future research direction w.r.t. how to increase the coverage of CastSan, and the possibility of reusing the metadata used by CastSan in other types of compiler based sanitizers is discussed. Finally, we present how CastSan can be extended to check for non polymorphic object type confusions as well.

Increasing Coverage. CastSan can only check casts between polymorphic classes. This is due to its dependence on virtual function pointers, which non-polymorphic classes do not have. Further we note that since the virtual class hierarchy which the vptrs are derived from is a transformation of the class hierarchy graph to trees, we think that this concept could be extended to non-polymorphic classes as well. To achieve this, a metadata runtime similar to HexTypes [125] could be used to store type metadata. Instead of only creating trees from polymorphic classes, all classes could be included into the trees. A polymorphic object would then save their own vptr while a non-polymorphic object stores it using the runtime metadata service. The cast check itself would then still be an efficient range based check, without the need to iterate over lists. Also, the amount of objects that need to be tracked during runtime could drastically be reduced, due to the fact that polymorphic objects do not rely on the metadata.

Reusing the Virtual Table Metadata. IVT uses the ordered vtables to provide a fine grained control flow integrity policy for the CFG forward-edges during runtime. As the vtable layout provides a limited view on the class hierarchy, it can be explored if it can replace runtime type information (RTTI) as a faster runtime type information provider. Together with the auxiliary vtables proposed in Section 11.2.4, the metadata can be extended to provide runtime type information for classes without vtables as well. Finally, note that RTTI on the other hand only provides data for polymorphic classes.

Dynamic Cast. C++'s `dynamic_cast` is used for checking the type safety of object casts at runtime. In LLVM this is implemented using RTTI metadata. RTTI is a slow recursive way of storing raw runtime type information. This allows for a secure but slow checks (up to 90 times slower than static dispatches, see Lee *et al.* [142] for more details) of object cast during runtime. In large and performance oriented programs it is often not advisable to use `dynamic_cast`, as its disadvantages out-weights its benefits. Similar to CastSan, `dynamic_cast` is limited to polymorphic objects. Therefore, CastSan could be extended to check `dynamic_cast` in addition to `static_cast`. Finally, we think that by extending CastSan in this way, programs that make heavy use of `dynamic_cast` could be speed up tremendously.

Extend to Non-polymorphic Objects. In future work, we want to use our static metadata based technique to extended existing purely runtime based object type confusion detection tools such as TypeSan and HexType. These tools use for both polymorphic and non-polymorphic object type checking a runtime library which adds considerable runtime overhead due to updates, search, and deletion of object type meta-data data. We think that our approach can be used to avoid the runtime tracking of object type metadata for polymorphic objects. Further, a complementary artificial virtual table like metadata class hierarchy can be built for non-polymorphic objects as well. Finally, in this way our technique can be used in this context as well, thus avoiding or considerable reducing the overhead introduced by the runtime compiler checking support.

Medata Usage for Other Sanitizer Types. We envisage that our portable projected virtual table hierarchy list technique can be used by other object type confusion detection tools to speed-up their object cast checking mechanism. In particular, our technique could be used to speed-up, for example, `C++` runtime casts such as, `dynamic_cast<>`, which have to perform a non-trivial traversal of RTTI information to check safety. This would help to further reduce the runtime overhead by reducing the computational overhead which is mainly caused by (1) intensive interaction between the instrumented code and compiler runtime libraries, and (2) variable time lookup checking mechanisms. (*i.e.,* Compiler-RT based).

11.2.5 Next Steps related to LLVM-CFI

In this Section, we briefly address possible future research directions.

Measuring Gadget Usability. In order to measure gadget usability w.r.t. a particular CRA, two situations have to taken into account. First, in case the application was not previously hardened with any policy one can decide based on the gadget set for the attack he/she wants to perform if a sequence of machine instructions can be used during the attack and if these are with reasonable effort reachable. Second, in case the binary was hardened with a protection policy against CRAs things get more complicated since the attacker not only has to locate the particular types in the application but he has to know how to avoid getting into the previously inserted checks. On one hand, in case the checks are easy to spot then he can use what ever gadget he/she can get to. On the other hand, if the attacker can not determine which protection mechanism was applied to the program he does only know that the gadget is available but not usable for the attack. In order to measure gadget usability our LLVM-CFI can be extended with novel metrics which take into account individually for each gadget if is protected or not and if a gadget belongs to a certain type of CRA.

Measuring Gadget Linkability. The starting point for a gadget linkalibility metric which basically measures how hard or easy is to link gadgets to each other is as follows. First, the semantics used to link gadgets have to be defined and second, these have to be related to the relevant LLVM-CFI metrics which can quantify the presence of a COOP gadget inside a virtual function. A starting point in this research direction was made by Crafted [128] which present an LLVM compiler based tool which can determine all possible gadget chains inside a given program based on an augmented CFG, instruction mapper and a control flow verifier which can be used to determine if it is possible to execute a given instruction sequences identified by the instruction mapper without violating the control-flow restrictions of the defense policy.

Measuring Attack Success Rate. Measuring attack success rate is based on gadget (1) usability,(2) linkability, (3) gadget chain and (4) goal policy. In order to measure CRA success rate first, a metric should derived which can measure if a certain type of gadget is available and second, if this gadget is linkable (can be used in a certain gadget chain). More precisely, first, the gadget chain policy takes the rational of the attacker w.r.t. what he/she wants to achieves and translates it to the needed gadgets that have to be performed. This can be mapped to a certain gadget chain. Second, the goal of the attacker has to be determined in terms of what he want to achieve by chaining and calling his gadgets one by one after each other. Third, the sequence of how he can achieve his goal needs to be mapped to all possible gadget chains which can be formed out of the available gadget set. An attack is in our opinion succeeds if the last gadget in the gadget chain was reached (successfully used) and if the obtained result matches with the initial goal of the attacker. Finally, in future we want to derive a metric which can asses the success rate of an CRA by taking into account the points mentioned above (1) - (4).

