FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation in Informatik

Strengthened, Composable, and Quantifiable Software Integrity Protection

Mohsen Ahmadvand

FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Lehrstuhl IV - Software and Systems Engineering

Strengthened, Composable, and Quantifiable Software Integrity Protection

Mohsen Ahmadvand

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. Dr.-Ing. Georg Carle
Prüfer der Dissertation:	
1.	Prof. Dr. Alexander Pretschner
2.	Prof. Dr. Bjorn De Sutter

Die Dissertation wurde am 18.11.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 09.03.2021 angenommen.

München, den 7. Juni 2021 Mohsen Ahmadvand

Acknowledgments

I want to express my gratitude to my advisor, Prof. Alexander Pretschner, for his continuous encouragement and guidance throughout this venture. Alex, you were kind, honest, open, and very helpful to me. I want to thank Prof. Bjorn De Sutter (my second advisor) for his guide and support. Bjorn, your feedback and support helped me very much when I was very frustrated with my experiments. I want to thank Dr. Sebastian Banescu for his valuable feedback and help throughout my doctoral studies. My greatest appreciations go to my industrial research partners, Keith Ball, Dr. Antoine Scemama, and Hans Strack-Zimmermann, for their exceptional support. Folks, I was so lucky to have the chance to work with you. My sincerest gratitude goes to my friends, Tonya, Arash, Sergey, Alei, Amjad, Afshin, and my family.

Zusammenfassung

Man-At-The-End (MATE)-Angreifer sind allmächtige Gegner, gegen die es bislang keine universelle Gegenmaßnahme gibt. So gehört der Schutz der Programmintegrität zu den herausforderndsten Problemen im Zusammenhang mit MATE-Angreifern. In der Literatur ist ein breites Spektrum von Schutzmaßnahmen beschrieben, von denen jede einzelne die Widerstandsfähigkeit gegen bestimmte Angriffe auf spezielle Funktionalitäten von Programmen erhöht. In dieser Arbeit führen wir zunächst eine umfassende Studie durch, um verschiedene Integritätsschutz Systeme in der Literatur zu vergleichen und ihre Stärken und Schwächen zu bestimmen. Auf Basis der Ergebnisse unserer Studie schlagen wir neue Abwehrmechanismen und Kombinationen von Schutzmaßnahmen vor, um die Sicherheit und Anwendbarkeit dieser Konzepte zu verbessern. Obwohl das Zusammensetzen von Schutzmaßnahmen (im Gegensatz zur isolierten Anwendung) die Sicherheit intuitiv erhöhen kann, erzeugen diese Kombinationen neue Schwierigkeiten, deren Lösungsansatz ein nicht triviales Problem darstellt. Das Auflösen von Konflikten zwischen den Schutzmechanismen und die Suche nach optimalen Kombinationsmöglichkeiten stellen die Kernprobleme beim Zusammensetzen der Schutzmaßnahmen dar. In dieser Arbeit bewältigen wir diese Probleme, indem wir sie auf ein ganzzahliges, lineares Optimierungsproblem übertragen. Wir entwickeln einen Bezugsrahmen (sip-shaker) für das Zusammenstellen von Schutzmechanismen, der ein konfliktfreies Ineinandergreifen garantiert und gleichzeitig auf höhere Sicherheit und geringen, zusätzlich benötigten Aufwand optimiert. Um unsere Schutzmechanismen und Zusammenstellungen zu bewerten, führen wir Leistungs- und Sicherheitsanalysen mit Hilfe eines Datensatzes von Programmen aus der realen Welt durch (MiBench- und Kommandozeilen-Spiele). Da Konzepte zum Integritätsschutz lediglich die Angriffe erschweren, anstatt sie vollständig zu verhindern, ist es von entscheidender Bedeutung, die Widerstandsfähigkeit dieser Schutzmaßnahmen abzuschätzen. Der letzte Teil der Arbeit beschäftigt sich mit der Abschätzung der Widerstandsfähigkeit zusammengesetzter Schutzmaßnahmen gegen automatisierte Angriffe, die dem aktuellen Stand der Technik entsprechen. Um die Widerstandsfähigkeit zu quantifizieren, schlagen wir eine Abschätzung über zwei neuen Metriken vor: Lokalisierbarkeit (also die Wahrscheinlichkeit, Schutzmechanismen zu finden) und Überwindbarkeit (also die geschätzte Anzahl der Angriffsschritte um die Schutzmechanismen zu deaktivieren). Die Messgröße der Überwindbarkeit bietet ein Instrument, das stärkere Zusammenstellungen bevorzugt, die eine höhere Anzahl von Änderungen am Programmcode durch den Angreifer erfordern. Um die Lokalisierbarkeit von Schutzmaßnahmen zu beschreiben, entwickeln und vergleichen wir Lokalisierungsangriffe, die auf maschinellem Lernen und Programmspur-Analyse basieren, gegen die Zusammenstellungen von Techniken des Integritätsschutzes sowie der Obfuskation.

Abstract

Man-At-The-End (MATE) attackers are almighty adversaries against whom no silver-bullet countermeasure exists. Protecting program's integrity is amongst the challenging problems in the context of MATE attackers. A wide range of hardening measures was proposed in the literature, each of which adding resilience against certain attacks on particular assets of programs. In this thesis, we first conduct a comprehensive study to benchmark different integrity protection schemes in the literature to identify their strengths and shortcomings. Based on the outcome of the study, we propose new defenses and combinations (compositions) of protections to improve the security and applicability of schemes. Although composing protections (rather than applying them single-handedly) can intuitively improve the security, addressing induced challenges is a nontrivial problem. Resolving protection conflicts and finding optimal compositions are two perplexing steps in the protection composition. In this thesis, we tackle the composition problem by transferring it to an ILP optimization problem. We develop a composition framework (sip-shaker) that guarantees a conflict-free composition while optimizing for security and overhead. To evaluate our protections and compositions, we conduct performance and security evaluations using a data set of real-world programs (MiBench, and CLI games). Since integrity protection schemes harden against the attacks rather than completely neutralize them, it is vital to estimate how resilient the protections are against attacks. As the last part of the thesis, we set to estimate the resilience of composed protection against state-of-the-art automated attacks. We propose to quantify the resilience as a conjecture of two novel metrics: localizability (the probability of finding protections) and defeatability (the estimated attack steps to disable protections). The defeatability metric provides a means for favoring stronger compositions with a higher number of imposed patches on the attacker. To characterize the localizability of protections, we develop and benchmark machine-learning and taint-based localization attacks on compositions of integrity checking techniques and obfuscations.

Outline of the Thesis

Chapter 1: Introduction

This chapter presents an introduction to the topic and the fundamental issues addressed by this thesis. It discusses ideas, goals, and limitations of this work. The content of Section 1.2 (the motivating case study) was taken from a previously published work [9] co-authered by the thesis author.

Chapter 2: Taxonomy

After surveying the literature, we analyzed and further classified the reviewed articles. This chapter presents the outcome of our analysis encompassing a taxonomy for integrity protection, analysis of various correlations within the taxonomy, and, finally, identification of gaps in the literature. Parts of this chapter were previously publishd in a work [9] co-authored by the thesis author.

Chapter 3: Practical Integrity Protection with Oblivious Hashing

This chapter presents practical enhancements corresponding to fundamental problems of the Oblivious Hashing integrity protection technique, namely coverage, and input dependency. The entire content of this chapter was previously published in work co-authored by the thesis author [5].

Chapter 4: Virtualized Self-Checksumming

In this chapter, we present a novel approach to address the problems of self-checksumming integrity protection concerning applicability and composability. This chapter was previously published in a work [7] co-authored by the thesis author.

Chapter 5: Composition Framework

This chapter presents the design and implementation of our software integrity protection composition framework. The content of the chapter was previously published in work [8] co-authored by the thesis's author.

Chapter 6: Resilience

In this chapter, we present a methodology along with two metrics to estimate the resilience of (composed) integrity protection techniques. The resistance is captured as a conjecture of localizability and defeatability of protections w.r.t known automated attacks.

Chapter 7: Conclusions

This chapter presents the conclusions, lessons learned, future work, and the limitations of this thesis.

Appendix A: Brief Summaries of Selected Related Work

This chapter presents brief summaries of selected related work in the field of software integrity protection. This survey is the basis upon which we build a taxonomy for software integrity protection techniques.

Appendix B: Reproducibility Handbook

We made the entire protection toolchain, datasets, and scripts open source. This chapter capture the links, scripts, and manuals on how to run the experiments that were conducted throughout the thesis.

N.B.: *Multiple chapters of this dissertation are based on different publications authored or co-authored by the author of this dissertation. Such publications are mentioned in the short descriptions above. Due to the obvious content overlapping, quotes from such publications within the respective chapters are not marked explicitly. Bear in mind that the thesis author has the "lead author" role in all such publications.*

Contents

Part I.

Introduction and Structured Related Work

1. Introduction

This chapter presents an introduction to the topic and the fundamental issues addressed by this thesis. It discusses ideas, goals, and limitations of this work. The content of Section 1.2 (the motivating case study) was taken from a previously published work [9] co-authered by the thesis author.

1.1. Context

We live in a digital era; software systems surround us. We rely on these systems in almost all aspects of our lives. An increasing number of everyday objects exceedingly start to acquire the prefix of *smart-*, which refers to a broader range of functionalities enabled by a growing layer of software. Smart houses give remote control over household appliances to the owners. Voice-activated smart speakers and virtual AI assistants execute audio commands. On-demand TVs and streaming services (e.g., Netflix and Spotify) load proprietary data to clients' devices.

Advances in software technology have enabled software vendors to deliver more complex systems to web and mobile users. Consequently, advanced proprietary algorithms land in client applications more than before. Instant messaging and social media have introduced new forms of interactions (e.g., Snapchat supports strict retention policies, and reports screenshots to the message publishers). Online multiplayer games have enormous user bases (e.g., 125 million users played the Fortnite game in 2019)[1]. Users trust in applications with their private data and bank credentials. Besides conventional online banking applications and in-game purchases, crypto wallets are persisting tokens worth of thousands of dollars in client devices.

Some goods, such as cars, offer on-demand premium service activation. In this model, users can opt-in for or opt-out of premium services after the initial purchase. That is, depending on the purchased license, parts of the software components are activated or deactivated.

The current trend in software utilization has countless benefits for users and enormous profits for software vendors. IT companies are among the most profitable businesses in 21st century. Users benefit from a wider range of innovative software solutions. Companies develop solutions and rapidly deliver their products to users.

[1]https://techsprobe.com/top-10-most-popular-online-games-of-2019-most-played-games/

The software solutions are delivered to users as either executable programs for their devices or web applications rendered in their browsers. In both cases users get access to all or parts of programs artifacts (binaries, files, scripts, and libraries). The exposure of artifacts gives rise to several forms of conflict of interest between the involved entities. The conflicts include users vs. vendors, vendors vs. vendors, and users vs. users. Some users may attempt to get illegitimate access to the proprietary data accessible via applications. Similarly, users may seek ways to circumvent the licensing mechanism in applications to, for instance, get premium features for free. These two cases indicate exemplary ways in which users can act against vendors.

On the other hand, given the access to the artifacts, some rogue vendors may attempt to reverse engineer advanced IP algorithms developed by their competitors. The consequences of a successful extraction of an IP algorithm can potentially cost a vendor their business.

Users may have incentives to impose unauthorized modifications on a software. Obtaining advantages over other users of the software is a common motive. Such actions can harm other users that can readily backfire on the software owners causing immense reputation damages. Some Snapchat users may wish to secretly take screenshots from other users' posts. Players of online multiplayer games may want to gain dishonest advantages over other players by modifying a specific behavior of the game. Malicious users may attempt to steal crypto tokens from other users' digital wallets. These three cases present samples in which users' interest conflicts with other users. Insufficient or improper mitigations of threats of the like can hurt the user base and lead to substantial financial losses for a company. Therefore, it is an essential responsibility of companies to take protective measures against likely perpetrators.

Software misuse harms are not limited to IP, users' privacy, or financial loss. Nowadays, critical infrastructures such as power plant systems depend on sensors and their software to stay functional. Thousands of sensors supply traffic control and transportation scheduling systems with the necessary data to make optimal decisions. Perpetrators with various motives, ranging from terrorism and sabotage to outrageous retaliations, can potentially attempt to alter the intended behavior of the systems putting human lives in danger.

The common denominator in all the reviewed threats is the inevitable exposure of software artifacts to untrustworthy users. The access shall be granted to such users as it is, in some cases, necessary to run the software on their commodities. This situation falls within an attacker model referred to as Man-At-The-End (MATE). MATE leaves us with no choice but to seek measures to fortify software. In this thesis, we seek to find effective means to protect software systems against MATE attackers.

To battle tampering attacks, academia and industry have considered trusted hardware. Intel SGX [69], ARM TrustZone [141] are examples of such hardware that provide a degree of integrity protection. At the time that we are writing this thesis, IoT devices and sensors seem to be still far from having such hardware [60]. The existing hardware comes with a set of limitations. Besides side-channel attacks [35, 91, 120], for programs to benefit from these technologies, practitioners need to partition their programs into trusted and untrusted regions. Interactions between the two partitions impose expensive context switches and

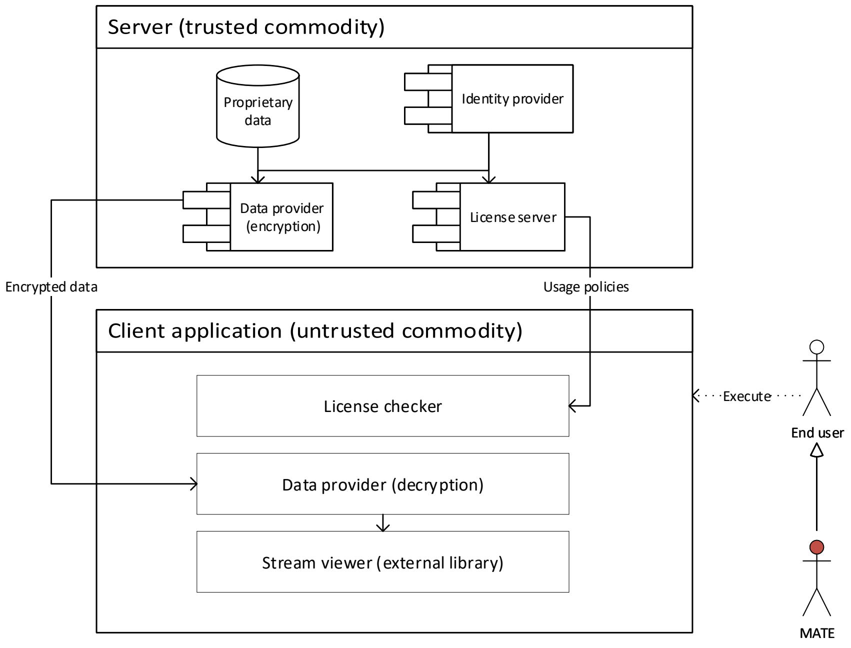

Server (trusted commodity)
Proprietary data
Identity provider
Data provider (encryption)
License server
Encrypted data
Usage policies
Client application (untrusted commodity)
License checker
Data provider (decryption)
Stream viewer (external library)
Execute
End user
MATE

Figure 1.1.: Architecture of a example DRM system to lend proprietary content.

therefore need to be minimized. That is, coping with the problem of context switches raises challenges in terms of performance and security [125]. Contents of secure partitions are limited as the program code dealing with inputs and outputs (system calls) has to reside in the untrusted region inevitably. The interface to the trusted partition needs to be protected as well. For this purpose, adding an extra layer of protection using software-based approaches appears to be a viable option. We believe that software-based protections are and will be relevant until the mentioned problems are solved. That is why we focus on software-based integrity protection in this thesis.

1.2. Motivating Case Study

In this section, we present a simplified digital rights management (DRM) system as a motivating example for the thesis. DRM enables authors to protect their proprietary digital data while sharing them with other people. To do so, DRM enforces a set of *usage policies* at the user end (client-side) that limits what users can actually do with the protected content.

The goal of our sample DRM system is to enable secure lending of proprietary content to end-users. For the sake of simplicity, we assume that two usage policies are employed in this system: users can lease a protected media (digital content) **i)** for a certain period of time or **ii)** for a particular number of views.

The system, as depicted in Figure 1.1, is comprised of two compartments: a server (which

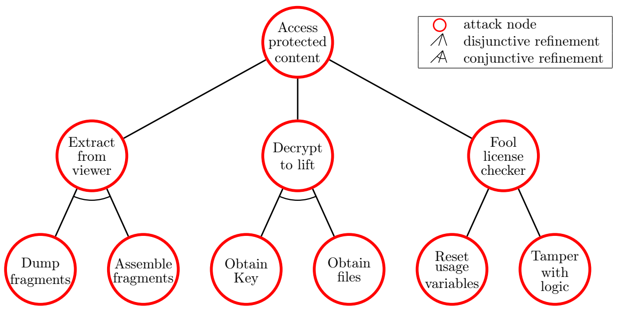


Figure 1.2.: Attack tree capturing potential MATE attacks on the sample DRM system.

runs on trusted commodity[2]) and a client (which runs on untrusted commodity and hence is exposed to MATE attacks). On the server side, three components, viz. `license server`, `data provider (encryption)` and `proprietary data`, are collaborating to deliver the desired functionality. On the client side, three components, viz. `license checker`, `data provider (decryption)` and `stream viewer`, are handling user requests.

The `license server` is in charge of generating usage policies according to the purchased license by the users. It sends usage policies to client applications. The `data provider` and `proprietary database` work closely together. The `data provider` retrieves the requested protected content from the database and encrypts them with client keys. Instead of providing one encrypted file, the data provider breaks each file into chunks (fragments) and then encrypts them.

On the client-side, the `license checker` enforces the policies that are specified by the `license server`. That is, the license checker tracks the date of expiry and the number of views. As soon as the limit is met, the restriction is applied, for instance, by removing the protected content. The data provider decrypts the fragmented content in memory and continuously feeds them to the `stream viewer`. In this way, the complete file is never exposed as a whole on the client-side.

The entire system is developed by the *Sample DRM* company, except for the stream viewer. As stated in the figure, the stream viewer is an external library, for which Sample DRM only has access to the compiled binaries.

Threats. In this system, the protected content is transmitted to the client machines on which end-users have full control over program execution. This immediately gives rise to MATE attacks. In Figure 1.2 we present a set of potential threats from MATE attackers in form of an attack tree [117].

Starting from the root node, a perpetrator's goal is to use the protected content without the restriction which was specified in the usage policy. For this purpose, she has three options: extract fragments of the data from the stream viewer, exfiltrate the content by decrypting it, or circumvent the license checker or modify the policy.

[2]For the sake of simplicity, we assume servers are secured against MATE attacks. In reality, servers are susceptible to MATE attacks just like clients' machines; MATE attacks on servers fall under the category of insiders [6].

Extracting protected content from the viewer requires the attacker to dump memory fragments and later reassemble them. She can (among other possible attacks) tamper with the viewer to perform this malicious activity automatically. Consequently, an attacker can manipulate the viewer to dump and merge decrypted content behind the scene until the entire file is extracted.

Exfiltrating the content by decrypting requires the attacker to obtain the $client_{private}$ key as well as encrypted contents. Since the *data provider (decryption)* (which is located at the client side) has access to this key, it can be targeted by reverse engineering attacks first to locate the key and subsequently to extract it.

Circumventing the license checker is yet another way to use the protected content illegitimately. The license checker is effectively in charge of verifying the usage policies on access requests. Attackers can manipulate the usage policy, i.e., the usage variables for the number of views and expiry date. If the perpetrator manages to manipulate them, then the DRM protection is defeated. Furthermore, they can tamper with the license checking logic, for instance, to accept any license as valid regardless of the actual validity of the license.

The threats mentioned above are a representative set of MATE threats manifestation of which can lead to the disclosure of DRM's proprietary contents. Whether attackers perpetuate modifications in the client software or not draws a definite line between the attacks. In our threat analysis, extraction and decryption attacks may require no changes in the software. However, fooling the license checker requires the attacker to tamper with the DRM client. In this thesis, we concentrate on attacks that include software tampering.

1.3. Problem Statement

MATE refers to a class of adversaries whose access to the software execution environment grants them mighty power to monitor, disrupt, and tamper with the running programs [139]. Consequences of such actions could range from bypassing license checks [15], cheating in games [79], to compromising critical infrastructures [109, 158]. With the growing dependence of our infrastructure on software systems and the emergence of IoT in our daily life, the need for protecting software integrity is no longer only a matter of IP protection but also a threat to our safety.

Against MATE attackers, there exists no silver-bullet countermeasure. As mentioned earlier, besides technical problems (partitioning, interface vulnerability, side channels, and vendor-specific development expenses) hardware-based protections are not yet available on plenty of computing devices. Theoretically-secure schemes such as indistinguishability obfuscation [86] are still considered to be impractical [18]. This is perplexing as practical *software integrity protection* (SIP) measures can only raise the bar against attackers [5, 9, 19, 22, 47]. That is, given enough time, resources, equipment, and skills, attackers eventually win [72]. However, hardening measures, in some use cases, can potentially cause enough trouble for attackers such that defeating the protection becomes very challenging, which is economically no longer attractive [45].

In the past two decades, numerous software protections were proposed in the literature. Each measure essentially aims at thwarting a particular type of attack(s). In other words, they add protections on certain *integrity assets* whereby mounting certain attacks becomes relatively (as opposed to no protections) harder. These assets include, but are not limited to, the program's code (logic), control flow, sensitive data (e.g., hardcoded rates in a smart meter program), etc. [9]. However, for attackers often successfully mounting a single attack is sufficient to undermine assets, i.e., "security is no stronger than its weakest link".

Not only protections fall short of providing a comprehensive protection but also how good they protect integrity assets (how hard it is to defeat them) is unknown. Despite the existence of a multitude of protection schemes, we are not aware of a comprehensive study that compares the advantages and disadvantages of these schemes. No holistic study was done to measure the effectiveness of such schemes. As a direct consequence, it is impossible to quantify the security of protection techniques. Inadequate (or lack thereof) integrity protection could lead to severe breaches in systems. Given that the effectiveness of integrity protection techniques is unknown, MATE attacks remain a significant threat to the security of systems. Given that software integrity protection is the defense we consider in this thesis against MATE, we set to characterize those schemes.

Research Problem. How good are software integrity protection techniques?

1.4. Purpose of the Study

Despite the existence of a multitude of protection methods, we believe the offered security, advantages and disadvantages, practicality, and limitations of the protections are unclear. Some disadvantages or limitations can potentially render protections impractical. Furthermore, a majority of prior research work focused on applying a single protection scheme. Practitioners often utilize a range of protections concurrently to maximize the offered protection. Simultaneously, academics tend to have an opaque feeling that SW-based SIP is problematic/useless because all known schemes have somehow been broken in the past. We argue that there are similar and complementary (side channels; applicability, e.g., in IoT) problems for HW-based techniques. The purpose of security is not full 100% protection– but rather to raise the bar. And this requires understanding how good/resilient specific schemes, and their combinations, are. The applicability, amenability, and further usage constraints of combined protections come with their advantages and disadvantages.

In this thesis, we strive to enhance the security of software systems against MATE tampering attacks. Notwithstanding the impossibility of unconquerable defenses, we aim to raise the bar to the degree that imposes the highest toll on perpetrators.

To achieve our purpose, we set our objectives to **i)** exploring efficient and secure constructs to protect software integrity; and **ii)** building a unified method to assess the strength of protections. Besides exploring novel composition approaches, we intend to thoroughly study the extent to which protections can benefit from compositions.

1.5. Research questions

The general question we are aiming to answer is:

Research question. *Which security guarantees at which cost could software integrity protection schemes offer against automated MATE attackers?*

We derive the following subquestions from our general research question:

RQ1 Which aspects of software integrity protection techniques are security-relevant?

RQ2 How do integrity protection techniques mitigate attacks?

RQ3 What are the shortcomings of the existing protection schemes?

RQ4 How can we improve protection schemes aiming for enhanced security?

RQ5 How can we compose protections to mitigate a wider range of attacks?

RQ6 How to quantify the resilience of individual and composed protection techniques?

1.6. Gaps

Sections 2.6, 3.2, 4.2, 5.2 and 6.2 give an overview of the related work; we see the gaps in the literature as outlined:

G1 Relevant attacks, the advantages, and disadvantages (trade-offs) of SIP techniques in mitigating different attacks were not analyzed with a set of unified criteria previously (addressing RQ1);

G2 The actual security guarantees (resilience) and performance bounds of some protection schemes are unknown (addressing RQ2);

G3 Some well-known protection schemes come with practical shortcomings and hence inadequate or ineffective to be used in practice (addressing RQ3 and RQ4);

G4 Despite the existence of plenty of hardening measures to address particular attacks, effective approaches to compose integrity protections (i.e., lower overhead and higher security) were not addressed in the literature (addressing RQ5);

G5 How good SIP techniques are, in terms of the actual imposed time and cost on adversaries' end, was not previously studied (addressing RQ6); and

G6 Studying protection schemes' practicality and effectiveness is strikingly hard as there exists very limited[3] public implementation of integrity protections (prerequisite for addressing RQ2-RQ6)

[3]There exist some open source software protection tools. O-LLVM (`https://github.com/obfuscator-llvm/obfuscator`) and ASPIRE (`https://aspire-fp7.eu/`) are two well-known projects. Nonetheless, O-LLVM authors did not open source their tamperproofing technique

1.7. Research Design

We research four directions to fill up the identified gaps in the literature: **i)** survey, analyze, and classify the existing protections in the literature (G1 and G2); **ii)** identify the shortcomings of the current protections and subsequently propose enhancements (G3); **iii)** explore novel approaches to combine protections and advance efficient protection composition techniques (G4); and **iv)** research and develop measures to quantify the security of (composed) protection schemes (G5). We briefly discuss the delta between state of the art and this thesis for each of the directions.

1.7.1. Taxonomy

To classify protections, Collberg et al. used natural science and human history to derive a set of 11 different defense primitives [64]. These primitives focus on defense concepts rather than security guarantees. Similarly, [134, 135, 186] proposed classifications with a rather frail emphasis on software integrity protection. Akhunzada et al. studied MATE attackers from the motives perspective [10]. The main gap here is the lack of a comprehensive classification and analysis of software integrity protection schemes. To bridge the gap, we conduct a literature survey and subsequently construct a taxonomy for software integrity protection techniques.

In our survey, we identified four prominent protection schemes, namely Self-Checksumming (SC), Oblivious Hashing (OH), Call Stack Integrity Verification (CSIV), and Result Checking (RC), with high potential applicability to a wide range of integrity protection problems. The original papers came with limited evaluations. Consequently, we are unable to study the effectiveness of those protections, which is a gap in the literature (G2). Therefore, we implement the mentioned protections. Since the LLVM ecosystem comes with powerful program analysis features and significantly improves the portability, we develop the selected protections as compiler transformation passes in LLVM.

1.7.2. Scheme Enhancements

We conduct empirical evaluations on the protections and thereby identify a set of security as well as applicability (coverage) problems. We research and propose enhancements for SC and OH. Let us briefly describe how these protections operate before stating the limitations/enhancements.

Self-checksumming (SC) is a software protection technique that allows programs to detect changes in their binary representation and memory using so-called *guards* [47]. Upon detections, SC may call a response mechanism, for example, aborting the execution or self-repair. SC operates by injecting a set of guards that hash the desired (sensitive) segments of program code in memory (at runtime) and verify that they match to expected values. Consequently, if perpetrators tamper with the program code using certain techniques, SC guards can detect them. Once inconsistencies are detected, a *response mechanism* is triggered which carries out a punishing action [139], e.g., by injecting a fault into the stack that eventually causes a crash in the application. SC guards can further be hardened by a set of interconnected guards that protect each other [47].

Lifted Self-Checksumming

The SC checks have to be executed at runtime. When checks take place locally (i.e., no remote verifier), the expected hash values need to be pre-computed and, after compilation, inserted into the executable. Not only does this approach require knowledge of the underlying system's architecture, it also mandates a post-patching step to put these expected checksums (hashes) into predefined places. Patching binaries is an extremely tedious and error-prone process [22]. This process also limits the use of other obfuscation techniques, as one needs to maintain a set of placeholders (i.e. contiguous sequences of 4 or 8 bytes in the code segment) for the pre-computed checksums at known offsets in the executable. Applying obfuscation would likely change the offsets and contiguous layout of the placeholders.

To eliminate this post-patching step, we leverage the compiler framework LLVM [119] to implement self-checksumming atop virtualized instructions. For this, we also implement virtualization obfuscation. LLVM already implements backends for different system architectures, which removes the required architecture knowledge and post-patching. Applying the guards at a higher abstraction level removes the post-compilation patching process at the binary level entirely. We bypass the pathing process by first applying virtualization obfuscation [87] and then adding the guards in the custom interpreter bytecode.

Practical Oblivious Hashing

OH captures evidence of the program execution by computing a trace hash over the values stored in memory corresponding to arbitrary read and write accesses in a program. These hash values are then matched with expected ones during execution. The expected values have to be precomputed and stored in the programs upon distribution. Simply put, OH checks the execution effects in memory, as opposed to checking the code itself. Therefore, it is harder to identify (stealthy) or trick (resilient) OH in contrast to SC.

Although OH is a more appealing protection measure in principle, it is unable to protect input-dependent program traces, for instance, memory accesses with values dependent on user-input or environment variables. The limitation is mainly because hashing such traces results in different hashes for different program executions.

An effective strategy to cope with nondeterminism requires addressing two challenges - **i)** how to automatically extract deterministic program segments, and **ii)** how to protect non-deterministic ones. The second problem is more striking as a large portion of program traces are expected to be input-dependent. For instance, only 0.5% of program instructions evaluated in [5] are deterministic. In this thesis, we design and develop novel techniques to cope with the mentioned limitation of OH to build more robust protection.

1.7.3. Composition

Composing protection is a viable option to enhance the resilience of protections. The reason being that a "well" composed protection may force attackers to defeat a multitude of protections, some of which may reside in cross-checking network of checkers. A composition of protections intuitively adds resilience to the protection.

However, it turned out that composing protection is nontrivial due to the potential raise of *conflicts* and trade-offs that users need carefully reconcile. To the best of our knowledge, the existing techniques in the literature [167, 170] make conservative assumptions as to whether protection schemes are composable or not. More importantly, the conflicts and security-performance trade-offs were not thoroughly studied. We design and develop a composition framework that optimizes the composed protection yielding better resilience and smaller overheads.

1.7.4. Quantification

After reviewing the literature on the security of protections, we conclude that security is comprised of two factors: **i)** the difficulty of localizing protections (*localizability*) and **ii)** the difficulty of defeating/disabling/bypassing protections (*defeatability*).

Localizability

SIP routines are generally injected into programs of interest via a range of tools including source code rewriters, compiler passes, linkers, and instrumentations. These tools leave recognizable patterns in the protected binaries that could potentially expose protection routines [9]. To mitigate pattern-based attacks, we inevitably have to utilize diversification obfuscation [25] on protection guards. How effectively such diversified guards can counter pattern-based attacks appears to be a research gap. We aim to measure the effectiveness of obfuscation in countering pattern-based attacks on integrity protection guards. We base our study on guards that are generated by our SIP-toolchain[4] (namely, CSIV [2], SC [47], OH [49] and SROH [5] protection techniques). We expect that the primitive pattern-based attacks (to a great extent) fail to recognize obfuscated guards. We utilize machine learning techniques to localize the protection guards hardened by combinations of obfuscations. Furthermore, we set to measure the effectiveness of taint-based attacks [150] in localizing combinations of protections.

Defeatability

Quantifying protection defeatability (ease of bypassing) is rather challenging. The difficulty lies in the extended attack vector that enables MATE attackers to mount their attacks on protections. The perpetrators' expertise is another important dimension that one has to

[4]https://github.com/tum-i22/sip-toolchain

consider in the quantification of the defeatability. In this thesis, due to the lack of access to a pool of skilled hackers, we set to come up with a metric to rather approximate the difficulty of bypassing protections. We are interested in ranking combinations of protections based on the imposed work to disarm entangled protections. To this end, we propose a metric that approximates an attacker's effort in terms of the number of required modifications (patches) for combinations of protections.

1.8. Contributions

C1 We close the G1 gap by proposing a taxonomy for software integrity protection capturing advantages and disadvantages, constraints, shortcomings, and security guarantees of different protections (refer to [9]).

C2 To address G2, we benchmark protection techniques w.r.t. security, performance using datasets of real-world programs (refer to [5, 7, 8]).

C3 We fill the G3 gap by proposing novel protection extensions and combinations (refer to [5, 7, 22]) to enhance the security and resilience of protections;

C4 Aiming for holistic protection (G4), we propose a generic technique for effective (i.e., conflict-free and optimized for better performance) protection composition (refer to [8]);

C5 To address G5, we propose a quantification methodology to estimate the resilience of protections (refer to [**?**]). As part of our methodology we make the following contributions:

- C5.1. Empirically evaluate the stealth of protections against pattern-based, taint-based, and machine-learning-based attacks; and
- C5.2. Propose a metric to estimate the required modification effort from adversaries in defeating protections;

C6 We open source all the developed SIPs, composition framework, and evaluation benchmarks to support the reproducibility of results and thereby address G6 (available at `https://github.com/tum-i22/sip-toolchain`).

1.9. Assumptions, Limitations, and Scope

Physical and Side Channel Attacks

MATE attackers with physical access to the machines can mount side channels and physical attacks. These attacks, although very related to the problem at hand, require a whole different line of defense, i.e., more associated with the field of hardware security. Therefore, we consider side-channel and physical attacks out of the scope of this thesis.

Human Attackers

MATE attackers usually refer to the human attackers. Given attackers' ingenuity, it is challenging (if not impossible) to evaluate the resilience of protections against actual human actors. The difficulty stems from the scarcity of a reliable number of skilled professionals who are willing to participate in the research. Moreover, a universally accepted basis is needed to estimate the level of expertise of such participants. We do not have access to such professionals nor are aware of an agreement to measure attackers' skills. Therefore, we are unable to conduct human-oriented experiments. Instead, we resort to automated attacks based on the literature.

Hardware-Assisted Protection

Due to the unavailability of the trusted hardware in plenty of machines, partitioning problems, and trusted partitions' interface vulnerability we focus on software-based protections. It is worthwhile to emphasize that software-based protections can potentially mitigate attacks on trusted partitions' interfaces.

1.10. Structure

Chapter 2 presents a taxonomy for integrity protection techniques. In Chapter 3, we introduce our enhancements to the Oblivious Hashing protection. Chapter 4 presents our proposed virtualized self-checksumming protection. In Chapter 5, we present our protection composition framework. Chapter 6 covers our technique for quantifying the resilience of protections. We conclude this thesis and discuss future work in Chapter 7. Appendix A provides brief summaries of selected related work. Besides, in Appendix B, we cover notes on the reproducibility of the results.

2. Taxonomy

After surveying the literature, we analyzed and further classified the reviewed articles. This chapter presents the outcome of our analysis encompassing a taxonomy for integrity protection, analysis of various correlations within the taxonomy, and, finally, identification of gaps in the literature. Parts of this chapter were previously publishd in a work [9] co-authored by the thesis author.

2.1. Overview

Since the seminal work of Aucsmith [15], a great deal of research effort has been devoted to fight MATE attacks, and many protection schemes were designed by both academia and industry. Advances in trusted hardware, such as TPM and Intel SGX, have also enabled researchers to utilize such technologies for additional protection. Despite the introduction of various protection schemes, there is no comprehensive comparison study that points out advantages and disadvantages of different schemes. Constraints of different schemes and their applicability in various industrial settings have not been studied. More importantly, except for some partial classifications, to the best of our knowledge, there is no taxonomy of integrity protection techniques. These limitations have left practitioners in doubt about effectiveness and applicability of such schemes to their infrastructure. In this chapter, we propose a taxonomy that captures protection processes by encompassing system, defense and attack perspectives. Later, we carry out a survey and map reviewed papers on our taxonomy. Finally, we correlate different dimensions of the taxonomy and discuss observations along with research gaps in the field.

2.2. Introduction

MATE attackers have direct access to the host on which the program is being executed. In most cases, there is no limitation whatsoever on what they can control or manipulate on the host. In effect, a user who installs an application on their machine has all controls over the computation unit. In this setting, perpetrators can inspect programs' execution flow and tamper with anything in program binaries or during runtime, which in turn, enable them to extract confidential data, subvert critical operations and fiddle with the input or output data. Cheating in games, defeating license checks, stealing proprietary data, displaying extra ads in browsers are typical examples of MATE attacks.

Attacker goals and motives While there are different goals that MATE attackers may pursue, we particularly focus on attacks violating software integrity; other attacker goals are out of our scope.

Reputation, financial gains, sabotage, and terrorism are non-exhaustive motivations for attackers to target the integrity of software systems. Disabling a license check is a classic example of MATE attacks in which perpetrators can potentially affect the software vendor's revenue, for instance, by publicly releasing a patch for popular software. Adversaries may harm a company's reputation by manipulating program behavior, for instance, to show inappropriate ads. Perpetrators may further target the safety and security of critical systems. Corresponding consequences of attacks on a nuclear power plant system could be dire. Akhundzada et al. [10] elaborate on MATE attacker motives in more detail.