11.2.6 Next Steps related to ρFEM

Static Analysis. As noted before, our approach for protecting indirect callsites, which are not virtual dispatch callsites, is not optimal. One obvious improvement would be to use a second stage of static analysis to infer more information about the function pointer being used at indirect callsites. Since for most indirect callsites the pointer is chosen from a small number of function targets right before the call happens, we assume that by using a static analysis algorithm to narrow down the target set would result in substantial precision improvement. Thus, one main requirement is that the algorithm used had to be conservative in order to not break the program.

Dynamic Analysis. While we consider dynamically generated checks like the ones of shadow stack techniques of questionable use, they can work alongside ρFEM. Incorporating dynamic information to ρFEM would allow to check the value on the shadow stack not only with the potentially malformed return address, but also to compare it with our static set. A shadow stack storing only a tiny flag to differentiate between indirect, virtual and direct calls could be used to narrow down, the type of call which was made. Such a shadow stack would only have to be a few bytes in size and therefore easier to hide in memory due to the so obtained higher address space entropy.

11.2.7 Next Steps related to τCFI

Improving the Structural Matching. Improving the structural matching capability is in our opinion the most important further venue of future research, as we need a reliable way to match ground truth information against the resulting binary. This is important because since it is a prerequisite to the ability to generate reliable measurements and reduces the current uncertainty (*i.e.,* we rely on the number of calltargets per callsite to match callsites and furthermore assume that the order within ground truth and binary is the same).

Improving the Patching Schema. Devising a patching schema that is based on Dyninst functionality, which allows annotation of calltargets such that these can hold at least 4-byte of arbitrary data is another future research goal. This is required to hold the type data that we generate using our classification. Keeping the runtime overhead of this binary patching schema low is the yet another future research goal.

Using Pointer and Memory Analysis. Introducing pointer/memory analysis to distinguish between 32-bit and 64-bit values and actual addresses to even further restrict the possible number of calltargets per callsite is another future reearch target. This requires in our opinion a more precise data flow analysis, then calculating possible values for registers at each instruction.

Filtering Forward Edges. As demonstrated by the VCI [78] and Marx [218] tools it is possible to reconstruct a quasi class hierarchy (*i.e.,* no class root node, edges are not oriented) from a stripped program binary. Next binary checks based on paths formed by the object calling relationships can be inserted. In future work, we want to implement several algorithms similar to the ones described in the above tools in order to reconstruct a quasi class hierarchy with high accuracy and use it to compute possible calltarget sets for each previously detected callsite. Finally, these sets can be superimposed on the sets determined by our currently available forward edge policy with the goal to further reduce the legitimate calltarget set per callsite.

11.3 Final Remarks

In this thesis, we presented several tools which can help mitigate advanced CRAs in the user space. We think that our highlighted limitations of available CRAs mitigation tools, our results, and especially the problems related to current static CFI defenses are useful for the design and implementation of future approaches to combat existing and future CRAs more effectively.

Bibliography

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control Flow Integrity. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2005.

[2] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control Flow Integrity Principles, Implementations, and Applications. In *Transactions on Information and System Security (TISSEC)*. ACM, 2009.

[3] A. V. Aho and T. G. Peterson. A minimum-distance error-correcting parser for context-free languages. In *SIAM Journal of Computation*, 1972.

[4] J. P. Anderson. Computer Security Technology Planning Study. Technical report, AFSC, October 1972.

[5] D. Andriesse, X. Chen, V. v. d. Veen, A. Slowinska, and H. Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX, 2016.

[6] D. Andriesse, A. Slowinska, and H. Bos. Compiler-Agnostic Function Detection in Binaries. In *Proceedings of the European Symposium on Security and Privacy (Euro S&P)*. IEEE, 2017.

[7] Apache. Apache Httpd, 2017. `https://httpd.apache.org/`.

[8] Apache. Apache Traffic Server. 2017. `http://trafficserver.apache.org/`.

[9] ARM. C++ ABI for the ARM Architecture, 2015. `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0041e/IHI0041Ecppabi.pdf`.

[10] K. Ashcraft and D. R. Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2002.

[11] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley. Enhancing Symbolic Execution with Veritesting. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1083–1094. IEEE/ACM, 2014.

[12] K. Avijit, P. Gupta, and D. Gupta. TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. In *Proceedings of the Conference on Security Symposium (SSYM)*. ACM, 2014.

[13] M. Backes and S. Nuerenberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX, 2014.

[14] G. Balakrishnan and T. Reps. DIVINE: Discovering Variables in Executables. In *Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 2007.

[15] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard Version 2.0, 2010. `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf`.

[16] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard Version 2.0. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2010.

[17] E. Bauman, Z. Lin, and K. Hamlen. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*, San Diego, CA, February 2018.

[18] A. R. Bernat and B. P. Miller. Anywhere, Any-Time Binary Instrumentation. In *Proceedings of the Workshop on Program Analysis For Software Tools and Engineering (PASTE)*, 2011.

[19] T. Bletsch, X. Jiang, and V. Freeh. Mitigating Code-reuse Attacks with Control-flow Locking. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2011.

[20] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-Oriented Programming: A New Class of Code-Reuse Attack. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2011.

[21] BlueLotus. bctf challenge: Bypass vtable read-only checks., 2015. `https://goo.gl/4RYDS2`.

[22] BlueLotus Team. Bctf challenge: Bypass vtable Read-only Checks, 2015. `https://goo.gl/4RYDS2`.

[23] D. Bounov, M. D., S. A. Carr, and M. Payer. CFIXX: Object Type Integrity for C++ Virtual Dispatch. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2018.

[24] D. Bounov, R. Gökhan K., and S. Lerner. Protecting C++ Dynamic Dispatch Through VTable Interleaving. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2016.

[25] S. Brookes, R. Denz, M. Osterloh, and S. Taylor. NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[26] D. Bruening. DynamoRIO. `http://dynamorio.org/home.html`.

[27] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically Protecting Against Integer-based Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2007.

[28] D. Brumley, I. Jager, T. Avgerinos, and E. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the International Conference on Computer Aided Verification (CAV)*. Springer, 2011.

[29] D. Brumley, D. X. Song, T. Chiueh, R. Johnson, and H. Lin. RICH: Automatically protecting against integer-based vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2007.

[30] Bubblemark. Balls Browser Benchmark, 2017. `http://bubblemark.com/`.

[31] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2008.

[32] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-Flow Integrity: Precision, Security, and Performance. In *Computing Surveys (CSUR)*. ACM, 2017.

[33] J. Caballero and Z. Lin. Type Inference on Executables. In *Computing Surveys (CSUR)*. ACM, 2016.

[34] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, volume 8, pages 209–224. USENIX, 2008.

[35] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM (CACM)*, 56(2):82–90, 2013.

[36] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX, 2015.

[37] N. Carlini and D. Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX, 2014.

[38] E. N. Ceesay, J. Zhou, M. Gertz, K. N. levitt, and M. Bishop. Using type qualifiers to analyze untrusted integers and detecting security flaws in C programs. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2006.

[39] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-oriented Programming Without Returns. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2010.

[40] G. Chen, H. Jin, and D. Zou. SafeStack: Automatically Patching Stack-based Buffer Overflow Vulnerabilities. In *Proceedings of the Transactions on Dependable and Secure Computing (TDSC)*, volume 10. IEEE, 2013.

[41] P. Chen, H. Han, Y. Wang, X. Shen, X. Yin, B. Mao, and L. Xie. IntFinder: Automatically Detecting Integer Bugs in x86 Binary Program. In *Proceedings of the International conference on Information and Communications Security (ICICS)*, 2009.

[42] P. Chen, Y. Wang, and Z. Xin. Brick: A Binary Tool for Run-time Detecting and Locating Integer-based Vulnerability. In *Proceedings of the International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2009.

[43] X. Cheng, M. Zhou, X. Song, M. Gu, and J. Sun. Automatic Fix for C Integer Errors by Precision Improvement. In *Proceedings of the Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 2–11. IEEE, 2016.

[44] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2014.

[45] R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya. ARCHERR: Runtime Environment Driven Program Safety. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 2004.

[46] V. Chipounov, V. Kuznetsov, and G. Candea. S2E: A Platform for in-vivo Multi-path Analysis of Software Systems. *SIGPLAN Notices*, 46(3):265–278, 2011.

[47] Chromium. CVE-2016-1612: Bug Description and Reward, 2016. `https://goo.gl/9SxjEA`.

[48] Chromium. Octane Browser Benchmark, 2017. `https://chromium.github.io/octane/`.

[49] Clang. Clang SafeStack. `https://clang.llvm.org/docs/SafeStack.html`.

[50] Clang. Clang/LLVM Compiler Framework. `https://clang.llvm.org/`.

[51] Clang. Clang 3.9 Documentation - Control Flow Integrity, 2017. `https://goo.gl/gnmoHU`.

[52] Clang. Clang-CFI Cast Checker Metadata, 2017. `https://goo.gl/JkGDjL`.

[53] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2004.

[54] Z. Coker and M. Hafiz. Program Transformations to fix C Integers. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 792–801. IEEE/ACM, 2013.

[55] M. Conti, P. Larsen, S. Crane, L. Davi, M. Franz, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[56] M. Corp. PREfast analysis tool. `https://msdn.microsoft.com/en-us/library/ms933794.aspx`.

[57] C. Cowan, S. Beatie, J. Johansen, and P. Wagle. PointguardTM: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the Conference on USENIX Security Symposium (SSYM)*. ACM, 2003.

[58] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the Conference on Security Symposium (SSYM)*. ACM, 1998.

[59] S. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a TRaP: Table Randomization and Protection against Function-Reuse Attacks. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[60] C. Crispin, P. Wagler, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade*. *DARPA Discex*, 2000.

[61] J. Criswell, N. Dautenhahn, and V. Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2014.

[62] T. Dang, D. Wagner, and P. Maniatis. *Towards Improved Mitigations for Two Attacks on Memory Safety*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2017.

[63] T. H. Y. Dang, P. Maniatis, and D. Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2015.

[64] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Return-Oriented Programming without Returns on ARM. Technical report, No. HGI-TR-2010-002, Ruhr-University Bochum.

[65] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin. HAFIX: Hardware-assisted Flow Integrity Extension. In *Proceedings of the Design Automation Conference (DAC)*, 2015.

[66] L. Davi, P. Koeberl, and A.-R. Sadeghi. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *Proceedings of the Design Automation Conference (DAC)*, 2014.

[67] L. Davi, A.-R. Sadeghi, and M. Winady. ROPdefender: A Detection Tool to Defend Against Return-Oriented Programming Attacks. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2011.

[68] L. de Moura and N. Bjørner. Z3: an Efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS/ETAPS)*. Springer, 2008.

[69] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT. In *Proceedings of the International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA)*. ACM, 2014.

[70] B. Demsky and M. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 2003.

[71] D. Dewey and J. Giffin. Static Detection of C++ VTable Escape Vulnerabilities in Binary Code. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2012.

[72] W. Dietz, P. Li, J. Regehr, and V. Adve. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. In *Empirical Software Engineering*. Springer, 2005.

[73] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding Integer Overflow in C/C++. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 25, page 2. ACM, 2015.

[74] R. Ding, C. Qian, C. Song, W. Harris, T. Kim, and W. Lee. Efficient Protection of Path-Sensitive Control Security. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX, 2017.

[75] Dromaeo. Dromaeo Browser Benchmark, 2017. ˙`http://dromaeo.com/?v8`.

[76] T. Dullien. Turing completeness, weird machines, Twitter, and muddled terminology, 2018. ˙`http://addxorrol.blogspot.com/2018/10/`.

[77] Eclipse. Eclipse CDT. ˙`https://eclipse.org/cdt/`.