Local attacks Local attackers normally have the full privilege on the software system as well as physical access to the hardware. Such attackers can readily tamper with softwares at any stage (at rest, in-memory and in-execution). Moreover, they could potentially load a malicious kernel driver to bypass security mechanisms employed by software. For instance, [177, 182] present two attacks to defeat protection schemes on a modified Linux kernel. Physical access to the host has the same effect. It enables attackers to tamper with systems' hardware and/or configurations. [85] shows that perpetrators can install a malicious hardware to disclose confidential data.

Remote attacks MATE attackers do not necessarily require physical access to the system of interest. Many harmful attacks can be carried out remotely. This type of attacks is known as *Remote-Man-At-The-End* (RMATE) [59]. In this model, attackers need to either remote access the target system or deploy their attacks by tricking end-users. For example, Banescu et al. [19, 22] discuss two RMATE attacks on Google Chromium that were carried out by deployed malicious payloads into the Google Chromium browser. These payloads (in the form of plugins) alter the browser's original behavior to execute malicious activities, e.g., showing extra adds or collecting user information.

Intruders are another type of attackers that are very similar to RMATE attackers. Intruders penetrate software systems normally through their public interfaces (e.g., websites) to find vulnerabilities. A successful exploit may enable them to tamper with the system's integrity, e.g., by materializing a buffer overflow or SQL injections. While RMATE attacks have much in common with intruders, the main difference lies in the access privilege that RMATE attackers posses before and to manifest their attacks. Simply put, RMATE attacks exclude exploiting vulnerabilities and hence starts by a granted access to the host or a program that it contains. Nevertheless, an intruder (after successfully compromising the security) can carry out RMATE attacks. Therefore, the borderline between the two is blurry.

Further attack types Another realization of MATE attacks is through *repackaged software* [128]. In this attack, perpetrators obtain software bundles (normally popular ones) and modify them with malicious codes to create counterfeit versions. Later, this repackaged software is shipped to software hubs to be installed by victims. Since malicious operations are normally dormant, counterfeit software appears to be the same as the original ones to end-users. That is, they can remain on user devices for a while and eventually harm their assets, for instance, by deleting user files after three hours of program usage. Nevertheless, attackers first need to get users to install repackaged software. However, it does not seem to be an obstacle for attackers: a recent study has shown that 77% of the popular applications available on Google play store, a trusted software repository for Android application, have repackaged versions [129].

Targeted malware is another form of MATE attacks in which sophisticated malware is designed to violate a particular system's integrity. Stuxnet [109] is a malware that manipulated programmable logic controllers' code, causing severe damage to the Iranian nuclear program. ProjectSauron [110] is another example of targeted malware where governmental sensitive information was covertly collected and subsequently transmitted to the attacker's server.

In light of these threats, researching, developing, and deploying protection mechanisms against MATE attackers is of paramount importance. In particular, the integrity of software demands protection.

Approaches to protect software integrity Generally, integrity protection refers to mechanisms that protect the logic and data of particular software. Integrity protection is a part of the *Software Protection* field, which is also known as *tamperproofing*. Collberg defines tamper-proofing as "[a way] to ensure that [a program] executes as intended, even in the presence of an adversary who tries to disrupt, monitor, or change the execution" [58].

At a high level, integrity protection techniques are comprised of two main mechanisms: `monitor` and `response`. The monitor mechanism detects inconsistencies by monitoring the system's invariants at a particular representation. The response mechanism engages upon noticing divergence form the known good state of the system and reacts in a way punishing to the attacker [61]. In the case of repackaged software and RMATE attacks, these reactions will be limited to terminating the process, informing users about the violation of integrity or notifying a home server (phoning home).

Unlike cryptographic protocols, protection schemes against MATE fail to provide hard security guarantees, e.g., by the difficulty of solving a a computationally complex problem such as the discrete logarithm in a secure multiplicative group [148]. Dedic et al. [72] argue that there is no tamper-resistant method that resists against polynomial-time adversaries. It means all the protection schemes, given enough time and resources, are eventually defeated by attackers. However, protection schemes can raise the bar against perpetrators by increasing the cost and effort needed to violate system integrity. The introduced cost and effort are often sufficient to mitigate RMATE, repackaged software, or targeted malware. However, we are not aware of any studies that qualify protection resilience thoroughly.

Tamperproofing is commonly used in combination with *obfuscation* [66] software protection technique. Obfuscation aims to reduce the understandability of adversaries by complicating programs. In contrast to tamperproofing, obfuscation remains unaware of the program modifications. Moreover, tamperproofing can detect the occurrence of tampering attacks and respond to such attacks in a way punishing to actors [61].

Advances in hardware security have positively impacted integrity protection research. *Trusted Platform Module* (TPM) (`https://trustedcomputinggroup.org/`) is an approach in which software protection meets hardware security [137, 140]. TPM is a tamper-resilient micro-controller designed to carry out securely sensitive operations, such as secure boot, encryption, attestation, random number generation, etc.

Recently, Intel has also been working on secure hardware modules. Their Software Guard Extensions (SGX), introduced in 2015, are finding their way in academic research [26].

Gap Despite a multitude of protection schemes that mitigate certain attacks on different system assets, we are not aware of a comprehensive study that compares the advantages and disadvantages of these schemes. No holistic study was done to measure the completeness and effectiveness of such schemes. How these schemes operate and what components they are comprised of not identified nor plotted in context.

Current classifications lack the level of detail required to evaluate and select schemes by practitioners. We are not aware of any classification beyond what was proposed by Collberg et al. [61, Chapter 7]. In effect, classifications were done only at an abstract level, i.e., monitor and response mechanisms. While integrity protection schemes comprise far more components and impact various parts of the system, they introduce unique constraints that may become completely inapplicable in certain application contexts. These limitations have left practitioners in doubt about the effectiveness and applicability of such schemes to their infrastructure.

Contribution We propose a taxonomy encompassing system, defense, and attack views and extract relevant criteria in each view. These three views aim at capturing a holistic view of protection mechanisms. The system view captures the components of interest of the system to protect. The defense view elaborates on the characteristics of the defense mechanism. The attack view draws the attacker model and resilience of the schemes to certain attacks.

We evaluate our taxonomy by mapping over 54 reviewed research papers and discuss their advantages and disadvantages. We further correlate different dimensions of our taxonomy and discuss insightful observations concerning security, resilience, performance, applicability, and usage constraints in practice.

2.3. Proposed Taxonomy

Current classifications of integrity protection techniques lack a level of the detail that is required by practitioners to evaluate and select schemes. To the best of our knowledge, the existing classifications remain abstract, i.e., merely describing monitor and response mechanisms. However, integrity protection schemes comprise far more components and impact various parts of the system. For example, they may introduce unique constraints which make them inapplicable in certain application contexts. Such limitations have left practitioners in doubt about the effectiveness and applicability of integrity protection schemes to their infrastructure.

This section proposes a taxonomy for software integrity protection techniques that provide a holistic classification and thereby facilitates the usage of protection schemes in different contexts. We build the taxonomy based on the protection process that a user follows when striving to protect a program's integrity. This process starts with identifying system assets and continues with the identification of possible attacks and defense mechanisms.

As a consequence, our taxonomy is comprised of three dimensions: **(i)** a *system view*, **(ii)** an *attack view* and **(iii)** a *defense view*. The *system view* describes the system as a whole and the encapsulated assets, granularity, and representations in different program life-cycles. It is the integrity of the assets that ought to be protected. Examples include license checkers and, in our DRM example scenario, data providers. The *attack view* captures actions that perpetrators may carry out to undermine the integrity of the assets above. For example, Figure 1.2 showed that tampering with the logic of license checker could harm system assets. Finally, the *defense view* encompasses mechanisms to prevent or detect attacks on different representations of the assets. For instance, we can utilize logic protection measures to mitigate the risk of tampering attacks on license checker. Unlike attacks, defense mechanisms may have implications on the system life-cycle since they may alter a system in various ways, e.g., performance, integration, and incident handling.

In building the taxonomy, we noticed strong dependencies between the elements within and across the three dimensions. Because such relations are widely captured using class diagrams [37, 103], we make use of UML class diagrams [153] to model and depict such relations. Concretely, our model uses *association*, *inheritance*, *aggregation* and *composition* relations with the same meaning as defined in the UML specification. Association indicates two elements are related. Simply put, one can access an attribute or a method of the other. Aggregation depicts the part-of relationship between two elements. Inheritance expresses specialization of a particular element, while composition represents the set of elements that compose a specific piece. In the following, we first introduce the taxonomy along with a few examples. Then, we discuss the dependencies between different views. Later, Section 2.4 maps related literature to the classified attack and defense mechanisms.

2.3.1. System

The system dimension captures the characteristics of the system to be protected. This dimension is comprised of `asset`, `lifecycle activity`, `representation` and `granularity` main classes along with some more specific subclasses for each class. In the following, we first describe these main classes, their relation and then elaborate on each of them and their subclasses. Figure 2.1 presents the system view of the taxonomy.

`Asset` is the core class of this view as it captures the elements whose integrity need to be protected against attacks. In our taxonomy, assets include `behavior` and `data` the tampering of which may harm system users or software producers. Tampering with the license checking behavior and manipulating usage count variables data (number of views and expiry date) in the sample DRM are examples of behavior and data assets, respectively.

`Lifecycle activities` capture the different stages that a program undergoes in its lifecycle. Depending on the stage of a program, assets have different `representations`. Simply put, assets are exposed and presented in different representations from program source code all the way to the program code pages in the main memory. Each of these stages have access to different representation of the assets, however. That is, particular asset representations are only available in particular lifecycle activities.

An asset can have various `granularities`. Granularities correspond to different abstraction levels, e.g. a license check may correspond to a source code C function along with its set of statements, or, more fine-grained, a basic block within one function.

1. **Integrity assets** include valuable *data* or sensitive *behavior(s)* of the system, tampering with which renders the system's security defeated.
   a) **Data** refers to any sensitive information that is produced or processed by the application. Application input, configuration and database files are examples of such data assets. For instance, in the sample DRM, usage count variables are data assets.
   b) **Behavior** is the effect of program execution. Similar to tampering with data, subverting an application's behavior (logic) can have obnoxious consequences. For instance, in the sample DRM, stream viewing decrypted fragments as well as license checking are behavioral assets.
2. **Representation.** Assets can have multiple representations, viz. *static*, *in-memory* and *in-execution*, depending on the program state from start to finish.
   a) **Static** captures assets when they are at rest, i.e. stored on disk. These assets representations are accessible via the file system.
   b) **In-memory** represents assets in memory, i.e. RAM or virtual memory. Process memory captures this representation which includes *code* and *data* invariants of processes.
   c) **In-execution** captures tangible effects of assets during execution. *Trace* and *hardware counters* are subclasses of the in-execution representation. Meanwhile, trace has a subclass that comes with timing data, i.e. *timed trace*.
3. **Granularity.** Assets have different scales and levels of detail that vary from an instruction to the entire application. For example, a license check is usually represented as a function whereas control flow integrity depends on a large set of branching instructions in the application.

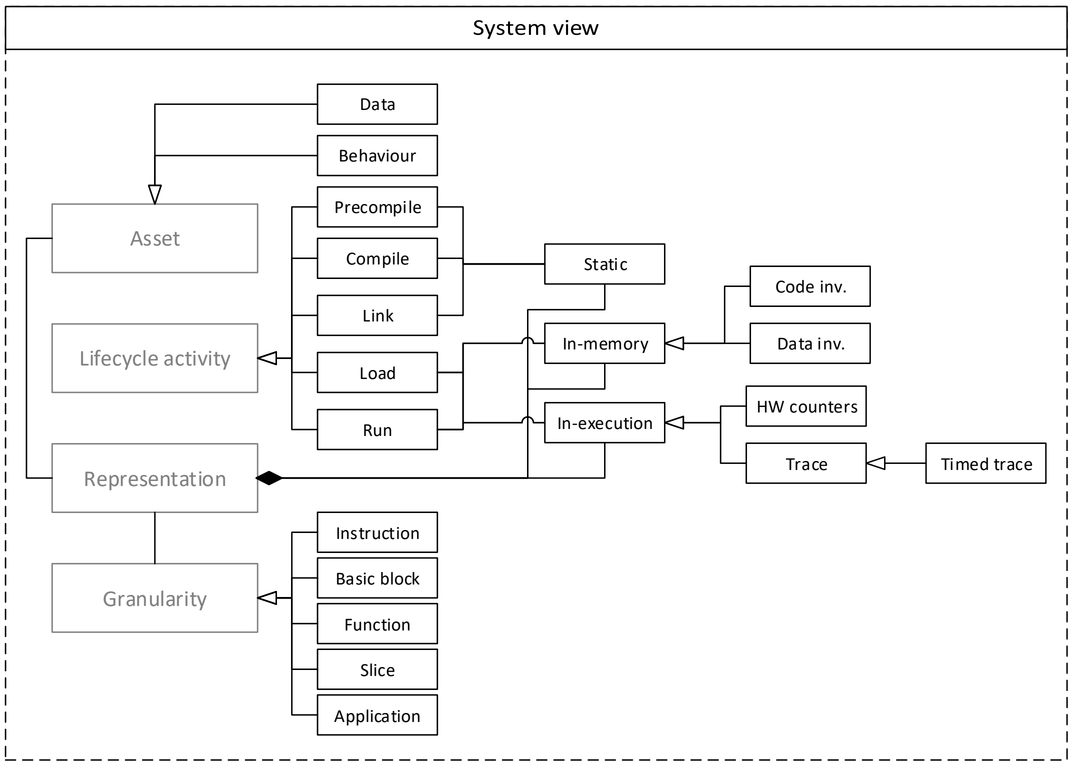


Figure 2.1.: System view dimension of the taxonomy depicting the system main classes as well as subclasses and their relations.

a) **Instruction.** A single instruction is evaluated as security critical.

b) **Basic block (BB).** A set of consecutive instructions with exactly one entry point and one exit point.

c) **Slice.** A set of consequential instructions scattered in a program, e.g. the branching instructions that dictate the control flow of a program.

d) **Function.** A complete function, e.g. *licenceCheck()*, is the goal of protection.

e) **Application.** The entire application or library is supposed to be protected, i.e. the entire program is security critical. For instance, a power plants controller application is in its entirety sensitive.

4. **Lifecycle.** The lifecycle indicates a series of different states that each program undergoes from development all the way to execution, viz. *pre-compile*, *compile*, *post-compile*, *link*, *load* and *run*. Every program has to go through these activities strictly in the mentioned order in order to eventually get executed. However, if a program is not

developed in-house, i.e. source codes are not at hand, the first two activities (pre-compile and compile) are out of reach on the protector side, as these activities require access to source codes.

As depicted in the system view (Figure 2.1) by association links, the first four states (pre-compile, compile, post-compile and link) deal with the static representation of the assets. The last two states (load and run) are concerned with both the in-memory and in-execution representations. In the following we describe these states:

a) **Pre-compile.** This is the state in which a program's artifact, which are represented statically, e.g. in the form of application source code and other bindings such as testing, database and setup scripts, etc, are delivered. Source-level transformations could be triggered at this stage to add protection routines.

b) **Compile.** In this stage, a compiler transforms the program's (static) source code into the targeted machine code. It runs several passes starting from lexical analysis and finishing with binary generation. Additional protection passes can be integrated into the compiler pipeline.

c) **Post-compile.** After compilation, static program artifacts are transformed into executable binaries or libraries. Since the source code is no longer available in this phase, protection schemes operating at this level have to utilize disassembler tools to recover a representation (normally in assembly language) of the application on which they could carry out protection transformations.

d) **Link.** Refers to the state in which a *linker* obtains a set of static compiler-generated artifacts, combines them into a single binary and relocates address layout accordingly. Protections in this phase not only could carry out transformation on the program and all its (static) dependencies, but can also mediate (and potentially secure) the process of binary combination and address space management.

e) **Load.** This is the stage in which the *loader* (i.e. provided by OS) loads a previously combined executable into the memory, resolves dynamic dependencies and finally carries out the required address relocations. Unlike the previous states, this state deals with both the in-memory and in-execution representations of assets. More importantly, it is visited every time a program is subject to execution. Therefore, load time protection transformations can turn protection into a running target to render attacker's previous knowledge about the protection irrelevant. Loader transformations, apart from being executed on every program start, can also verify (and potentially protect) dynamically linked dependencies.

f) **Run.** This stage begins as soon as the loader triggers the program's *entry point* instructions and lasts until the program is terminated. Run also operates at the in-memory and in-execution representations of the assets. Moreover, the run

[0]Except for dynamic dependencies.

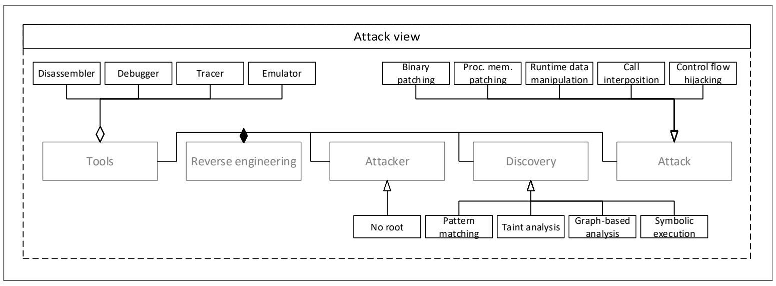


Figure 2.2.: Attack view dimension of the taxonomy showing the attack main classes, subclasses and relations.

state can constantly mutate a program to alter off-board or on-board protections. Although at runtime dynamic dependencies are already resolved, still a protection mechanism can authenticate the loaded dependencies and decide upon unloading or reloading them.

The system view captures critical elements along with their various representation in the system that attacker have interest in undermining their integrity. Both the attack and defense views are applied on a system and hence the system view is the base of the taxonomy. In the following, we first elaborate on the attack view aiming for identifying potential threats and techniques to violate system integrity assets. Later on we discuss the defense view that provides means to mitigate or raise the bar against identified threats (from the attack view) and common attack tools.

2.3.2. Attack

The attack dimension expands over the *attacker* view, encompassing high-level attacks, tools and their relation to other dimensions of the taxonomy, viz. system view and defense view. As depicted in Figure 2.2, the attack view is comprised of the `Attacker`, `Reverse engineering`, `Tools`, `Discovery` and `Attack` main classes. Attacker represent the perpetrator whose goal is to harm system assets by violating system integrity. To do so, attackers use reverse engineering techniques. Reverse engineering is a process in which an attacker utilizes (offensive) tools to discover assets and possibly protection mechanisms. The process normally ends with manifestation of a concrete attack that harms system integrity after defeating protection measures (if any).