[78] M. Elsabagh, D. Fleck, and A. Stavrou. Strict Virtual Call Integrity Checking for C ++ Binaries. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2017.

[79] D. B. Emery. HeapShield: Library-Based Heap Overflow Protection for Free. Technical report, No. 06-28, University of Massachusetts, Amherst, Computer Science Dept., 2006.

[80] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskosr. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[81] A. Fokin, Y. Derevenets, A. Chernov, and K. Troshina. SmartDec: Approaching C++ decompilation. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, 2011.

[82] G. Foundation. The GNU Compiler Collection. ˙`https://gcc.gnu.org//`.

[83] I. Fratric. Runtime Prevention of Return-Oriented Programming Attacks. ˙`http://ropguard:googlecode:com/svn-history/r2/trunk/doc/ropguard:pdf`.

[84] E. Gamma, J. Vlissides, R. Johnson, and R. Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[85] V. Ganesh, T. Leek, and M. Rinard. Taint-based Directed Whitebox Fuzzing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 474–484. IEEE/ACM, 2009.

[86] GCC. GCC's shadow stack implementation. ˙`https://gcc.gnu.org/ml/gcc/2016-04/msg00083.html`.

[87] GCC. Discussion Between Programmers and GCC Developers, 2009. ˙`http://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475#c2`.

[88] GCC. shadow stack proposal, 2016. `https://gcc.gnu.org/ml/gcc/2016-04/msg00083.html`.

[89] X. Ge, W. Cui, and T. Jaeger. GRIFFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2017.

[90] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *Proceedings of the European Symposium on Security and Privacy (Euro S&P)*. IEEE, 2016.

[91] X. Ge, N. Talele, M. Payer, and T. Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *Proceedings of the European Symposium on Security and Privacy (Euro S&P)*, 2016.

[92] P. Godefroid. Random Testing for Security: Blackbox vs. Whitebox Fuzzing. In *Proceedings of the International Workshop on Random Testing (RT)*, pages 1–1. ACM, 2007.

[93] P. Godefroid and M. Y. Levin. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2008.

[94] E. Goktas, A. Oikonomopoulos, R. Gawlik, B. Kollenda, E. Athanasopoulos, C. Giuffrida, G. Portokalidis, and H. Bos. Bypassing Clang's SafeStack for Fun and Profit. In *Blackhat Europe*, 2016. `https://goo.gl/zKMHzs`.

[95] Google. CVE-2016-1612: Differential View, 2016. `https://codereview.chromium.org/1531583005/diff/1/src/ic/ic.cc`.

[96] Google. CVE-2017-3106: Object Type Confusion in Adobe F. Player v. 26.0.0.137, 2017. `https://goo.gl/gakD25`.

[97] Google. Google Chrome, 2017. `https://www.chromium.org/`.

[98] Google. The Chromium Projects, Chromium, 2017. `https://goo.gl/uE486n`.

[99] Google. Undefined Behavior Sanitizer, 2017. `https://goo.gl/ELrNKj`.

[100] Google. Google's OSS-Fuzz, 2018. `https://opensource.google.com/projects/oss-fuzz`.

[101] J. Gray. C++: Under the Hood, 1994. `http://www.openrce.org/articles/files/jangrayhood.pdf`.

[102] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proceedings of the Conference on Data and Application Security and Privacy (CODASPY)*. ACM, 2017.

[103] Z. Gu, E. T. Barr, D. J. Hamilton, and Z. Su. Has the bug really been fixed? In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2010.

[104] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2014.

[105] H. M. Haddad and H. Shahriar. Rule-Based Source Level Patching of Buffer Overflow Vulnerabilities. In *Proceedings of the International Conference on Information Technology: New Generations (ITNG)*. ACM, 2013.

[106] I. Haller, E. Goktas, E. Athanasopoulos, G. Portokalidis, and H. Bos. ShrinkWrap: VTable Protection Without Loose Ends. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2015.

[107] I. Haller, Y. Jeon, H. Peng, M. Payer, and C. Giuffrida. TypeSan: Practical Type Confusion Detection. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[108] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowser: A Guided Fuzzer to Find Buffer Overflow Vulnerabilities. In *Proceedings of the Conference on Security (USENIX Security)*, pages 49–64. USENIX, 2013.

[109] M. J. Harrold, F. Steinmann, F. Tip, and A. Zeller. Fault Prediction, Localization, and Repair. *Dagstuhl Seminar 13061*, February 2013.

[110] A. Ibing. Parallel SMT-constrained symbolic execution for Eclipse CDT/Codan. In *Proceedings of the International Conference on Testing Software and Systems (ICTSS)*, 2013.

[111] A. Ibing. SMT-Constrained Symbolic Execution for Eclipse CDT/Codan. In *Proceedings of the International Conference on Software Engineering and Formal Methods (WS-FMDS)*. Springer, 2013.

[112] A. Ibing. Path-Sensitive Race Detection with Partial Order Reduced Symbolic Execution. In *Proceedings of the International Conference on Software Engineering and Formal Methods (WS-FMDS)*. Springer, 2014.

[113] A. Ibing. A Fixed-Point Algorithm for Automated Static Detection of Infinite Loops. In *Proceedings of the International Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 2015.

[114] A. Ibing. Architecture Description Language based Retargetable Symbolic Execution. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 2015.

[115] A. Ibing. Symbolic Execution Based Automated Static Bug Detection for Eclipse CDT. In *International Journal on Advances in Security*, 2015.

[116] IBM. ProPolice, 2005. `https://web.archive.org/web/20070212032750/http://wiki.x.org/wiki/ProPolice`.

[117] Intel. Intel Processor Trace. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`.

[118] Intel. Control-flow Enforcement Technology (CET), 2016. `https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf`.

[119] ISO/IEC. Working Draft, Standard for Programming Language C++ N4618, 2016. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4618.pdf`.

[120] ISO/IEC. Working Draft, Standard for Programming Language C++, N4296, 2018. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf`.

[121] K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer. Block Oriented Programming: Automating Data-Only Attacks, 2017. `https://arxiv.org/abs/1805.04767`.

[122] Itanium. C++ ABI. `https://mentorembedded.github.io/cxx-abi/abi.html`.

[123] M. Jacobs and E. C. Lewis. SMART C: A Semantic Macro Replacement Translator for C. In *Proceedings of the Source Code Analysis and Manipulation Working Conference (SCAM)*, 2006.

[124] D. Jang, T. Tatlock, and S. Lerner. SafeDispatch: Securing C++ Virtual Calls from Memory Corruption Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2014.

[125] Y. Jeon, P. Biswas, S. A. Carr, B. Lee, and M. Payer. HexType: Efficient Detection of Type Confusion Errors for C++. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[126] JetStream. JetStream Browser Benchmark, 2017. `http://browserbench.org/JetStream/`.

[127] E. JFace. JFace. `https://wiki.eclipse.org/JFace`.

[128] E. Johnson, T. Zhao, and J. Criswell. Poster: CRAFTED: Code Reuse Analysis for Trusted and Effective Defense. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2017.

[129] I. JTC1/SC22WG21. ISO/IEC 14882:2013 Programming Language C++ (N3690), 2013. `https://isocpp.org/files/papers/N3690.pdf`.

[130] U. Khedker, A. Sanyal, and B. Sathe. *Data flow analysis: Theory and Practice*. CRC Press, 2009.

[131] U. P. Khedker, A. Sanyal, and B. Karkare. *Data Flow Analysis*. CRC Press, 2009.

[132] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2013.

[133] T. Kornau. Return-Oriented Programming for the ARM Architecture. 2009. `http://www.zynamics.com/downloads/kornau-tim--diplomarbeit--rop.pdf`.

[134] Kraken. Kraken JavaScript Benchmark, 2017. `https://krakenbenchmark.mozilla.org/`.

[135] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2014.

[136] A. Labs. Libsafe, 2001. `https://directory.fsf.org/wiki/Libsafe`.

[137] B. Lan, Y. Li, H. Sun, C. Su, Y. Liu, and Q. Zeng. Loop-Oriented Programming: A New Code Reuse Attack to Bypass Modern Defenses. In *Proceedings of International Conference On Trust, Security And Privacy In Computing And Communications (Trustcom)*. IEEE, 2015.

[138] A. Laskavaia. Codan- C/C++ static analysis framework for CDT. In *EclipseCon*, 2011.

[139] V. Le, M. Afshari, and Z. Su. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI)*. ACM, 2014.

[140] D. Le Berre and A. Parrain. The SAT4J library, 2017. `http://www.sat4j.org/`.

[141] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. In *Transactions on Software Engineering (TSE)*, 2012.

[142] B. Lee, C. Song, T. Kim, and W. Lee. Type Casting Verification: Stopping an Emerging Attack Vector. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX, 2015.

[143] J. Lettner, B. Kollenda, A. Homescu, P. Larsen, F. Schuster, L. Davi, A.-R. Sadeghi, T. Holz, and M. Franz. Subversive-C: Abusing and Protecting Dynamic Message Dispatch. In *Proceedings of the Annual Technical Conference (USENIX ATC)*. USENIX, 2016.

[144] Lighttpd. Lighthttpd, 2017. `https://www.lighttpd.net/`.

[145] Z. Lin. LibsafeXP: A Practical and Transparent Tool for Run-time Buffer Overflow Preventions. In *Proceedings of the Annual Information Assurance Workshop (IAW)*. IEEE, 2006.

[146] Z. Lin, X. Jiang, D. Xu, B. Mao, and L. Xie. AutoPaG: Towards Automated Software Patch Generation with Source Code Root Cause Identification and Repair. In *Proceedings of the Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2007.

[147] Z. Lin, X. Zhang, and D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2010.

[148] C. Liu, J. Yang, L. Tan, and M. Hafiz. R2Fix: Automatically Generating Bug Fixes from Bug Reports. In *Proceedings of the Intertional Conference on Software Testing, Validation and Verification (ICST)*. ACM, 2013.

[149] LLVM. Clang CFI, 2017. `https://goo.gl/W7aMF9`.

[150] LLVM. LLVM link time optimization: Design and implementation. 2017. `https://goo.gl/r3RH2U`.

[151] LLVM. LLVM Team, The LLVM compiler infrastructure project, 2017. `http://llvm.org/`.

[152] LLVM. The LLVM Compiler Infrastructure, 2017. `https://llvm.org/`.

[153] LLVM. The LLVM Gold Plugin, 2017. `https://goo.gl/UjFxih`.

[154] F. Long, S. Sidiroglou-Douskos, D. Kim, and M. Rinard. Sound Input Filter Generation for Integer Overflow Errors. In *SIGPLAN Notices*, volume 49, pages 439–452. ACM, 2014.

[155] E. LTK. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. ˙https://eclipse.org/articles/Article-LTK/ltk.html.

[156] D. Lucas, A. Dmitrienko, M. Egele, T. Fischer, T. Holz, R. Hund, S. Nurnberger, and A.-R. Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2012.

[157] M. Maiffret and R. Permeh. Code Red, 2001. https://en.wikipedia.org/wiki/Code_Red_(computer_worm).

[158] R. Majumdar and K. Sen. Hybrid Concolic Testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 416–426. IEEE/ACM, 2007.

[159] P. Marinescu and C. Cadar. Make Test-zesti: A Symbolic Execution Solution for Improving Regression Testing. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 716–726. IEEE/ACM, 2012.

[160] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières. CCFI: Cryptograhically Enforced Control Flow Integrity. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[161] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2015.

[162] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2016.

[163] Memcached. Memcached a General-purpose Distributed Memory Caching System, 2017. ˙https://memcached.org/.

[164] Microsoft. Changes to Functionality in Microsoft Windows XP Service Pack 2. ˙https://technet.microsoft.com/en-us/library/bb457151.aspx.