In the following, we discuss each main class of the attack view in more details along with their subclasses.

1. **Reverse engineering (RE).** RE is located at the heart of the attack view. It encompasses attack, attacker, discovery and tools. RE aims at harming system assets.
2. **Attack.** Tampering attacks themselves can have multiple forms (inheritance) based on which representation of the assets they are applied to. Attacks are carried out on a representation of the assets, thus there is a dependency between attacks and system representations. In the following, we elaborate on different attacks on different asset representations.
   a) **Binary patching.** A successful exploit at the file system level can tamper with the static representation of the assets.
   b) **Process memory patching.** An exploit at the program level or OS level may enable attackers to directly tamper with the process memory.
   c) **Runtime data manipulation.** Similar to memory patching, program or OS level compromises can enable an attacker to tamper with the program's dynamic data that includes the input to a program, the produced output by a program. The runtime data is allocated in system stack and heap.
   d) **Call interposition.** Once a system is compromised, attackers may, for instance, intercept system calls to inject malicious behavior. Call interpositions are out of the scope of software protection, they rather fall under infrastructure security. For this reason, we did not include them in our paper survey. But we still believe it is a potential threat to software integrity and hence should be listed in the taxonomy.
   e) **Control flow hijacking.** Calls and branches define the execution flow of the program. Attackers target the program control flow in a wide range of attacks (e.g. return-oriented programming [162] and buffer overflows).
3. **Discovery.** In order for attackers to violate the integrity of a protected system they need to identify assets or protection routines in a given representation of a program. We name this phase the *discovery* phase. This implies a relation between the system representation and discovery.

   Banescu et. al [20] formulated attacks as search problems (e.g. identifying protection guards in the application can be formulated as a search problem). To the best of our knowledge, no study has addressed the difficulty of discovering protection techniques. In our survey, however, we found four techniques that are commonly referred in the literature, viz. `pattern matching`, `taint analysis`, `graph based analysis` and finally `symbolic execution`. Therefore, we resort to these four common approaches in discovering protection measures.
   a) **Pattern matching.** Manually analyzing a large and complex program is labor and resource intensive. Therefore, normally attackers try to identify and defeat protection mechanism by employing automated attacks based on pattern matching, using for example *grep*. Pattern matching is not limited to search for strings,

it also can search for properties of an artifact (e.g., entropy). Such attacks are plausible if and when an application commits to the usage of a recognizable pattern in their protection. Note that pattern matching can be applied on all representations.

   b) **Taint analysis.** Tainting analysis is a technique in which the influence of a particular input on program instructions can be examined. This tool can facilitate the detection of protection routines for perpetrators. For instance, taint analysis can help attackers to find the connection between check and response functions by following the influence of check variables on response instructions [150].

   c) **Graph based analysis.** A protected program can be represented as a graph in which basic blocks are the graph nodes and jumps express edges. Dedic et al. in [72] argue that in most cases protection nodes are weakly connected as opposed to the other nodes and hence easier to detect.

   d) **Symbolic execution.** Enables adversaries to see which inputs to the program triggers the execution of which part of the program. Since the program is actually being executed to discover execution paths, nothing can remain unseen for symbolic execution engines as long as the constrain solver is successful. This unique feature of symbolic execution can enable attackers to visit all hidden (obfuscated/encrypted/dynamically loaded or generated) instructions of the protection scheme.

4. **Tools.** An attacker can utilize a set of tools to support reverse engineering activity, e.g. to carry out attacks or to identify assets or defense measures in place. To the best of our knowledge, the resilience of different schemes against reverse engineering tools has not been studied. Therefore, in our taxonomy we made a set of assumptions to decide whether to mark a scheme resilient against a particular tool or not. We will discuss these assumptions with substantial details in Section 2.5.7. In the following we discuss some generic tools that reverse engineers normally use.

   a) **Disassembler.** An attacker may utilizes dissembler to disassemble a binary potentially to analyze the protection logic.

   b) **Debugger.** Another tool that enables the attacker to monitor the execution of a program in a slow paced sequential manner. In this attack, debugger can access or alter any runtime data.

   c) **Tracer.** Analyzing program's execution traces could potentially reveal protection mechanism. This becomes more useful when a program employs obfuscation/encryption to hide its logic. However, the traces can reveal the executed instruction (after decryption and de-obfuscation). In this event, attackers may employ more intelligent analysis to defeat a stealthy protection.

   d) **Emulator.** Attackers can employ emulators to study program execution in a lab manner. All system calls and executed instruction can be closely monitored and thus deepen the knowledge of the program internals. With the help of snapshots, steps could be reversed to recover from faulty states, which in turn facilitates the attack.

5. **Attacker.** The actor who carries out disrupting actions to violate the integrity of the system is the attacker. When classifying attackers, distinguishing RMATE and MATE attackers, although appealing, is out of reach. The reason being that the two have very much in common, the main difference is the physical access that MATE attackers possess. Therefore, we classify our attackers to two groups: *attackers with root privileges* and *attackers without root privileges*. This is indicated with `no root` title (corresponding to attackers without root accesses) in our taxonomy in Table 2.2.

The attack view captures potential threats along with the tools and techniques that attackers can use to violate system integrity. The same attacks could be used by adversaries to circumvent or even defeat protection mechanisms as well. Therefore, in order for defense mechanisms to be effective, it is crucial that they resist against attacks and tools.

2.3.3. Defense

This dimension can be seen as the bridge between the system and attack dimensions. It serves as the core of the taxonomy by capturing the activities in which system assets are protected using a set of measures against potential threats. As can be seen in Figure 2.3, the defense dimension is comprised of four main classes: `measure`, `protection level`, `trust anchor` and `overhead`.

The measure refers to a method that is employed to mitigate integrity attacks on programs. Protection level indicates the level of abstraction at which protection is employed.

Trust anchor specifies whether the measure relies on any root of trust, e.g. trusted hardware, or not. Finally, the overhead reports on the performance impact of a measure on the system.

1. **Measure.** A protection measure is the foundation of the protection activity. It can be done either completely locally (i.e. a program verifies its own integrity) or remotely (i.e. a trusted party remotely attests integrity). The inheritance relation between the measure, *local* and *remote verification* stands for this matter. Each measure itself is composed of five distinctive actions: *transform*, *monitor*, *check*, *response* and *harden*. These classes, although pursuing different objectives, contribute in protection a program. In the following, we describe the role of each action in the protection process.

   a) **Transform.** This action applies transformations on representations of assets. As mentioned earlier, these transformations can be carried out at different program lifecycle depending on the targeted representation and the artifact at hand. This indicates a link between the transform and lifecycle activity.

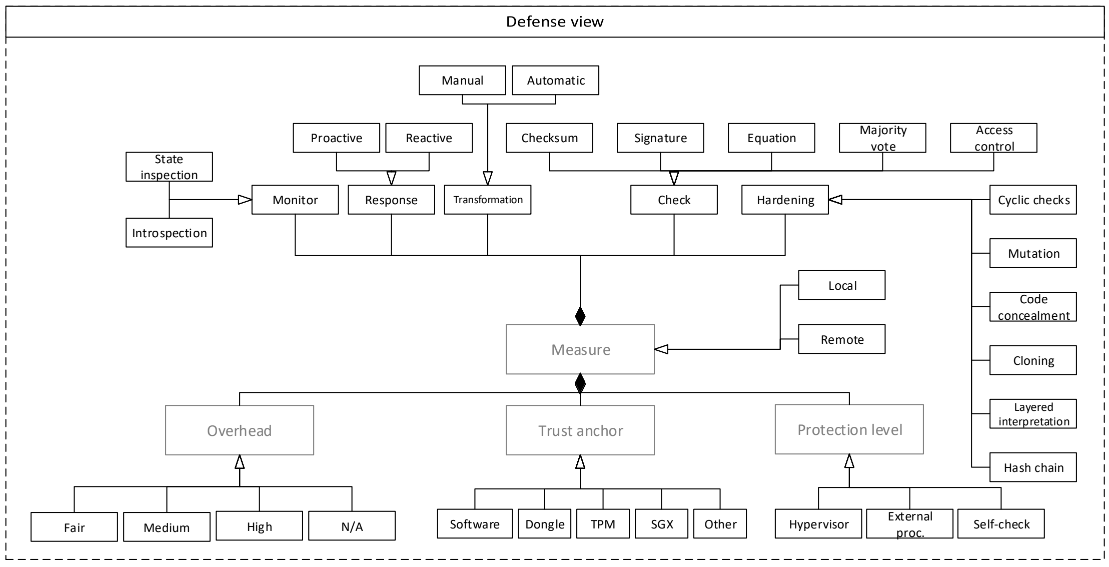


Figure 2.3.: Defense view capturing the main classes, subclasses and relations of the defense dimension.

b) **Monitor.** This component can actively inspect different representations of the system assets. Obviously, this component has to have a access to the system, otherwise it cannot audit anything. Hence, there is a tight binding between monitoring and representation. For monitoring system representations there are two approaches that are widely used in the literature, viz. introspection and state inspection.

   i. **Introspection** monitors a set of static invariant properties of a program, e.g. code blocks, to detect tampering attacks.

   ii. **State inspection** monitors the result/effect of the execution of a program, e.g. function return values, to reason about its integrity.

c) **Check.** This component provides a mean to reason about the collected data by the monitor element, and decide whether assets are compromised or not. The output of this component, based on the employed technique, could be a binary (Yes/No) or accompanied with a confidence number. In the following we discuss the checking mechanisms that are commonly used in protection schemes.

   i. **Checksum.** We classify any mathematical operation that converts large block of data into compact values, which is ideal for comparisons, as checksum based techniques. CRC32 and hash functions fall under this category of checking mechanism.

ii. **Signature.** This refers to a cryptographic protocol for integrity verification by means of verifying the digital signature of artifacts.

iii. **Equation.** The result of a mathematical equation evaluation (with program runtime features) defines whether program integrity was violated or not.

iv. **Majority vote.** A set of functionally equivalent components disjointly carry out a computation and then collectively decide upon the integrity of the outcome.

v. **Access control.** A policy enforcement point beyond the control of the program and (possibly) attacker mediates the access to the critical resources.

d) **Response.** Based on the check component's decision, the response component reacts in a punishing way to attackers. This reaction could vary from process termination, performance degradation, or even attempting to recover compromised assets. Technically speaking, there are two classes of response mechanism: *proactive* and *reactive*.

i. **Proactive.** Schemes utilizing this class of response act intrusively. That is, upon the report of integrity violating, immediate actions are taken to prevent attacks.

ii. **Reactive.** In some cases less intrusive and stealthier responses are desirable. In this model of schemes, the detection of integrity violation does not result in obvious reactions such as program terminations. A reactive response may, for instance, silently report on the violation without making the attacker or user realize.

e) **Harden.** Since the measure itself can be subject to attacks, a consolidation technique is employed by the measure. For example a simple routine (say `monitor()`) that is solely responsible for auditing program state can easily get manipulated by an attacker. The same goes for response mechanisms. An open termination of the program as a response to tampers only adds a weak security.

Thus, it is crucial to add strength to protection measure using hardening techniques. Hardening aims to impede discovery process and thus raises the bar against attacks. This directly relates to the discovery and attack activities in the attacker view.

The hardening can be seen as adding a hard problem for the attacker to solve. This does not necessarily have to be a $np - complete$ problem, in some cases even a quadratic one will cause enough trouble for attackers and exhausts their resources, specially when the attacker is forced to manually solve the problem. For instance when an automated pattern matching fails to defeat the mechanism, attackers are doomed to manually analyze a large portion of the code, which, in turn, presumably deteriorates their success rates. In our literature review, we have identified six different hardening techniques that are commonly used by integrity protection measures. We discuss these techniques as follows.

i. **Cyclic checks.** In this model, protection is strengthened by using a network of overlapping protection nodes. Therefore, a particular asset (representation) may get inspected by more than one checker. In some variation of the cyclic checks, the checkers themselves are also protected by the very same mean.

ii. **Mutation.** Defeating a protection technique normally is the result of a process in which an attacker first has to acquire necessary knowledge about the scheme and its hardening problem, and then, utilizing the knowledge to break the protection. To this end, mutation techniques try to render attacker's prior knowledge irrelevant by frequently evolving the protection scheme.

iii. **Code concealment.** In scheme analysis, attackers often start with the program's binary code and base their studies on it. Code concealment tries to impede this process by concealing a particular representation of the code. Nonetheless, program instructions have to get translated into legitimate machine code right before the execution. The later this translation occurs, the more an attacker is troubled. Some approaches tend to flush translated instructions after execution, such that, at no point in time, the attacker gets a chance to glance over to the entire application code at once. Program obfuscation and encryption are examples of this technique.

iv. **Cloning.** Multiple copies of sensitive parts are shipped in the application and at runtime a random predicate defines which clone shall be executed. This enables the protection scheme to remain partially functional even after successful attacks. In Section 2.5.8 we will discuss this measure in the context of concrete attacks.

v. **Layered Interpretation.** To utilize layered protection a program have to be run within an emulator or a virtual machine. Some schemes entirely virtualized the target application, while others virtualize some parts of the application. This technique enables employment of protection measures at a higher level of abstraction. Since programs has to be run by the host (hypervisor/emulator), they can be verified before execution and executed only if they pass the verifications.

vi. **Hash Chain.** Evolving keys and hash chain assures past events cannot be forged or forgotten. This technique is widely utilized to maintain an unforgeable evidence of the system state.

2. **Trust anchor.** In an abstract sense, this criterion specifies whether the scheme is entirely implemented in *software* or it is assisted by a trusted hardware module.

Hardware based approaches are generally assumed to be harder and more costly to circumvent, due to the required equipments and knowledge at the attacker end. On the other hand, hardware based schemes add more cost, to acquire the required modules, for each client. In the following we discuss different trust anchors.

a) **Software.** Indicates that the scheme security is purely based on software hardening mechanisms without any trusted hardware whatsoever.

b) **Dongle.** Refers to hardware keys that could be used to store small chunks of data (e.g. keys) or program with more resilience against tampering attacks. None of the reviewed schemes in our survey utilizes dongles. Thus, we exclude it from our further classifications. However, according to [147] dongles are popular in practice.

c) **TPM.** Trusted platform module is a quite mature hardware chip that is used by plenty of hardware assisted integrity protection schemes.

d) **SGX.** Intel Software guard extension (SGX) offers dynamic process isolation to mitigate runtime integrity attacks.

e) **Other.** An indicator for reliance on hardware modules other than the explicitly stated ones.

3. **Protection level.** Indicates the enforcement of a protection scheme. It can happen (stated with inheritance relation) in three different abstraction levels:

    a) **Self check.** This level is also known as internal protection in which a protection scheme is integrated into the program to be protected. Simply put, this process is in charge of carrying out its own integrity verifications and further decisions about how to react to compromises. Applying this technique to a distributed architecture adds more resilience. Mainly because the protection schemes react disjointly, a single point of failure is prevented. On the negative side, however, all the decisions are solely based on the state of the program-to-protect, thereby these decisions may lack contextual information.

    b) **External process.** A dedicated process, Integrity Protection Process (IPP), monitors protected programs and responds accordingly. Unlike self-check approaches, IPP can combine multiple sources of contextual information to improve reasoning and thus responses. Also, it enables a mean for high level policy enforcement. These features, in turn, enhance both security and usability of a protection measure. However, there are two drawbacks in this model: **a)** it is challenging to capture the actual state of the programs via an external process, e.g. a malicious program can forge good states for IPP, and **b)** granted the high permission level of IPP, it can become a critical component to attack; single points of failure.

    c) **Hypervisor.** Protection is a part of a hypervisor logic. This entails that all processes are being virtualized and executed on top of the security hypervisor. The current state of the industry highly advocates this model. The hypervisor, in contrast to external process, does not have the problem of state forgery, as it (in theory) can capture all stealthy actions. Nevertheless, the problem of single point of failure remains as a concern.

4. **Overhead.** Performance overhead is an important aspect of integrity protection schemes. Practitioners indeed need an estimation of the overhead on their services. However, a great deal of the protection schemes have not been throughly evaluated, so performance bounds are unknown. We classified performance bounds into four classes as follows.
    a) Fair($0 < overhead < 100\%$).
    b) Medium($101\% < overhead < 200\%$).
    c) High($overhead > 201\%$).
    d) N/A. This class represents schemes with lack of experiment results or unjustifiable numbers due to limited experiments.

The defense view elaborates on techniques to protect integrity against attacks along with implications of such protections on system. Hence defense acts as a bridge between system and attack views. Moreover, defense view can be seen as a classification for integrity protection techniques, which can facilitate scheme selection by users.

2.3.4. View dependencies

Previously, we discussed all the three views of the taxonomy, their classes and dependencies in each view. In this section we provide an overview of the dependencies between the classes from different views. The high-level overview of these dependencies is depicted in Figure 2.4.

In the system view (in the middle), an association between the representation and asset (*"Contains"*) expresses the fact that system assets are exposed in different representations.

In the attacker view (on the right), the support of the tools in the whole process of reverse engineering is expressed by the link between the tools and reverse engineering (the *"Supports"* link). Attackers are the actor to execute an attack, this is also highlighted by an association between attacker and attack. The asset identification, which is the first objective of attackers, is depicted in form of the association between the discovery and representation classes (the *"Identifies assets"* association). The second objective of attackers, after identifying assets, is to tamper with the representation of interest. This is captured by the link between the attack and representation (captioned with *"Tampers with"*).