[165] Microsoft. The STRIDE Threat Model, 2009. ˙https://docs.microsoft.com/en-us/previous-versions/commerce-server/ee823878(v=cs.20)?redirectedfrom=MSDN.

[166] Microsoft. PREfast analysis tool. In *Microsoft Corporation*, 2010.

[167] Microsoft. Windows Control Flow Guard, 2015. ˙http://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx.

[168] Microsoft. Microsoft Mitigation Bypass Bounty and Bounty for Defense Program , 2017. ˙https://technet.microsoft.com/en-us/security/dn425049.aspx.

[169] Microsoft. SafeInt, 2018. ˙http://safeint.codeplex.com/.

[170] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE'19, pages 1187–1198, Montreal, Quebec, Canada, 2019.

[171] MITRE. 2011 CWE/SANS Top 25. `http://cwe.mitre.org/top25/.

[172] MITRE. CWE-121. `http://cwe.mitre.org/data/definitions/121.html.

[173] MITRE. Heartbleed Bug. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[174] MITRE. Integer Coercion Error. `https://cwe.mitre.org/data/definitions/192.html.

[175] MITRE. Integer Underflow (Wrap or Wraparound). `https://cwe.mitre.org/data/definitions/191.html.

[176] MITRE. IO2BO: Integer Overflow to Buffer Overflow. `https://cwe.mitre.org/data/slices/680.html.

[177] MITRE. Mitre's Common Weakness Enumeration (CWE). `https://cwe.mitre.org/.

[178] MITRE. Numeric Truncation Error. `https://cwe.mitre.org/data/definitions/197.html.

[179] MITRE. Off-by-one Error. `http://cwe.mitre.org/data/definitions/193.html.

[180] MITRE. Signed to Unsigned Conversion Error. `https://cwe.mitre.org/data/definitions/195.html.

[181] MITRE. Unexpected Sign Extension. `https://cwe.mitre.org/data/definitions/194.html.

[182] MITRE. Unsigned to Signed Conversion Error. `https://cwe.mitre.org/data/definitions/196.html.

[183] MITRE. Mitre Corporation, CVE-2002-0639: Integer overflow in sshd in OpenSSH. 2002. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-0639.

[184] MITRE. Mitre Corporation, CVE-2002-1490: Integer overflow in NetBSD 1.4. 2002. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1490.

[185] MITRE. CVE-2010-2753: Integer overflow in Mozilla Firefox, Thunderbird and SeaMonkey. 2010. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-2753.

[186] MITRE. CWE/SANS Top 25 Most Dangerous Software Errors. 2011. `https://cwe.mitre.org/top25/.

[187] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz. Opaque Control-Flow Integrity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2015.

[188] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX, 2009.

[189] A. Moneger. Stitching numbers-Generating ROP payloads from in memory numbers, 2014. `http://bofh.nikhef.nl/events/defcon/DEF%20CON%2022/DEF%20CON%2022%20presentations/Alexandre%20Moneger%20-%20Updated/DEFCON-22-Alex-Moneger-Stitching-numbers-Generating-ROP-payloads-from-in-memory-numbers.pdf`.

[190] M. Monperrus. A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches: Essay on the Problem Statement and the Evaluation of Automatic Software Repair. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2014.

[191] M. Monperrus. Automatic Software Repair: a Bibliography. *Computing Surveys (CSUR)*, 51:1–24, 2017. `https://hal.archives-ouvertes.fr/hal-01206501/file/survey-automatic-repair.pdf`.

[192] R. T. Morris. Morris Worm, 1988. `https://en.wikipedia.org/wiki/Morris_worm`.

[193] Y. Moy, N. Bjørner, and D. Sielaff. Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis. Technical report, MSR-TR-2009-57, 2009.

[194] P. Muntean, C. Eckert, and A. Ibing. Context-Sensitive Detection of Information Exposure Bugs with Symbolic Execution. In *Innovative Software Development Methodologies and Practices (InnoSWDev)*. ACM, 2014.

[195] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert. τFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *Proceedings of the International Symposium on Research in Attacks, Intrusions, and Defenses (RAID)*. Springer, 2018.

[196] P. Muntean, V. Kommanapali, A. Ibing, and C. Eckert. Automated Generation of Buffer Overflow Quick Fixes Using Symbolic Execution and SMT. In *Proceedings of the International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*. Springer, 2015.

[197] P. Muntean, M. Monperrus, H. Sun, J. Grossklags, and C. Eckert. IntRepair: Informed Repairing of Integer Overflows . In *Transactions on Software Engineering (TSE)*. IEEE, 2019.

[198] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert. Analyzing Control Flow Integrity with LLVM-CFI . In *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2019.

[199] P. Muntean, M. Neumayer, Z. Lin, G. Tan, J. Grossklags, and C. Eckert. ρFEM: Efficient Backward-edge Protection Using Reversed Forward-edge Mappings. In *Annual Computer Security Applications Conference (ACSAC)*. ACM, 2020.

[200] P. Muntean, M. Rahman, A. Ibing, and C. Eckert. SMT-constrained symbolic execution engine for integer overflow detection in C code. In *Proceedings of the Information Security for South Africa (ISSA)*. IEEE, 2015.

[201] P. Muntean, S. Wuerl, J. Grossklags, and C. Eckert. CastSan: Efficient Detection of Polymorphic C++ Object Type Confusions with LLVM. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 2018.

[202] A. Mycroft. Lecture Notes, 2007. ˙https://goo.gl/F7tUZj.

[203] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI)*. ACM, 2009.

[204] Nginx. Nginx web server, 2017. ˙https://nginx.org/en/.

[205] H. D. T. Nguyen, A. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program Repair via Semantic Analysis. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2013.

[206] NIST. Juliet Test Suite v1.2 for C/C++. ˙http://samate.nist.gov/SRD/testsuites/juliet/Juliet_Test_Suite_v1.2_for_C_Cpp.zip.