The defense view (on the left) abstractly captures the defense process. A protection measure may rely on a trust anchor for additional strength, which is shown with the link captioned with *"Strengthens"*. In order to enable protection, a measure needs to transform a representation of the system in one (or more) of the life cycle activities. This is illustrated by the association between the measure and life cycle activity *("Transforms")*. Once a protection

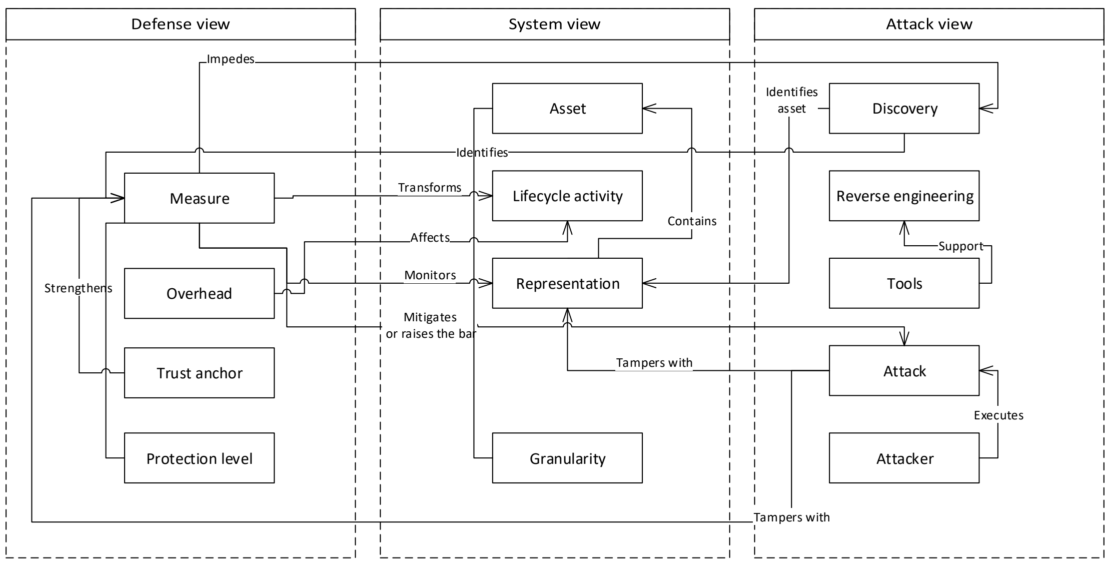


Figure 2.4.: The high-level overview of the taxonomy showing the main classes of the defense, system and attack views along with their relations.

measure is in place, one (or multiple) representation of a system is actively or passively monitored, the *"Monitors"* association between the measure and representation shows this relation. Protection measures could potentially affect system performance and introduce overheads, which is shown with the *"affects"* link between the overhead and life cycle activity. Protection measures based on their hardening measure and response mechanism impede asset discovery and mitigate or raise the bar against attacks. The *"Impedes"* link from the measure to discovery (in the attack view) and *"Mitigates or raises the bar"* between the measure and attack depict these two interrelations. However, protection measure is not the end of the game for attackers. They can target the protection mechanism itself, discover its logic and disable its protection. The *"Identifies"* link between the discovery and measure as well as *"Tampers with"* between the attack and measure captures this matter.

In this context, the resilience of protections can be approximated by two means: **i)** the degree to which defenses impede discovery (localization) attacks (mainly through obfuscations), and **ii)** the difficulty of overcoming protections on adversaries. Later in this thesis, we propose two metrics to approximate the resilience as a conjecture of *localizability* and *defeatability* in Chapter 6.

At this point we have covered all the three views of the taxonomy and their interrelations. For a complete view, Table 2.1 depicts the entire taxonomy. To support readability, for each element of the taxonomy we give an example in the context of sample DRM system. All the examples relate to the DRM system and thus we avoid repeating the context in each example.

2.4. Applying the proposed taxonomy

We conducted a survey on integrity protection and tamper resistant techniques for software systems. This by no means is exhaustive, but we believe it serves as a representative of different techniques in the literature. Table 2.2 demonstrates the mapped literature on the proposed taxonomy.

Since there is no single research work that analyzes the resilience of protection schemes against MATE tools and discovery methods, we were not able to directly map the reviewed works to these criteria (in the attack dimension). To address this limitation, in the Sections 2.5.7 and 2.5.8 we reason about schemes' resilience based on their hardening methods. This enables us to first understand the role of hardening measures in the resilience of schemes and subsequently to complete the mapping. The space limitation hinders detailed discussion of the reviewed schemes. Instead, in the next section we correlate various aspects of the reviewed literature and discuss our findings.

2.5. Observation and Analysis

As part of our analysis we correlate interesting elements in the taxonomy from different views and discuss our findings. These correlations are those that we particularly find interesting for practitioners.

However, there are far more possible correlations that one can carry out. To address this concern, we published a web-based tool that enables end users to correlate any arbitrary pair of elements from different dimensions. The tool is accessible at `http://www22.in.tum.de/tools/integrity-taxonomy/`. In the following we report on the dependencies of the taxonomy dimensions.

2.5.1. Asset protection in different representations

We defined integrity as a property of software that applies to both data and behavior. From the user perspective, it is a daunting task to find protection schemes that protect data and/or logic at a particular representation.

Signature verification is one of the techniques that users commonly utilize to protect integrity. The static signature verification is natively supported by almost all operating systems. These schemes are rather a non-preventive security measure to protect users, but not softwares. Clearly, this does not match the MATE attacker model, where the user is also the attacker.

Besides, these techniques have two shortcomings: **(i)** their security relies on operating system settings, which could be easily manipulated, and **(ii)** they suffer from the *Time*

Table 2.1.: Software integrity protection taxonomy along with examples in the context of the sample DRM.

System view	Assets		Behavior	There are some sensitive logics that need to be integrity protected, e.g. license checker routine and stream viewer
			Data	Usage variables need to be protected against manipulation attacks. Otherwise, attackers can readily circumvent the usage policies
			D&B	The usage variables and any routine that has deals with these variables, in this case content provider, need to be protected
	Representation		Static	Malware or attacker may tamper with the DRM binaries to carry out their attacks
		Mem.	Code invar.	The static shape of the license checker code blocks in the memory is a target to attack for attackers
			Data invar.	Usage count variables in memory are the targets of tampering attacks
		Exe.	Trace	Tampering attacks causes identifiable changes in the program traces. For instance, dumping decrypted fragments in the stream viewer introduces additional calls in the trace
			Timed trace	Tampering with program routine will alter the program response time. For instance disabling license check routine by simply returning true could introduce new timing bounds
			HW counters	Hardware performance counters capture some technical details about the executing routine, such as number of branches, indirect calls and etc. Tampering with the license checker inevitably affects these numbers.
	Granularity		Instructions	All the individual instructions that access $client_{private}$ key need to be protected
			BB	The basic block in which license checking takes place need to be protected
			Function	The licenseChecker() and getExpiryDate() methods need to be protected
			Slice	The chain of instructions that read from or write to the usage count variables need to be protected
			Application	The entire stream viewer component (application) need to be protected
	Lifecycle a.		Pre-compile	License checker and data provider can be protected at the source code level
			Compile	License checker and data provider can be protected at the compiler level
			Post-compile	Stream viewer can only be protected in a post-compile process, because it is an external library
			Load	Every time the license checker is loaded in the memory a (new) protection is applied
			Run	The license checker routine is updated on the fly (during execution)
Attack view	Reverse engineering	Tools	Disassembly	Attackers may disassemble the DRM client to understand the logic
			Debugger	Attackers could inspect the license checking control flow using a debugger
			Tracer	Attackers could deepen their knowledge about the fragmented decryption in the stream viewer by dumping a trace of the program
			Emulator	An emulator could be utilized to facilitate program analysis
		Discovery	Taint analysis	Attacker can use tainting analysis to detect the checks that enforce usage policies
			Sym. exec.	Symbolic execution could be utilized to find instances of usage variables for which the program delivers the protected contents
			Memory split	Attackers could execute memory split attack to defeat self-checksumming based protections
			Patt. Match.	Attackers could use pattern matching to detect license checks, protection routines and usage policy enforcement points
		Attack	Binary	Attackers can tamper with the program executables at rest
			Proc. mem.	Attackers could leverge in-memory patching to carry out their attacks
			Run. data	Program input and output data is under control of the attackers.
			Ctrl flow	Attackers can subvert program control flow to for instance decrypt content without license check
Defense view	Meas.		Local	The DRM client is fully in charge of the integrity protection
			Remote	Security state of the DRM client is to be reported to the main server
	Mon.		State inspection	Monitor and check the results of the program execution
			Introspection	Read code pages of the license checker compare it to a known value
	Resp.		Proactive	Any detected attack raises the alarm and a delayed or an immediate punishing response is triggered
			Reactive	All accesses to the protected content (genuine or forged) is permitted, however, the server is notified about the violation of the usage policies in a postmortem verification. Server can add such users into a blacklist or file a lawsuit against them
	Trans.		Manual	Protectors have to execute manual tasks in the course of protecting a program, for instance, by developing non-trivial clones for the license checker
			Automatic	The protection transformation requires a limited user intervention and hence it is to a great extent automatic
	Check		Checksum	A checksum is computed over the code blocks of the license check
			Signature	Signature of the components are verified at runtime
			Equation eval	Access data forms a set of verifiable equation, e.g. total number of accesses is less than total number of views specified in the usage policy
			Majority vote	License checker is cloned and executed simultaneously by different threads/processes. Afterwards, a majority vote decides whether the license is valid or not
			Access control	All access requests need to pass through a master node which securely enforces usage policies
	Hardening		Cyclic checks	A network of integrity checkers collectively prevent software manipulation
			Mutation	The license checker code blocks are constantly modified at runtime to mitigate passive attacks
			Code conceal.	The License checker code is encrypted and only decrypted at runtime
			Cloning	The license checker is duplicated and every time at runtime a randomly chosen clone is executed
			Layered interp.	The client application utilizes a random instruction set that can only be executed on a custom secure virtual machine
			Hash chain	Usage event logs are securely chained to maintain an unforgeable evidence of the system usage. Such information could be used in postmortem verifications
	Overhead		Fair	Target clients have limited computation resources and thus a low overhead is desired
			Medium	Target clients have fairly good computation resources
			High	Sensitive parts of the program shall be protected strictly, high overheads are tolerated
			N/A	No information about overhead constraints
	T. anchor		TPM	Client systems do have a TPM chip
			SGX	Client systems support Intel SGX
			Other	Other trusted modules are decided to be used, e.g. a custom-built hardware
			Software	The protection is completely software based
	Prot. Lvl.		Internal	The client application is in charge of verifying its own integrity
			External	A dedicated integrity protecting process frequently verifies the integrity of client applications
			Hypervisor	A secure hypervisor is shipped with the client installation bundle. The client application can only be executed using the provided hypervisor

Table 2.2.: Applied taxonomy on the surveyed literature.

			PaperID	[2]	[15]	[19]	[22]	[26]	[31]	[34]	[41]	[42]	[43]	[47]	[49]	[55]	[65]	[72]	[73]	[74]	[82]	[83]	[87]	[88]	[97]	[98]	[99]	[100]	[101]	[102]	[104]	[105]	[108]	[112]	[114]	[115]	[118]	[131]	[132]	[133]	[137]	[140]	[144]	[145]	[146]	[149]	[161]	[164]	[169]	[171]	[180]	[189]	[50]	[51]	[174]	[187]	[190]
Defense view	Prot. lvl.		Hypervisor							×						×				×	×																				×				×							×					×
			External					×	×				×				×		×			×							×	×			×	×	×	×	×				×		×		×			×	×	×			×					
			Internal	×	×	×	×				×	×		×	×			×					×	×	×	×	×			×	×						×	×	×				×			×				×	×		×	×	×	×	
	Trust anchor		Software	×	×	×	×		×		×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×	×	×	×	×		×		×	×	×	×		×			×	×		×	×	×	×	×	×	×		×		×	×
			Other							×																															×																
			SGX					×																																												×		×		
			TPM							×										×											×		×						×		×	×			×												
	Overhead		N/A		×				×		×	×	×		×			×	×			×			×	×						×				×	×	×			×		×							×	×				×	×			
			High							×				×													×	×					×													×				×							
			Medium			×	×														×								×	×								×												×				×	×	×
			Fair	×				×								×	×			×			×	×										×					×		×		×	×	×		×					×					
	Hardening		Hash chain							×										×										×		×		×					×		×	×			×												
			Layered inter.					×		×						×				×	×		×	×							×			×			×					×												×	×		×		×
			Cloning																								×										×																				
			conceal.		×	×	×																															×								×					×			×	×		×
			Mutation														×							×														×																		
			Cyclic checks				×							×									×	×		×																												×	
	Check		Access control			×		×		×						×																									×									×		×		
			Majority vote															×				×				×				×																		×								
			Equation eval																															×					×																		
			Signature		×								×																	×					×	×																		×			
			Checksum		×		×							×					×	×					×	×			×	×	×	×	×				×				×	×	×				×	×				×	×	×			×	×
	Trans.		Automatic	×			×	×	×	×	×						×											×	×	×	×		×	×		×		×						×	×												×	×	
			Manual		×	×						×	×	×	×	×		×	×	×	×	×	×	×	×	×	×					×			×		×		×	×	×	×			×	×	×	×	×	×	×	×	×	×			×
	Resp.		Reactive																											×												×															
			Proactive	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
	Mon.		Introspection		×		×	×					×	×		×		×	×	×	×		×	×	×						×	×			×	×		×				×			×	×						×	×	×	×	×	×
			State inspection	×		×			×	×	×	×			×							×				×	×	×	×	×			×	×					×	×	×		×	×			×	×	×	×						×	
	Meas.		Remote					×									×		×			×						×	×	×			×	×						×		×			×		×		×								
			Local	×	×	×	×		×	×	×	×	×	×	×	×		×		×	×		×	×	×	×	×				×	×			×	×	×	×	×		×		×	×		×		×		×	×	×	×	×	×	×	×
Attack view	Attack		Control flow	×		×			×						×												×			×							×		×				×	×										×			
			Runtime data			×		×			×	×			×							×				×							×	×															×								
			Process memory		×		×	×		×				×						×			×	×	×			×	×		×	×				×		×			×				×	×		×			×	×	×		×	×	×
			Binary										×				×	×	×																×							×								×							
	No root			×		×	×			×	×	×	×			×			×			×												×	×	×			×				×	×	×				×	×							
System view	Lifecycle activity		Run														×																					×																			
			Load																					×																																	
			Post-compile	×	×		×	×		×			×	×		×			×		×	×	×	×	×		×	×	×			×			×	×			×	×	×		×	×	×	×	×				×	×					
			Compile												×																	×																			×							
			Pre-compile			×					×	×								×						×					×			×	×			×											×					×	×	×	×	×
	Granularity		Application					×					×			×			×	×								×	×	×		×			×	×					×	×		×					×			×					
			Slice			×					×				×									×		×					×			×				×												×			×			×	
			Function							×							×				×	×				×							×				×			×					×	×	×	×					×		×		×
			BB		×		×							×				×					×	×			×																×								×						
			Instructions	×		×			×			×																																									×			
	Representation	In exec.	HW counters																																			×										×								×	
			Timed trace																									×	×											×								×	×									
			Trace	×		×			×						×												×			×							×						×	×										×			
		In mem.	Data shape								×	×										×				×							×	×															×								
			Code shape		×		×	×		×				×						×			×	×	×						×	×				×		×			×					×		×			×	×	×		×	×	×
		Static											×				×	×	×																×							×			×					×							
	Assets		Data and behav.					×		×										×																																					
			Data			×		×		×	×	×								×		×											×	×								×							×								
			Behavior	×	×		×	×	×	×			×	×	×	×	×	×	×	×	×		×	×	×	×	×	×	×	×	×	×			×	×	×	×	×	×	×				×	×	×	×		×	×	×	×	×	×	×	×

of Check–Time of Use (ToCToU) limitation in which signatures are verified on the static representation of programs upon execution, while many attacks (including in-memory patching) could potentially occur after execution is started. [106] reported that over 60% of attacks published by CERT were either buffer overflows or of ToCToU attacks.

Due to the aforementioned limitations, the static signature verification is not an optimal technique as it fails to protect other asset representations during execution.

A natural question that may come to ones mind is why protection for a certain representation instead of protecting assets in all the representations. The answer is threefold. First, attackers might not have the right privileges on some representations, e.g. accessing in-execution representation of processes requires *root* privilege, except when attackers find a way to inject their attacks into the program of interest. Reasonably, to avoid unnecessary overhead we aim to protect the representations that are at risk. Second, some risks might be acceptable for a certain use case, e.g. in IoT programs naively protecting static representation of a program might be sufficient, given that runtime protection imposes unacceptable overhead. Finally, 100% protection of assets in all representations is technically infeasible. Therefore, to shed light on this concern, we correlate software integrity assets to different representations in order to identify corresponding protection schemes.

The desired correlation is depicted in Table 2.3. In the following we discuss the relevant schemes according to their asset-to-protect.

Table 2.3.: The correlation between the integrity asset protection and representation.

		Integrity assets		
		Behavior	Data	Data and behavior
Representation	Static	[73] [72] [146] [171] [65] [114] [43]	[140]	-
	Code inv.	[50] [174] [187] [190] [74] [97] [131] [34] [137] [22] [105] [87] [104] [47] [149] [15] [164] [88] [115] [180] [189] [26]	[74] [34] [26]	[74] [34] [26]
	Data inv.	[98]	[108] [41] [169] [112] [83] [42]	-
	HWC	[132] [187] [164]	-	-
	Timed trc.	[133] [100] [161] [164] [101]	-	-
	Trace	[51] [99] [2] [49] [102] [118] [31]	[19]	-

Behavior protection schemes

As the correlation suggests, a majority of behavior protection schemes with a total number of 22 [15, 22, 26, 34, 47, 50, 74, 87, 88, 97, 104, 105, 115, 131, 137, 149, 164, 174, 180, 187, 189, 190] target the *code invariants* representation in their protection mechanism.

Meanwhile, [98] referred to the shape (invariants) of computed data to reason about the behavior integrity. [132, 164] took hardware performance counters (e.g. the number of indirect calls and elapsed times) into account to evaluate programs' integrity.[187] used Physically Unclonable Functions in combination with self-checksumming to bind software to specific hardware.

Some efforts were found in the literature to raise the bar against program manipulation when programs are at rest (binary). [43, 73, 171] are examples of hardening binaries by statically analyzing them and subsequently adding resilience against tampering attacks. [72] rather presented a theoretical concept to design more resilient integrity protection schemes. [65, 114, 146] proposed techniques to protect program binaries after distribution.

In order to manifest some tampering attacks, attackers inevitably need to change, inject or reorder instructions. In this situation, the order in which program instructions are being executed along with the data that they read from or write to, i.e. a program trace, is a source of knowledge to detect such attacks. This information could be extracted form a program's execution trace. [49, 99] incorporated memory accesses and branching conditions in the program trace into a hash variable to later match it against the expected trace hash. [102] passively verified the execution traces that are reflected into log records. [118] duplicated sensitive regions in the program to add more resilience to attacks by forcing perpetrators to tamper with all clones of sensitive codes. [51] embedded the control flow branching conditions into an artificial neural network model to conceal and at the same time protect the control flow integrity. The correlation also indicates that timing analysis (timed trace) is utilized in [2, 31, 100, 101, 132, 133, 161, 164] to verify integrity of the program behavior. The assumption in the mentioned schemes is that a certain untampered routine expresses identifiable execution time characteristics.