[207] NIST. Juliet Test Suite v1.2 for C/C++. 2017. ˙http://samate.nist.gov/SARD/testsuites/juliet/Juliet_Test_Suite_v1.2_for_C_Cpp.zip.

[208] NIST. Type Confusions Reported by U.S. NIST from Jan. 2008 to Sept. 2017. 2017. ˙https://nvd.nist.gov/vuln/search/results?adv_search=false&form_type=basic&results_type=overview&search_type=all&query=type+confusion.

[209] B. Niu and G. Tan. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2013.

[210] B. Niu and G. Tan. Modular Control-Flow Integrity. In *Proceedings of the Symposium on Programming Language Design and Implementation (PLDI)*. ACM, 2014.

[211] B. Niu and G. Tan. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Inegrity. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2014.

[212] B. Niu and G. Tan. Per-Input Control-Flow Integrity. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[213] NodeJS. NodeJS, 2017. ˙https://nodejs.org/en/.

[214] A. One. Smashing the Stack for Fun and Profit, 1996. ˙http://phrack.org/issues/49/14.html.

[215] OWASP. Buffer Overflows article on OWASP, 2017. ˙https://www.owasp.org/index.php/Buffer_Overflows.

[216] V. Pappas, M. Polychronakis, and A. D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX, 2013.

[217] T. Parr. *Language Implementation Patterns: Techniques for Implementing Domain-Specific Languages*. O'Reilly UK Ltd.; Edition: 1, 2010.

[218] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida. MARX: Uncovering Class Hierarchies in C++ Programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2017.

[219] PaX. Team, Address Space Layout Randomization (ASLR), 2001. `https://pax.grsecurity.net/docs/aslr.txt`.

[220] M. Payer. *Software Security: Principles, Policies, and Protection*. HexHive Books, 0.32 edition, May 2018.

[221] M. Payer, A. Barresi, and T. R. Gross. Fine-Grained Control-Flow Integrity through Binary Hardening. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. Springer, 2015.

[222] F. M. Q. Pereira, R. E. Rodrigues, and V. H. S. Campos. A Fast and Low-overhead Technique to Secure Programs Against Integer Overflows. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*. IEEE/ACM, 2013.

[223] T. Petsios, V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis. DynaGuard: Armoring Canary-based Protections against Brute-force Attacks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2015.

[224] J. Pewny and T. Holz. Control-flow Restrictor: Compiler-based CFI for iOS. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2013.

[225] M. Pomonis, T. Petsios, K. Jee, M. Polychronakis, and A. D. Keromytis. IntFlow: Improving the Accuracy of Arithmetic Error Detection Using Information Flow Tracking. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 416–425. ACM, 2014.

[226] A. Prakash, X. Hu, and H. Yin. Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2015.

[227] A. Prakash and H. Yin. Defeating ROP Through Denial of Stack Pivot. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2015.

[228] X. Qu and B. Robinson. A Case Study of Concolic Testing Tools and Their Limitations. In *Proceeding of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 117–126. IEEE, 2011.

[229] G. Ramalingam. The Undecidability of Aliasing. In *Proceedings of the Transactions on Programming Languages and Systems (TOPLAS)*. ACM, 1994.

[230] Raykoid666. The Exploitant-Security Info and Tutorials, 2009. `https://raykoid666.wordpress.com/`.

[231] Redis. Redis In-memory Database, 2017. `https://redis.io/`.

[232] J. G. J. Rossie and D. P. Friedman. An Algebraic Semantics of Subobjects. In *Proceedings of the Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 1995.

[233] J. Salwan. ROPgadget - Gadgets Finder and Auto-roper, 2011. ˙`http://shell-storm.org/project/ROPgadget/`.

[234] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. VTPin: Practical VTable Hijacking Protection for Binaries. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2016.

[235] D. Sarkar, M. Jagannathan, J. Thiagarajan, and R. Venkatapathy. Flow-insensitive static analysis for detecting integer anomalies in programs. In *Proceedings of the Conference on IASTED International Multi-Conference, ACTA Press*, 2007.

[236] R. Sauciuc and G. Necula. Reverse Execution With Constraint Solving. Technical report, No. UCB/EECS-2011-67, EECS Department, University of California, Berkeley, May 2011.

[237] S. Schirra. Ropper, 2017. ˙`https://github.com/sashs/Ropper`.

[238] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2015.

[239] F. Schuster, T. Tendyck, J. Pewny, A. Tendyck, M. Steegmanns, M. Contag, and T. Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Springer, 2014.

[240] R. C. Seacord. Addison Wesley Professional, 2008.

[241] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Software Engineering Notes*, volume 30, pages 263–272. ACM, 2005.

[242] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (On the x86). In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2007.

[243] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. Technical report, No. 10012, New York University. Courant Institute of Mathematical Sciences. Computer Science Department, 1978.

[244] A. Shaw, D. Dogget, and M. Hafiz. Automatically Fixing C Buffer Overflows Using Program Transformations. In *Proceedings of the Conference on Dependable Systems and Networks (DSN)*. IEEE/IFIP, June 2013.

[245] S. Shiraishi, V. Mohan, and H. Marimuthu. Quantitative Evaluation of Static Analysis Tools. In *Proceedings of the Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2014.

[246] S. Sidiroglou, G. Giovanidis, and A. D. Keromytis. A Dynamic Mechanism for Recovering from Buffer Overflow Attacks. In *Proceedings of the International Conference on Information Security (ISC)*. ACM, 2005.

[247] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, P. Piselli, and M. Rinard. Automatic Error Elimination by Multi-application Code Transfer. Technical report, No. MIT-CSAIL-TR-2014-024, 2014.

[248] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard. Targeted Automatic Integer Overflow Discovery Using Goal-directed Conditional Branch Enforcement. In *SIGPLAN Notices*, volume 50, pages 473–486. ACM, 2015.

[249] Sidiroglou-Douskos, S. and Lahtinen, E. and Rinard, M. Automatic Discovery and Patching of Buffer and Integer Overflow Errors. Technical Report No. MIT-CSAIL-TR-2015-018, Massachusetts Institute of Technology, 2015.

[250] A. Smirnov and T. Chiueh. Automatic Patch Generation for Buffer Overflow Attacks. In *Proceedings of the Third International Symposium on Information Assurance and Security (IAS)*, pages 165–170, 2007.

[251] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Proceedings of the European Symposium on Security and Privacy (Euro S&P)*. IEEE, 2013.

[252] SPEC. Standard Performance Evaluation Corporation, SPEC CPU 2006. 2017. `https://goo.gl/NtmYy8`.

[253] H. Sun, C. Su, Y. Wang, and Q. Zeng. Improving the Accuracy of Integer Signedness Error Detection Using Data Flow Analysis. *International Journal of Software Engineering and Knowledge Engineering*, 25:1573–1593, 2015.

[254] H. Sun, X. Zhang, C. Su, and Q. Zeng. Efficient Dynamic Tracking Technique for Detecting Integer-overflow-to-buffer-overflow Vulnerability. In *Proceedings of the Information, Computer and Communications Security (CCS)*, pages 483–494. ACM, 2015.

[255] H. Sun, X. Zhang, Y. Zheng, and Q. Zeng. IntEQ: Recognizing benign integer overflows via equivalence checking across multiple precisions. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 1051–1062. IEEE/ACM, 2016.

[256] Sunspider. SunSpider 1.0.2 JavaScript Benchmark, 2017. `https://webkit.org/perf/sunspider/sunspider.html`.

[257] G. Tan and T. Jaeger. CFG Construction Soundness in Control-Flow Integrity. In *Proceedings of the Programming Languages and Analysis for Security Conference (PLAS)*, 2017.

[258] M. Theodorides. Breaking active-set backward-edge cfi. Technical report, 2017. `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-78.html`.

[259] M. Theodorides and D. Wagner. Breaking Active-Set Backward-Edge CFI. In *Proceedings of the International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2017.

[260] M. Theodorides and D. Wagner. Breaking Active-Set Backward-Edge CFI. In *Proceedings of the International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2017.

[261] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX, 2014.

[262] F. Tip. A survey of program slicing techniques. In *Journal of Programming Languages*, 1995.

[263] F. Tip, J. D. Choi, J. Field, and G. Ramalingam. Slicing Class Hierarchies in C++. In *Proceedings of the Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 1996.

[264] U.S. National Institute of Standards and Technology (NIST). Juliet Test Suite v1.2 for C/C++. ˙`https://samate.nist.gov/SRD/testsuite.php`.

[265] Valgrind. Valgrind Home, 2018. ˙`http://valgrind.org/`.

[266] V. v. d. Veen, D. Andriesse, E. Göktas, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical Context-Sensiticve CFI. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2015.

[267] V. v. d. Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida. The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[268] V. v. d. Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A Tough call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2016.

[269] S. Wang, P. Wang, and D. Wu. Reassembleable Disassembling. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 627–642, Washington, D.C., 2015.

[270] T. Wang, C. Song, and W. Lee. Diagnosis and Emergency Patch Generation for Integer Overflow Exploits. In *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 255–275. Springer, 2014.

[271] T. Wang, T. Wei, G. Gu, and W. Zou. TaintScope: A Checksum-aware Directed Fuzzing tool for Automatic Software Vulnerability detection. In *Proceedings of the Symposium on Security and Privacy (S&P)*, pages 497–512. IEEE, 2010.

[272] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2009.

[273] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving Integer Security for Systems with KINT. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, volume 12, pages 163–177. USENIX, 2012.

[274] Z. Wang and X. Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2010.

[275] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, Raleigh, NC, October 2012.

[276] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin. Securing Untrusted Code via Compiler-Agnostic Binary Rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*, Orlando, FL, December 2012.

[277] W. Weimer. Patches as better bug reports. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*. ACM, 2006.

[278] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2009.

[279] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kermelis, J. Yang, and W. Aiello. Shuffler: Fast and Deployable Continous Code Re-Randomization. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 2016.

[280] R. Wojtczuk. UQBTng: A Tool Capable of Automatically Finding Integer Overflows in Win32 Binaries. *Chaos Communication Congress (22C3)*, 2005.

[281] P. Wollgast, R. Gawlik, B. Garmany, B. Kollenda, and T. Holz. Automated Multi-architectural Discovery of CFI-Resistant Code Gadgets. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 2016.

[282] xLab. Return Flow Guard, 2016. `http://xlab.tencent.com/en/2016/11/02/return-flow-guard/`.

[283] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin. CONFIRM: Evaluating Compatibility and Relevance of Control-flow Integrity Protections for Modern Software. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019.

[284] P. Yuan, Q. Zeng, and X. Ding. Hardware-Assisted Fine-Grained Code-Reuse Attack Detection. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Springer, 2015.

[285] B. Zhang, C. Feng, B. Wu, and C.Tang. Detecting Integer Overflow in Windows Binary Executables based on Symbolic Execution. *Proceedings of the International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2016.

[286] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. VTrust: Regaining Trust on Virtual Calls. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2016.

[287] C. Zhang, C. Song, K. C. Zhijie, Z. Chen, and D. Song. vTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. ISOC, 2015.

[288] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, pages 71–86. Springer, 2010.

[289] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2013.

[290] M. Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the Conference on Security (USENIX Security)*. USENIX, 2013.

[291] M. Zhang and R. Sekar. Control Flow and Code Integrity for COTS binaries: An Effective Defense Against Real-ROP Attcks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*. ACM, 2015.

[292] Y. Zhang, X. Sun, Y. Deng, L. Cheng, S. Zeng, Y. Fu, and D. Feng. Improving Accuracy of Static Integer Overflow Detection in Binary. In *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. Springer, 2015.

[293] M. Zhao, J. Grossklags, and P. Liu. An Empirical Study of Web Vulnerability Discovery Ecosystems. In *Proceedings of the Conference on Computer and Communications Security (CCS)*. ACM, 2015.