Data protection schemes

Tampering with a program's data can enable attackers to manifest integrity violating attacks. The simplest way to do so is by attaching a debugger and subverting the program flow, for instance, by flipping a register's value. As the correlation suggests, several schemes protect the program's data integrity. [108] introduced a hash-chain alike concept for data integrity with the help of a semi trusted entity. [169] equipped system data flow analyzer nodes and routed the traffic to them for integrity analysis. [83] proposed to duplicate sensitive processes and compared their computed values based on a majority vote scheme. [42] used data flow graph to verify whether the application conforms to the genuine data flow model at runtime. [112] collected data invariants in a program and verified them at runtime. [49] computed a cumulative hash of memory accesses at the instruction level to form a hash trace of the program execution.

Data and behavior protection schemes

Protecting data and behavior together might be a more appealing option when the program to protect possesses both sensitive data and logic. In our survey, we found three schemes that aim for data and logic protection. [74] proposed a technique in which a program's logic as well as data are isolated from external processes by a secure hypervisor. They also designed a process to encrypt data before persistence, so that the data is never exposed to adversaries in plain text. [34] in addition to logic and data isolation, introduced a secure inter-process data propagation in a (nearly) real time manner. [26] designed a fully isolated execution environment for the process as a whole on Intel SGX hardware commodity. All the three schemes rely on trusted hardware.

The presented correlation singles out relevant schemes for protecting different integrity assets (behavior, data and data and behavior) in different representations. This facilitates the scheme selection based on the assets and representations at the user end.

2.5.2. MATE attack mitigation on different asset representations

A program possesses different representations depending on its state, viz. static, in-memory and in-execution. Each of theses representations are subject to tampering attacks. Protections on representations closer to the time of use (i.e. execution) could prevent or detect a wider range of tampering attacks. That is, the static representation of a program is implicitly protected, if the very same program utilizes in-memory protections.

In this chapter, we limit our adversary model, regardless of how attacks are executed, to four generic attack goals: binary patching, process memory patching, runtime data modification and control flow hijacking. Constrained by their permissions, attackers can choose different program representations as their attack targets.

In this correlation we aim to identify protection techniques that mitigate the aforementioned generic attacks on different representations. For this purpose, we relate representations in the system view to attacks in the attack view. This is depicted in Table 2.4. We discuss our findings classified by attacks, viz. binary, control flow and process memory as follows.

Table 2.4.: Defense mechanisms against tampering attacks for different (asset) representations.

		Representation					
		Code inv.	Data inv.	HWC	Static	Timed trace	Trace
Tampering attacks	Binary	-	-	-	[140] [73] [72] [171] [65] [114] [43]	-	-
	Control flow	-	-	[132]	-	-	[51] [19] [2] [99] [144] [49] [102] [118] [145] [31]
	Process memory	[50] [174] [187] [190] [74] [97] [131] [34] [137] [22] [105] [87] [104] [47] [149] [15] [164] [88] [115] [180] [189] [26]	-	[164] [187]	[146]	[100] [164] [101]	-
	Runtime data	-	[108] [41] [169] [98] [112] [83] [42]	-	-	-	[19] [49]

Binary

Static manipulation (aka static patching) targets a program while it is at rest (prior to execution). To prevent these sort of attacks the surveyed literature suggests static representation verification as well as a set of hardening measures to raise the bar against tampering attacks in the host. Note that we do not include signature verifications supported by operating systems, as they do not target MATE attackers. In the following we review the schemes that protect the static representation of programs.

Static representation verification [114] and [73] proposed remote file integrity protection using file system hashing. The main difference lies in the fact that the latter initiates the hash with a remote challenge. However, this technique can easily be defeated if the attacker keeps a copy of the genuine files just for the sake of hash computations.

Hardening measures Neisse et al. [140] suggested to plug (hardware-assisted) configuration trackers in the system to monitor modifications on the file system. All modifications are then signed by a TPM within the host and then reported to a third party verifier to judge about the maliciousness of the actions.

[43] designed a protection scheme to defend against dependency injecting and tampers with the update scripts in a system by utilizing dynamic signature verification and system call interposition in the kernel. They argued these two can mitigate the persistence of a compromise in the system. Meanwhile, [72] proposed to strongly connect integrity protection code with the normal program control flow and discussed theoretical complexity using a graph based game.

To add resilience against dynamic attacks such as buffer overflows in the static representation, [171] proposed an efficient means to detect potential integrity pitfalls in a distributed system by first merging all network wired nodes into a unified control flow graph, so called *Distributed Control Flow Graph DCFG* and then running a static analyzer to find all user-input reachable buffer overflows. In their way, intuitively, pitfalls can be identified with less false positives as opposed to analyzing each node individually.

Collberg et. al in [65] proposed to protect client programs' integrity by forcing constant and frequent updates. In their approach, a trusted server constantly uses software diversification techniques to change a server's method signatures, which in turn coerces the client to update all the program artifacts right after each mutation.

Control flow

As the correlation suggests, there are two representations that were utilized to prevent a program's control flow manipulation, viz. HW counters and traces.

Hardware performance counters were used in [132] to reason about the program's control flow integrity based on performance factors such as the number of calls and the elapsed time on each branch.

The trace representation was used by 10 protection schemes. These schemes are listed as follows. Banescu et al. in [19] proposed a technique to ensure only genuine internal calls can access assets. This is done via a runtime integrity monitoring that traces stack's return addresses and compare them to a white list of call traces. Stack inspection works well for synchronous calls, however, due to the incomplete trace information, trigger and forget calls cannot be handled in this way. To cope with asynchronous calls, caller threads are verified using a Message Authentication Code (MAC mechanism).
Abadi et al. in [2] proposed a scheme to protect the integrity of programs' control flow by injecting a set of caller reachability assertions in the beginning of program branches.

Chen et al. in [49] proposed to compute a checksum over a program's execution trace by hashing its assignment and branching instructions. These hashes are later verified at desired locations in the protected program. Similarly, [99] proposed to compute a hash over a program's assignments and executed branches. These hashes later are used to dynamically jump to the right memory locations. Mismatches in hashes crash a program as they potentially cause jumps to illegitimate addresses. Dynamic jumps are created by the mean of interleaving multiple basic blocks in super blocks with multiple entry and exit points. In effect, the hash values determine which entry/exit points the program shall take at runtime.

To capture an unforgeable evidence of the traversed program's control flow, Jin et al. in [102] proposed a protection mechanism based on secure logging (forward integrity). Their scheme is detective. That is, tampering attacks are detected in a postmortem analysis after they took place. [118] mitigated call graph manipulation by utilizing opaque predicates to randomly switch between a set of nontrivial clones of sensitive program logic (input by human experts). [145] designed a mitigation for ROP attacks by monitoring branching behavior and size of fragments executed before each branch. [31] proposed a mechanism in which a program-to-protect subscribes to an external process which dictates the genuine control flow of the application. The external process can be protected by other expensive integrity protections without introducing extensive overhead on the main application. [51] designed a protection scheme that uses a neural network model to calculate the destination block of protected branching conditions.

Process memory

Protecting process memory based on *code invariants* is the most common approach in the literature. In the following, we briefly introduce different schemes based on this technique.

Self-encryption is a defense mechanism utilized by a number of protection schemes. Aucsmith [15] proposed a scheme that utilizes code block encryption and signature matching to mitigate tampering attacks on process memory. The author refers to the integrity checking codes as *Integrity Verification Kernel (IVK)*. These IVKs, for better stealth, are interleaved with the program functionality. Using this technique, the entire application code except for the starting block is encrypted. From the memory layout of each executed block a key is derived which can decrypt the next block. [180] is another encryption based protection in which the key to decrypt the next basic block is derived from a chain of hashes of previous blocks. Thus, tampering with blocks breaks the chain and results in an unrecoverable program. [149] applied the self-encryption with re-encryption after each invocation to protect Android intermediate code. Protected programs using this scheme, however, cannot use reflection nor host service contracts (otherwise unreachable) due to the code encryption. [174, 190] resorted to code encryption aided by trusted hardware to protect the sensitive code in the memory.

Self-hashing is another mechanism that is employed by a number of process memory protection schemes. [164] used reflection to obtain program codes at runtime. Program codes are subsequently verified using a hash function. In addition to hash verification, HW counters are also utilized to increase the resilience. Chang and Atallah in [47] proposed a software protection technique that builds up a network of compact protection blocks (aka Guards) that along with performing integrity checking of the application blocks (basic blocks) can also verify other protection guards. These protection guards, provided that an untampered version of the code is available, can potentially heal the tampered regions of the program. [22] proposed a technique that utilizes a network of cyclic checkers with multiple hash functions to prevent memory tampering attacks. This scheme can cope with the relocation problem in binaries and hence it protects instructions that contain absolute addresses. To the best of our knowledge, this scheme is the only self-checksumming protection measure whose source code is publicly available. [97] proposed a check and fix protection scheme that utilizes a set of linear testers that compute a hash over certain regions in the process memory. Upon mismatch detection correctors attempt to heal the tampered program.

[87] proposed a technique combining cyclic checks and self-encryption. In this scheme, first cyclic guards are injected and then the entire program is encrypted. An emulator is shipped with the program which can decrypt and eventually execute it. The decryption key is shipped into the binary using white-box cryptography, so attackers can not easily extract it from the program. [88] did a similar protection with the difference of using virtualization obfuscation instead of self-encryption. In addition to that, a set of self-checking guards are generated on the fly to protect the dynamically translated program in the cache. [104] applied control flow flattening obfuscation transformation in combination with self-hashing protection guards. In their scheme, random intervals of a program are checked using CRC32 checksums.

[131] used constant mutation at runtime so that the same memory address is used for multiple code and data blocks in one execution.

Secure hypervisor is used by some protection mechanisms to protect the process memory. [74, 190] used secure hypervisors to verify code pages of the protected programs. [115] proposed a measure that utilizes a secure emulator, code singing and encryption. First programs code blocks are signed and encrypted. Later, the secure emulator can decrypt and

[0]https://github.com/google/syzygy/tree/integrity

execute the genuine code blocks. [189] proposed a hypervisor monitoring tool called VMI to introspect the memory of running virtual machines. VMI has to, however, be executed with root permission on the host. [34] aimed for integrity protection in IoT devices by proposing a concept of secure (tamper resistant) tasks enabled by a trusted hardware. Furthermore, a secure interprocess communication protocol is also designed. In order to pass a secure message, a source process loads the message m and destination id, d_{id}, into CPU registers and then triggers a secure IPC interrupt. The interrupt eventually makes m along with the s_{id} accessible for destination process/task. [137] proposed a hardware assisted hypervisor that enables users to define their custom integrity routines. Similarly, [105] proposed to equip hosts with custom attestation and measurement probes to detect violation of integrity.

[26] proposed a technique to use SGX secure enclaves to place the entire application in isolated containers. This prevents attackers form runtime process memory manipulations.

Static protection to isolate process memory was done in [146]. Their goal was to ensure genuine geolocation information by starting the service in a lightweight hypervisor which is secured by a TPM.

Timed traces are another representation which was used to build protection schemes. In effect, [100, 101] based their schemes completely on timing analysis to verify process memory. In this approach, once the attestation is launched, all active memory contents are copied to the flash memory. Then, the verifier sends a random seed that the attestation client uses for proof of computation. At different check-points the verifier sends further seeds and defines the memory addresses that need to be incorporated in the proof computation. Meanwhile, timing analysis could be used as an additional source of information. For instance, [164] used reflection as its core of integrity protection, but also utilized timing analysis for additional hardening.

Runtime data

To protect runtime data, data invariants and trace representations were utilized by the reviewed protection schemes. The correlation indicates that 7 schemes operate based on the data shape representation. [108] relied on an authenticated data structure (Merkle tree) to enable users to verify correctness and freshness of the access control policies and user data. [41] used a combination of static analyzer and memory analyzer to detect data tampering that leads to kernel exploits. [169] used SDN to route data to an extra node to verify network data integrity. [98] proposed to utilize a set of checkers are added to the program to check return values of the sensitive functions. Inputs for these checkers are generated using symbolic execution. [83], similar to [98], proposed to feed the return values of a given service to a set of clones and have a majority vote based scheme to decide upon integrity of the service. [112] proposed to first run a program using the Daikon tool [76] to capture dynamic data invariants. From these invariants later a set of guards are generated and injected into the program of interest. At runtime these invariant should hold, otherwise response function is triggered. [42] proposed a technique which uses data flow graph to build a data model. At runtime the conformance of the application data flow to the model is frequently verified.

Trace representation was used in two protection schemes to authenticate data integrity. [19] utilized white box cryptography to protect the integrity of a configuration file in the Chromium browser. A white box proxy enables Chromium to sign and verify genuine configuration files. While attackers have difficulties extracting the key from white box crypto, they might be able to subvert the execution flow and misuse the proxy to sign malicious configuration files. Therefore, this scheme authenticates any call to the proxy. [49] incorporated a selected set of memory references into hash variables. The constant program data can be protected using this measure. However, incorporating input data will make it impossible to precompute expected hashes, due to nondeterminism in programs.

In this correlation we identified and discussed the different schemes to mitigate particular MATE attacks, viz. binary, control flow, process memory and runtime data manipulations, at different representations. Amongst the reviewed schemes only [171] targeted distributed systems. This work, however, focused on adding resilience to the static representation against buffer overflows. All the other reviewed schemes focused on protecting assets in a centralized application, which might be in contact with a remote server. However, these schemes clearly fall short in protecting a software system that is comprised of a set of distributed nodes, communicating over network. Protecting program representations in a distributed architecture appears to be a major gap.

2.5.3. Defense integration to program lifecycle

Each defense comes with a mean in which monitoring probes are hooked, connections to response algorithm are established and hardening is applied. In a defense mechanism *Transformation* is the component that employs protection in a program by applying necessary modifications. Transformations may target different stages of a program varying from source codes to executable binaries. This will impose some constraints on the applicability of defenses to different contexts. For instance, one may have no access to the source code for a third party component. That is, for such cases, all the defenses based on source code transformations become irrelevant. The stream viewer in the DRM sample is an example of such components.

In addition to the constrains imposed by lifecycle activities, program representations on which protections are applied impose further constrains. For instance, monitoring memory can impose intolerable overheads in IoT devices. Therefore, it is of major importance to identify integration constraints of protection schemes in different application contexts. To capture this concern, we analyze the correlation between transformation target and system representation. Table 2.5 illustrates the outcome of this correlation. We discuss the correlation by iterating over lifecycle activities as follows.

Table 2.5.: Defense mechanisms for different (asset) representations and their correlation to program lifecycles.

		Representation					
		Static	Code inv.	Data inv.	HWC	Trace	Timed trace
Lifecycle	Pre-compile	-	[74] [164] [50] [174] [187] [190]	[108] [41] [98] [112] [42]	[164]	[19] [102] [118] [51]	[164]
	Compile	[171]	[104]	-	-	[49]	-
	Post-compile	[73] [146] [114] [43]	[97] [34] [137] [22] [87] [47] [105] [149] [15] [88] [115] [180] [189] [26]	[83]	[132]	[2] [99] [144] [145]	[133] [100] [161] [101]
	Load	-	[88]	-	-	-	-
	Run	[65]	[131]	-	-	-	-

Pre-compile

In the following we have a closer look at the schemes that operate on the source code level and discuss their correlation to the asset representations.

Code invariants

[164] proposed a scheme for remote verification of integrity protection. The idea is to retrieve a program's code shape using reflection and subsequently computing a hash over it. This hash is then reported to the remote party. These reflection calls are injected into the program source code.

[74] proposed a scheme that requires developers to generate integrity manifests for all programs-to-protect. Integrity manifest contain program measurements and a unique program identity. These manifests need to be shipped to a secure hypervisor which follows the measurements to evaluate the integrity of programs at runtime.

[50, 174] proposed protections aided by Intel SGX. Both schemes place the guards within the security enclave to harden their schemes. [187] combined self-checksumming with physically unclonable functions to add further resilience to their protection. [190] proposed a hypervisor-based protection that acts as an emulator for encrypted code.

Data invariants

According to the correlation outcome there are four schemes that operate on data shape to verify software integrity. We will discuss these schemes, focusing on their integrability to systems. Data invariants verification is the technique that is used in [112]. In order to capture these invariants, the *Daikon* tool is utilized, which requires programs to be instrumented and subsequently executed. Afterwards, in the protection phase, these invariants are verified by a set of protection guards.

[108] proposed a mechanism to protect program inputs using a set of data integrity polices specified by users. These policies are enforced by the mean of an authenticated data

structure [136].

In order to protect program functionality at runtime, [98] proposed a scheme for C# programs in which return values of program functions are tested in a network of cyclic checks. To construct checkers, the PEX symbolic execution tool [172] is used. PEX generates test cases (input-output pairs) for the functions of interest. Checkers are generated based on the test cases. They invoke a protected function with the test's input arguments and subsequently match the output of the function with the expected result of the test case.

Conversely, [41] designed a scheme that inspects kernel's dynamic data to detect tampering attacks. They claim that their scheme can protect up to 99% of such attacks on kernel.

Trace

The following schemes rely on trace data to evaluate program integrity. Given that log records intrinsically capture enough information about the executed trace, [102] based their scheme on recovering a program trace from its logs. In this scheme program code needs to be augmented with a comprehensive logging mechanism. A hash chain mechanism is utilized to preserve the forward security properties. The logs are supposed to be verified by a trusted external entity Since attackers cannot forge logs, the verifier can detect malicious activities during verifications. [118] proposed a scheme which aims at diversifying program's execution trace by randomly executing non-trivial clones of the sensitive functions. The downside is that all the non-trivial clones have to manually be implemented. [19] aimed at defeating control-flow hijackers by a set of stack trace introspection guards. These guards are injected in the prologue of sensitive functions in the source code. [51] proposed to train a neural network model of the program branch evaluations for a selective set of branching conditions. In this way, the attacker can see the trained model, but never sees the plain branching conditions.

Compile

Compiler level protection could potentially generate far more optimized protection routines thanks to built-in optimizations. Despite the benefits, compiler-based protections require access to the source code. Also, compiler transformations will target a specific compiler and hence restrict the choice of compilers at the user end.

Furthermore, since compiler optimization passes are oblivious to protections implemented, they could potentially break them and thus cause false alarms in protected programs. For instance, some compilers utilize enhanced memory caching by block reordering. This relocates program blocks after protection is laid out and hence breaks the hash checks in self-checksumming based protections, resulting in false alarms [22].

In our survey we only found three compiler based protections. Junod et al. [104] implemented a scheme that aims at protecting *code shape* by a combination of control flow flattening and tampering protection in a compiler pass (implemented in the LLVM infrastructure). On the negative side, no evaluation results on the tamper resistance overhead were published in this work. In an effort for securing distributed programs against buffer overflow exploits, [171] proposed a LLVM based analysis tool that combines the CFG of all individual programs to construct a distributed CFG. The distributed CFG is to improve the accuracy of vulnerability detection. Oblivious hashing [49] is another scheme that is implemented in abstract syntax trees. It operates on in-execution representation to authenticate program *trace*. In this transformation, program assignments and branching conditions are reflected in a hash variable that serves as a trace hash. Later, this hash could be verified at any desired point in the program.

Load

In our survey, we only found one research work that transforms program instructions at load time. Ghosh et al. in [88] proposed a tamperproofing technique on top of instruction virtualization obfuscation[1]. In this scheme, a mechanism was designed to protect translated instructions (code invariants) within the emulator in a program by safeguarding them using a network of checkers (similar to [47]) along with frequent cache flushes.

Run

Transforming applications at runtime is another option. These transformations render adversaries' knowledge obsolete by turning the protection into a moving target. In this setting, attackers have a limited time before the next mutation takes place, and hence their success rates deteriorate. Collberg et al. [65] proposed a scheme for client-server applications in which the server API is constantly mutated using diversification obfuscation techniques, forcing clients to update frequently. Assuming that the server is secure, attackers have a limited time to carry out reverse engineering attacks on client applications (*static*). Soon after the server API is mutated, client applications have to be updated to reflect API usage changes, which renders attackers knowledge obsolete.

The downside of this scheme, however, is the obligation of having the server and the clients constantly connected to receive highly frequent updates

Similarly, [131] proposed a tamper resistant scheme based on *dynamic code mutation* in which program's data and code in the process memory are regularly relocated such that code and data memory cells are indistinguishable. That is, a particular memory cell (which is allocated for the process memory) serves both data and code during the course of an execution. Dynamic code mutation aims for keeping the code unreadable until execution time. Therefore, we classified this scheme as *code invariants* protector.

[1]It is a technique in which program instructions are randomized such that only the shipped emulator can execute them [12]

The basic assumptions in this technique are that reverse engineering (and hence tampering with programs) is harder **a)** when program code and data are indistinguishable, and **b)** when there is no fixed mapping between instruction and memory cells so that the same memory region is used by multiple code blocks at a course of execution.

The general observation is that runtime transformations are technically much more challenging to implement, which justifies the limited number of schemes in this category.

Post-compile

Post compilation transformation operates at the binary level. That is, compiled programs and libraries, without presence of their source code, can be protected. Because of their applicability on majority of product line, plenty of schemes utilize post-compile transformation. In fact, we found protection schemes for all the system representations which carry out their transformations in a post-compile process. In the following, we elaborate on these schemes classified by their target representations.

Code invariants

Schemes that protect the code invariants (introspection) in a post-compilation transformation process are listed in the following. To protect integrity of Android applications [149] proposed a self-encrypting scheme that operates on the intermediate representation of Android programs, i.e. Dalvik executable files. Although the key is shipped with the program, this scheme adds resilience against tampering attacks. [22] applied self-checksumming transformation on windows portable executable (PE) binaries. [47] used cyclic network of checkers with repairers to recover tampered blocks to protect WIN32 (DLL and EXE) binaries. Similarly, [97] protected binaries based on self-checksumming technique. [34, 105, 115, 137, 189] proposed schemes that operate at the hypervisor level to protect program binaries. [26] used an isolation supported by Intel SGX to place the entire applications inside protected enclaves and protect them from attackers. [15, 87, 115, 180] proposed to encrypt program binaries in order to protect it. [88] designed a method that transforms program binary into a random instruction set to carry out protection.

Data invariants

We identified two schemes that protect data integrity in after compilation. In the scheme proposed by [83] multiple clones of a program are simultaneously executed to detect instances that diverge from the majority of the clones. Malone et al. in [132] proposed a scheme that has two phases: *calibration* and *protection*. In the first phase a program is executed while hardware performance counter (HPC) values are continuously collected. A mathematical tool is then utilized to identify hidden relations among the gathered data in form of equations, which essentially captures the effect of genuine execution of the program on HPC. Finally, a set of guards are injected in the program to evaluate those equations at different intervals. However, both Windows and Linux operating systems prohibit any access to HPC instructions in user-mode. This essentially requires applications to invoke a kernel call to read HPC values at runtime, which opens the door for call interceptions.

Static (file)

To protect the integrity of static files checksum based techniques were proposed in the literature. [114] introduced a tool to verify signatures of a large number of binaries on a system. [73] enabled a remote verifier to compute and verify checksums of static files residing on a server. In this model, the verifier sends a challenge with which the server has to initialize checksum variables. To protect operating system static services [43] proposed a technique to guarantee the integrity of standard operating system application by safeguarding core software package providers.

As a mean to protect specific services in a system, [146] developed a genuine geolocation provider service that is enclosed in a statically secured lightweight hypervisor.

Timed trace

[133, 161] are two consecutive schemes that authenticated the integrity of legacy systems (no multi core CPU) by measuring the time differences between genuine and forged execution of programs (timed traces). Similarly, [100, 101] also relied on timing analysis to detect malicious behaviors in remote programs. In these schemes, the verifier requests a checksum over certain regions of the process memory (memory printing) and at the same time measures the elapsed time. The checksum value and timing analysis enables the verifier to reason about integrity of the system. Nevertheless, during the memory printing process the system must stop functioning, otherwise the timing data will be inconsistent.

Trace

Trace representation is utilized by a set of protection schemes to protect control flow integrity. Abadi et. al in [2] proposed a scheme in which the integrity of control flow is verified by a simple token matching scheme. For this purpose, before every jump instruction a unique token is pushed into the stack. Subsequently, at all destinations (jump targets) a predicate is added to verify the token which should have been previously added to the stack. [99] leveraged invariant instruction length on x86 architecture to design a scheme with which tampering leads to an inevitable crash at runtime. [145] verified branching patterns to detect ROP attacks at the Windows 7 kernel, which can transparently protect all executing applications without any modification in them.

In the correlation of transformation lifecycle and protected representations, we classified schemes based on their representation-to-protect and their applicability on different program lifecycles. Our results indicates a gap in compiler based, load and run time protection schemes.

2.5.4. Correlation of transformation lifecycle and protection overhead

Another interesting direction to look into is the influence of the transformation lifecycle on the overhead of schemes. Compiler-level transformations leverage optimization passes in the compilation pipe line. Naturally, applying transformation at compiler-level should impose less overhead in comparison to post-compile transformation schemes. Mainly because binary level modifications occur at the end of the build pipe line, where no further optimizations are carried out. To capture this view we correlate overhead classes with the transformation lifecycles that is presented in Table 2.6. We map our surveyed papers on 4 classes of overhead indicators, viz. N/A, Fair, Medium and High.

Table 2.6.: The correlation between the transformation lifecycle and overhead.

		Lifecycle				
		Pre-compile	Compile	Post-compile	Load	Run
Overhead	Fair	[74] [112]	-	[2] [132] [55] [146] [137] [144] [87] [161] [88] [145] [189] [26]	[88]	[65]
	Medium	[19] [102] [174] [187] [190]	-	[82] [22] [180] [101]	-	[131]
	High	[108]	[171]	[99] [34] [100] [47] [149]	-	-
	N/A	[118] [41] [164] [98] [42] [51] [50]	[49] [104]	[73] [97] [133] [15] [115] [83] [114] [43]	-	-

Fair

As can be seen in Table 2.6, a majority of schemes with fair overhead are directly applied on the binaries, i.e. post-compile transformation. Precisely speaking, [88] with 10% overhead (in addition to the virtualization overhead which is reported to be about 30%) from the load category, [74, 112] with 8-10% overhead from the pre-compile category, and [65] with 4-23% overhead from the run category.

Medium

In our survey we identified 9 schemes with a medium overhead. In post-compile category [82] with 128%, [22] with 134% for non-CPU intensive applications and [180] with 107% are classified as schemes with a medium overhead. In the pre-compile category, [174], [187], [190], [19], and [102] are in the medium class. In the run category, [131] with an overhead of 107% (for a program with 70 protected basic blocks) falls under this overhead class.

High

In the high overhead class, we found 7 schemes for pre-compile and post-compile transformation activities. [108] imposes 5x slowdowns on reads and 8x slowdowns on writes.

Similarly, [149] introduces an overhead of 500%, [99] and [34] both are reported to impose 3x slowdown in protected applications. [171] utilizes an algorithm with complexity of $O(N^2 + 2^N)$.

N/A

We mapped all the schemes without performance evaluations or insufficient results into this class. A major problem with the performance evaluation of the reviewed protection schemes is lack of comprehensiveness. Thus, we are incapable of estimating the overhead of over 18 protection schemes. Surprisingly, the two compiler-level protection schemes [49, 104] also fall under this category.

This correlation once again shows that there is a gap in benchmarking protection schemes. To the extent that from our results we cannot approve nor reject the hypothesis that compiler-level transformations perform better.

We would like to emphasize here that this classification is just an estimation based on the authors' claim on the efficiency of their proposed schemes. Therefore, depending on what dataset was used in the evaluation process by the authors, the overhead might differ. A fair evaluation would be to use a constant dataset to measure overhead of different schemes.

Unfortunately, this is not possible due to two reasons. First, to the best of our knowledge, very few of the reviewed schemes were made open source. Secondly, even if the source codes were available we would not be able to compare their performance without further classifications. Because these schemes are designed for different operating systems and different representations. Thus, they need to be clustered technologically and then evaluated.

2.5.5. Check and Monitor representation

Protection schemes, as stated earlier, operate on a particular representation of the assets. This implies a sort of monitoring mechanism on the representation of interest. The collected data (in the monitoring process) need to be analyzed and ultimately verified by the scheme's *Check* compartment. Studying the different check methods for verifying different asset representations is particularly interesting. Table 2.7 depicts the correlation of check and representation items. Based on this correlation we report on different check methods that were used in the literature as follows.

Code invariants

Expectedly, the majority of code shape verifiers with a total number of 13 use checksum based techniques, which also includes hash functions In addition to checksums, [15, 115] employed signature matching as a secondary measure for tamper detection. [26, 34] maintained code invariants security by enforcing access control at a trusted hardware level.

Data invariants

In order to check the data shape, 3 schemes [83, 98, 169] used majority vote principle. On the other hand, [112, 132] used equation system to verify the integrity.

Hardware counters

[132, 165] used hardware performance counters in their checking mechanism.

Timed trace

[100, 101, 133, 161] computed a checksum alike routine and measure the elapsed time.

Table 2.7.: The correlation of the representation and check mechanism.

		Asset representation					
		Static	Code inv.	Data inv.	HWC	Timed trace	Trace
Check mechanism	Access control	-	[34] [26] [50] [174]	-	-	-	[19] [145]
	Checksum	[140] [73]	[50] [187] [190] [74] [97] [137] [22] [105] [104] [47] [149] [15] [88] [115] [180] [189]	-	-	[133] [100] [161] [101]	[102]
	Equation eval	-	-	[112]	[132]	-	-
	Majority vote	[72]	-	[169] [98] [83]	-	-	[102]
	Signature	[114] [43]	[15] [115]	-	-	-	[102] [51]

The correlation between checking mechanism in different protection schemes suggests that checksumming is the most used technique for integrity verification. Other checking measures such as signature verification, access control, equation evaluation and majority votes are equally unpopular.

2.5.6. Monitoring representation correlation with protection level (enforcement level)

In this subsection we are aiming to find out which representations could be monitored at which protection level. For this matter, we correlate *protection level* and *representation* in Table 2.8. Since we are interested in identifying the possibility of monitoring representations at different levels, we only indicate the number of citations instead of citing the references.

The correlation indicates that both the internal and external protection levels have no limitations on monitoring system representations, with the exception of traces which were not targeted by any of external protectors in the reviewed literature.

Furthermore, the correlation shows a gap in the hypervisor-based protection techniques in which none of the data shape, hardware counters and traces were used in the reviewed

Table 2.8.: The correlation of the Representation and protection level.

	Static	Code invariants	Data invariants	HW counters	Timed trace	Trace
Internal	2	13	3	1	-	8
External	5	5	4	1	5	2
Hypervisor	1	5	-	-	-	-

schemes. It is not clear whether this is due to a technical limitation of hypervisors or simply a research gap. Further studies should be conducted to address this gap.

2.5.7. Hardening vs. reverse engineering attacks and tools

Attacks perhaps are the least studied (and published) part of the integrity protection schemes. This is due to two main factors: **a)** plenty of security measures in industrial protection schemes such as Themida and VMProtect are not publicly disclosed, which in turn, has left researcher without enough knowledge about their security. **b)** Breaking schemes is unethical as it opens the gate for the attackers to compromise system's security. For example, the CIA Vault7 windows file sharing exploit has led to WannaCry ransomware which infected 213,000 windows machines in 112 countries [16].

Due to the lack of comprehensive research on attacks on integrity protection schemes, evaluating their security against various attacks is impossible. Therefore, as an initial step towards analyzing the security of these schemes, we made assumptions regarding the resilience of protection schemes against different attacks. In the following we state our assumptions:

1. **Disassembler:** code concealment and layered interpretation hardening measures hinder disassembly. Code concealment techniques commonly affect the correctness of static disassemblers [126].

2. **Debugger:** in a strict sense only schemes that directly utilize anti-debugger measures impede debug based attacks. These technique are, however, ad-hoc [3]. In this work, we consider hardening measures that impose some difficulties on the ability of debugging programs as counter debug measures. Cyclic checks, layered protection and mutation are the three hardening measures that we believe impede debug based attacks. Cyclic checks confuse attackers and exhaust their resources. Layered protection complicates the program execution flow and thus has an impact on the effectiveness of debugging. Mutation-based techniques renders debug knowledge useless after each mutation. Furthermore, SGX protected application cannot be debugged in production.

3. **Tracer:** enable attackers to monitor what instructions are actually being executed by the program. With the help of this tool, attackers could effectively bypass code concealment and virtualization as program instructions have to be translated and eventually executed. This is the point that a tracer can dump the plain instructions.

Table 2.9.: MATE tools resilience in the reviewed schemes.

Disassembler	Debugger	Tracer
[15, 19, 22, 51, 131, 149, 174, 180, 190]	[22, 26, 34, 47, 50, 55, 74, 82, 87, 88, 98, 104, 115, 131, 137, 174, 189, 190]	[26, 50, 65, 88, 99, 118, 131, 174, 187, 189, 190]

Nevertheless, frequent program mutation renders captured data useless. Furthermore, code clones are normally harder to capture using a tracer and thus it hinders tracer based attacks. *Intel SGX isolation*, by design, resist against tracers.

4. **Emulator:** enables adversaries to analyze protected programs in a revertible and side-effect free environment. In effect, this defeats response mechanism and gives unlimited attempts to attackers. Beside anti-emulation measures, which are again ad-hoc, only hardware based security appears to impede emulators. However, emulation on its own does not break an scheme, as attackers have to utilize other tools to detect and ultimately defeat the protection. Simply put, emulation acts as a facilitator for attacks. Therefore, we exclude emulator resilience from our analysis.

With the given assumption we have classified the reviewed literature based on their hardening measures. Table 2.9 represents the mapping between different schemes and their resilience to MATE Tools.

2.5.8. Resilience against known attacks

As mentioned earlier, the security of integrity protection schemes has not been thoroughly evaluated, due to the obscurity and lack of access to the resources. This has led to very few attempts on breaking such schemes.

To the best of our knowledge, three generic attacks on integrity protection schemes were published, two technical attacks and one conceptual work for manifesting attacks. In the following we discuss these attacks and map them to the hardening measures that could potentially address them.

1. **Pattern matching:** There is no paper on pattern matching attacks on different protections schemes. Additionally, most of the articles consider obfuscation as the golden hammer against pattern matching and entropy measurements. Unfortunately, we cannot evaluate the resilience of the reviewed schemes against pattern matching as their source code is not available. Therefore, we do not consider this attack in the mapping process.

2. **Memory split:** Wurster et al. in [182] proposed an attack to defeat self-hashing protection schemes by redirecting self-reads to an untampered version of the program loaded in a different memory address. For this purpose, they modified the kernel to

establish two distinct memory pipelines for self-read and instruction fetch (execution). In this setting, the execution pipeline can readily be tampered with by attackers and self-reads remain unaware of such modifications.

To address this generic attack, Giffin et al. in [89] proposed self-modifying code defense mechanism. The idea is to add a token to the program code at runtime and verify it accordingly. Dynamically generated tokens can detect redirection of self-memory reads in the protected processes.

Self-modifying code, despite its effectiveness in thwarting memory split attacks, opens the door for other integrity violating attacks. The reason being that enabling self-modification requires flagging writable memory pages as executable pages in the program. This raises the concerns about code injection attacks [11], in which maliciously prepared memory blocks could potentially get executed.

This attack only applies to introspection based techniques, i.e. techniques in which a program's code blocks have to be read at runtime. Thus, state inspection based techniques are resilience against memory split attack. In addition to that, self mutating protection schemes express the same effect as the defense that was proposed by [11]. All in all, our survey shows state inspection based and mutation based techniques mitigate the memory split attack.

3. **Taint analysis to detect self-checksumming:** Qiu et al. in [150] proposed a technique to detect self-checksumming guards in programs. Their attack relies on the fact that these class of protections forces a protected program to read its own memory at runtime. This obligation enabled them to utilize dynamic information flows in detection of self checksumming guards and check conditions (which trigger the response mechanism). Technically speaking, their approach captures a trace of the program (using Intel Pin) and then searches for a value X which is tainted by self memory values (backward taint analysis) and then detects those Xs that are used in a condition (forward taint analysis).

   This attack appears to be the ultimate attack on self-checking protection schemes as it can in theory detect all checks in a given program. However, tracing medium to large sized programs will generate massive trace logs that have to be analyzed. [22] reported analyzing a small trace (captured in 10 minutes) of Google Chromium using the tainting technique requires a long time in which only 1% of the traces were analyzed in a day.

   Since this attack relies on a Tracer tool (Intel Pin), we can mark schemes that resist against tracer as resilient to the attack presented in [150]. Needless to say that larger programs, due to the complexity of the backward and forward taint analysis, in general cannot be targeted by this attack in a reasonable time.

[1]Intel pin is a dynamic program instrumentation tool

Table 2.10.: Resilient schemes against the two known technical attacks.

Taint based attack	Memory split attack
[26, 50, 65, 88, 99, 118, 131, 174, 189]	[2, 19, 31, 34, 41, 42, 49, 65, 83, 88, 98, 99, 100, 101, 102, 108, 112, 131, 132, 133, 137, 144, 145, 161, 164, 169, 171, 174]

4. **Graph based analysis:** Dedic et al. in [72] indicate that flow graph analysis such as pattern matching and connectivity analysis (given that checker nodes are weakly connected to other graph nodes in a program) can help defeat the existing protection schemes. They developed a graph based game to formally present these attacks.

   Against these attacks, they proposed three design principles, viz. *strongly connected checks*, *distributed checks* and *threshold detection schemes*, that can significantly harden the identification and disabling of a tamper protection technique. Strongly connected checks requires checkers to be strongly connected to other nodes of the program. Distributed checks refers to a network of checkers which an attacker has to disable them all in order to successfully tamper with an application. Threshold detection schemes suggests to call a response function only when a k number of checks failed. This prevents attackers from sequentially detecting checkers in the program.

   This work can be seen as a conceptual work with no actual implementation of the defenses or attacks. It would be interesting to employ the suggested measures and carry out the attack to verify their claim. None of the reviewed schemes appears to utilize all the defenses together. Therefore, it might be the case that in theory all of the schemes are defeated by the graph based attack. Since this attack has not been implemented in practice, we rather leave it for further evaluations.

In Table 2.10 we map the schemes that resist against the two practical attacks.

2.6. Related Work

The closest and yet distant research to our work is ASPIRE (`www.aspire-fp7.eu`) project. It is a project funded by European Union for software protection. ASPIRE is designed to facilitate software protection by introducing a reference architecture for protection schemes. Conformance to this architecture enables practitioners to compose a chain of protections simultaneously, which offers more resilience [170]. Data hiding, code hiding, tamperproofing, remote attestation and renewability are the core protection principles that are addressed in ASPIRE.

ASPIRE introduces a security policy language expressed by annotations. To protect a program using this framework, end-users have to annotate programs with their desired security properties. A compiler tool chain is designed which analyzes these annotations and carries out necessary protection steps. To do so, the tool chain comes with three components: a source code transformation engine powered by TXL [68], a set of routines to link external protection schemes compatible with standard compilers (LLVM and gcc), and a link-time binary rewriting infrastructure for post-compilation protections powered by Diablo (`http://diablo.elis.ugent.be/`). All these enable rapid development of protection tools which could be used by different programs in a customizable and yet composable manner.

The ASPIRE framework focuses on a technical architecture for design and employment of a wide range of protection mechanisms (not only integrity protection). However, both the programs to protect and the protection schemes have to conform to the requirements of the tool chain. Specifically, the source code (in C/C++ language) of programs is mandatory. Furthermore, constraints of each scheme need to be specified in its manifest. The framework uses these manifests to report potential conflicts to users based on which they can identify and subsequently single out conflicting measures.

The ASPIRE project has not provided a taxonomy of integrity protection schemes and does not include comparison of different schemes. In fact, the outcome of this work can contribute in extending the ASPIRE architecture to accommodate a wider range of integrity protection schemes, namely by means of introducing new requirements for the annotation language (such as desired resilience against certain attacks), and extensions for the support of further protection schemes (such as hypervisor based and hardware assisted ones).

To the best of our knowledge, our work is the first taxonomy of integrity protection schemes. Therefore, reviewing existing taxonomies was not an option for us. Instead, we reviewed scheme classifications and surveys on integrity protection. In the following we report on the reviewed publications.

2.6.1. Existing Classifications

Mavrogiannopoulos et al. in [134] propose a taxonomy for self-modifying obfuscation techniques, which essentially is based on the concept of mutation.

Collberg et al. in [58] classify integrity protection techniques into four main categories: *self-checking*, *self-modifying*, *layered interpretation* and *remote tamper-proofing*. In a similar effort, Bryant et al. [38] classify tamper resistant systems into 6 categories, viz. hardware assisted, encryption, obfuscation, watermarking, fingerprinting and guarding. Considering the fact that watermarking on its own does not contribute to integrity protection, we exclude it from our analysis. Guarding also fits the self-checking category, thus it can be omitted. Likewise encryption is a substance of self-modifying primitive. After we resolved the conflicts, we study the structure of the five remaining categories of integrity protection techniques as follows.

Self-checking. This refers to a technique in which self-unaware programs are transformed into somewhat self-conscious equivalent versions. The transformation is done via equipping programs with a set of monitoring probes (checkers/testers), that monitor *some authentic features* of the target program, along with a group of assertions, who compare probe results to the corresponding *known expected values* and take *appropriate actions* against tampering attacks.

Depending on which features of a program are being monitored, self-checking itself has two subcategories: *Introspection* and *State Inspection*. The former exercises the shape of a program, for instance code blocks, whereas the latter is after monitoring the program's execution effects, for example the sum of a program's constant variables. Among the two, state inspection is more appealing because, unlike the introspection, it reflects the actual execution of a program, and thus it is harder to counterfeit. However, monitoring dynamic properties of a program is more difficult compared to the static code shape verification.

Self-modifying. Turning a program into a running target by mutating it at runtime is the idea behind self-modifying techniques. The majority of schemes that are based on this technique use encryption as the mean to mutate and ultimately to protect a program. However, there are some schemes that use instruction re-ordering, instead of encryption, to mutate the program.

Mavrogiannopoulos et al. in [134] define four main criteria in their proposed taxonomy for self-modifying obfuscation techniques: *concealment* (the size of program slices that are being modified at each mutation interval), *encoding* (the technique that is used to mutate the code, e.g. instruction reordering or encryption), *visibility* (whether the code is entirely or partially protected) and *exposure* (whether the actual code is permanently or temporarily obtainable at runtime). Nevertheless, in a more recent survey, virtualization was also proposed as a relevant technique in [186], but we rather consider virtualization as a subcategory of layered interpretation techniques.

Sasirekha et al. in [135] review 11 software protection techniques. Their research confirms the trade off between and security and performance for all the reviewed schemes based on which they suggested compiler-level protection as possible direction for further research. However, their work lacks any classification of techniques.

Layered interpretation. In this approach a program's instructions are replaced with a set of seemingly random byte codes. These codes can only be executed with a program specific emulator that is normally shipped with the program itself. Tampering with the program byte code may result in unrecognized byte code in the emulator and may eventually causes failures. However, some techniques (for instance [115]) perform integrity verifications at the emulator prior to executions. In general, layered interpretation techniques add resilience against static program analysis attacks. To the best of our knowledge, there is also no published taxonomy on layered interpretation protection methods.

Remote tamper-proofing. Protection schemes that enables a software systems to be verified externally fall under this category. In an abstract sense, remote tamper proofing has very much in common with *remote attestation* concept. Since remote attestation is a vast field of research that deserve a taxonomy on its own, its classification falls out of the scope of this work.

2.6.2. Comparison of schemes and surveys

Dedic et al. in [72] define *distributed check* and *threshold based detection* as significant resilience improver factors in integrity protection schemes. They use a graph based game and theoretically prove the enhanced complexity of the scheme with the aforementioned factors. From this work two interesting features are extracted: *detection* and *response algorithms*.

In an effort for security classification and unification of protection notation, Collberg et al. after defining a notation, use natural science and human history to derive a set of 11 unique defense primitives in [64]. These primitives state the defense concepts rather than security guarantees.

In an effort for MATE attack classification, Akhunzada et al. in [10] study different types and motives behind these attacks. As another outcome of their work, they stressed that performance factor is one of the main pain points of protection techniques. However, it appears that no study tackled this matter and there is a lack of a benchmark for protection schemes [10]. The focus of their work is not the techniques-to-protect themselves, but rather the attackers. Consequently, the security guarantees are not studied.

The main gap here is lack of a holistic view software protection process. The process itself is not well defined, business requirements are not elicited, constraints are not studied, and more importantly, security guarantees that different schemes offer are not studied.

2.7. Conclusions

MATE attackers have successfully executed serious integrity attacks far beyond disabling license checks in software system, to the extent that the safety and security of users is at stake.

Software integrity protection offers a wide range of techniques to mitigate a variety of MATE attacks. Despite the existence of different protection schemes to mitigate certain attacks on different system assets, there is no comprehensive study that compares advantages and disadvantages of these schemes. No holistic study was done to measure completeness and more importantly effectiveness of such schemes in different industrial contexts.

In this work, we therefore presented a taxonomy for software integrity protection which is based on the protection process. This process starts by identifying integrity assets in the system along with their exposure in different asset representations throughout the program lifecycle. These steps are captured in the system view, which is the first dimension of our taxonomy. In the next dimension, the attack view, potential threats to assets' integrity are analyzed and desirable protections against particular attacks and tools are singled out. Then, the defense dimension sheds light on different techniques to satisfy the desired protection level for assets at different representations. This dimension also serves as a classification for integrity protection techniques by introducing 8 unique criteria for comparison, viz. protection level, trust anchor, overhead, monitor, check, response, transformation and hardening. To support the understandability of the taxonomy we used a fictional DRM case study and mapped it to the taxonomy elements.

We also evaluated our taxonomy by mapping over 49 research articles in the field of integrity protection. From the mapped articles we correlated different elements in the taxonomy to address practical concerns with protection schemes, and to identify research gaps.

In the correlation of assets and representations on which protections are applied we identified relevant schemes for protecting different integrity assets (behavior, data and data and behavior). This facilitates the scheme selection based on the assets and representations at the user end. A major gap was identified in protecting integrity of distributed software systems.

In MATE vs. representation correlation we have shown different schemes that mitigate different MATE attacks, viz. binary, control flow, process memory and runtime data, at different asset representations.

In the correlation of transformation lifecycle and protected representations, we classified schemes based on their representation-to-protect and their applicability on different program lifecycles. Our results indicates a gap in compiler based, load and run time protection schemes.

To study the impact of the transformation lifecycle on the protection overhead, we correlated transformation lifecycles and scheme overheads. Despite the general belief that compiler based protection should perform better, our results suggest that post-compilation transformations perform quite optimal. Nevertheless, we were unable to verify the performance of compiler based protections due to lack of benchmark results. We believe the lack of reliable benchmark data as well as a benchmarking infrastructure is a significant gap in software integrity protection. This is due to the fact that only implementation of a few protection schemes were made public, for instance in self-checking schemes only [22] has published their source code. Thus, evaluating their performance, without re-implementing them, is infeasible.

The correlation between checking mechanism in different protection schemes suggests that checksumming is the most used technique for integrity verification. Other checking measures are equally unpopular. The correlation of representation and protection level shows a gap in hypervisor based protections.

Another major gap in the literature is to analyze resilience of protection schemes against attacks. Based on our classification data we evaluated the resilience of reviewed schemes against MATE tools and known attacks on integrity protections. In the resilience against MATE tools analysis, we proposed a set of assumptions for declaring whether a scheme resists again a certain MATE tool or not. The assumptions are based on the hardening techniques that are used by protection schemes. These assumptions were then used to classify schemes based on resilience against MATE tools.

In the last correlation we analyzed the resilience of reviewed schemes against known attacks on protection schemes. Our Survey indicated that there is a gap in studying the resilience of protection schemes against pattern matching attacks. This is again due to a lack of any benchmark of different techniques whatsoever. Regarding memory split and taint based attacks, we first identified hardening measures that impedes such attacks. Subsequently, we marked schemes as resilient to memory split and taint based attacks according to their hardening measures.

All in all, a big gap in integrity protection research is the lack of a benchmark of performance and security guarantees, both of which require a dataset of integrity protection mechanisms.

Limitation: Our taxonomy emphasizes the defense mechanisms, as the number of published attacks on integrity protections is incomparably fewer than published defenses. Mapping concrete attacks will be a good extension to our taxonomy.

Although we included some research papers on infrastructural security, our analysis's main focus was application-level integrity protection. In reality, there are plenty of ways in the infrastructure that adversaries can potentially exploit. Such exploits could lead to circumvention of defense mechanisms and a potential violation of the application integrity. For instance, one major class of attacks that we did not cover is prevention of call hooking attacks [1]. Another interesting direction is to map infrastructural integrity protection measures and known attacks on them to the taxonomy.

Part II.

Software Integrity Protection Enhancements

3. Practical Integrity Protection with Oblivious Hashing

This chapter presents practical enhancements corresponding to fundamental problems of the Oblivious Hashing integrity protection technique, namely coverage, and input dependency. The entire content of this chapter was previously published in work co-authored by the thesis author [5].

3.1. Introduction

Chen et al. proposed a stealthy protection technique (*Oblivious Hashing (OH)*) that neither relies on reading process memory nor is deceptable by the memory piping trick [49]. OH captures evidence of the actual execution of programs by computing a trace hash over the values stored in memory corresponding to arbitrary read and write accesses in a program. These hash values are then matched with expected ones during execution. The expected values have to be precomputed and stored in the programs upon distribution. Simply put, OH checks the execution effects in memory, as opposed to checking the code itself. Therefore, it is harder to identify (stealthy) or trick (resilient) OH than SC. Better stealth and resilience makes OH a better protection technique than SC.

Although OH is hence a more appealing protection measure in principle, it is unable to protect nondeterministic program traces, i.e., memory accesses with values dependent on user-input or environment variables. This is mainly because hashing such traces results in different hashes for different program executions. To remove any nondeterminism from trace hashes, [48] proposed a technique in which all variables in traces are canceled out by enforcing a reversible commutative function (e.g., addition) as the hash function. This, however, enables attackers to patch variables to any desired value, given that the modified value will be hashed and subsequently canceled out by the protection.

Gap. An effective strategy to cope with nondeterminism requires addressing two challenges - **i)** how to automatically extract deterministic program segments, and **ii)** how to protect nondeterministic ones. The second problem is more striking as a large portion of program traces are expected to be nondeterministic. For instance, only 13% of program instructions evaluated in [159] are deterministic. This percentage is far less for the programs in our dataset: a mere median of 0.5% as shown in Table 3.1. More importantly, to the best of our knowledge, no comprehensive study was done on the security, resilience and overheads of OH on real-world programs.

Contributions. We turn OH into a practical integrity protection technique. We **i)** develop a method for an automated detection of data and control flow dependent instructions in programs; **ii)** propose the short range oblivious hashing (SROH) technique as an extension to OH enabling protection of instructions residing in nondeterministic branches; **iii)** utilize a complementary self-checksumming-based technique to protect data dependent segments; **iv)** further harden the complementary protection by intertwining it with OH; **v)** enable OH to implicitly protect (nondeterministic) data dependent instructions; **vi)** employ our protection scheme on a dataset of 29 real-world applications to evaluate the security as well as the induced overheads; and finally **viii)** open source our cross-platform protection framework prototype written in LLVM.

3.2. Background & Related work

3.2.1. Software integrity protection

Protection schemes are comprised of three parts - i) *check*, ii) *respond*, and iii) *harden* [9]. Checking activities monitor a particular set of properties of a program to ensure that they conform to a *known good* state. Response activities react when the current state differs from the good state by, for instance, covertly terminating the program [64]. Hardening refers to a set of techniques that are used to add resilience to the *check* and *response* routines, making them harder to find or disable.

Authenticating accesses to sensitive data is one of the goals in integrity protection. Banescu et al. [19] proposed an access proxy that authenticates legitimate modifications on sensitive data. Legitimacy of such modifications are verified by means of authenticating the control flow and subsequently signing the data using a white-box HMAC [184]. Similarly, [31] proposed a scheme that introduces a light-weight secondary process for verifying the program control flow integrity. Such techniques primarily detect tampering attacks on the control flow. Consequently, attackers can arbitrarily manipulate the logic in a program as long as the control flow is preserved.

In order to ensure code integrity, a number of schemes such as [14, 149, 179] have been proposed. These impede tampering attacks by encrypting the code in a way such that only a genuine execution could decrypt the code and subsequently execute it, which is enforced by a trace-sensitive key derivation technique. However, the program has to eventually get decrypted in order to be executed. Therefore, well-timed memory dumps could defeat the protection.

State inspection is another type of integrity protection in which dedicated program invariants are verified. Ibrahim et al. [98] utilized a set of checkers to verify return values of the sensitive functions for arbitrary inputs as invariants. Their approach works for pure functions only as opposed to other protections. Andriesse et al. [13] used Return-Oriented

Programming gadgets to craft stealthy checkers. These checkers are bootstrapped with a simulated buffer-overflow attack. However, shipping a program with buffer-overflow vulnerabilities may open the door to other attacks. Moreover, anti-ROP techniques such as [145, 191] may prevent the execution of such verifiers. The hash of code segments (also known as self-checking) is another example of invariants that was used by a multitude of schemes [22, 47, 87, 88, 97]. Self-checking techniques, however, are unaware of the program execution environment and thus vulnerable to the memory split attack [183]. Moreover, they introduce atypical behavior (self-memory access) in programs, which in turn may make them vulnerable to the taint-based attack [150].

Another property of the program that can potentially counter tampering attacks is execution time, which was used by Jakobsson et al. [100, 101] to detect compromises. Pioneer [161] and Conquer [133] used the same idea to protect legacy systems. The mathematical relations among hardware performance counters were used as a clue of tampering attacks by Malone et al. [132]. All these schemes are greatly system-specific which requires individual tailoring for every execution host.

Oblivious hashing (OH) [49, 99] combines the deterministic memory values referenced by instructions in a program's execution trace by means of hashing. Given that these values are invariant for different runs, OH can verify the computed hashes in various parts of the program. This technique offers in-execution protection without introducing odd behaviors such as code segment accesses. Therefore, it is more appealing to use in practice than SC. Despite the benefits, OH falls short in protecting instructions working with nondeterministic data, which we tackle in this work.

3.2.2. Nondeterministic code detection

Detection of instructions that receive (or depend on) input/state in applications is challenging. Such inputs need not necessarily be entered by users. System state data (e.g., date and time, CPU heat measurements) have the same effect. We will use the term "nondeterministic data" throughout this chapter for data that is provided by the environment or user and that can vary across different executions.

Inconsistent hashes primarily occur when a program p processes some inputs and/or responds to some environmental states that influence OH computations. The influence can be in two forms: either an input (state) is directly incorporated into hashes or it contributes to a branch condition.

A naive idea to detect input-independent instructions is to run the program of interest with all possible inputs/states and identify the intersection of all executions [49]. However, running programs with all possible inputs/states, considering the changes over time and program size, does not scale. Moreover, with a manual procedure, the accuracy of results may decline.

To cope with this issue in a more systematic manner, we need to distinguish between two types of nondeterminism: **i)** *data dependency* which captures instructions where at least one of their operands depends on nondeterministic data, and **ii)** *control-flow dependency* which includes instructions in branches with a condition that depends on nondeterministic data.

There are multiple techniques that could potentially be used to single out nondeterministic instructions:

Information flow analysis. This is a technique that tracks data propagation (from sources) to any function of interest (sinks), e.g. standard output [168]. The difficulty of using the existing information flow analysis tools is on the one hand in specifying all the sinks in a program. In addition, here our objective is to identify all the (intermediate) instructions, the operands of which are either tainted with nondeterministic data or the execution branch of which depends on such data. However, information flow analysis addresses whether tainted data flowed into arbitrary (unsafe) functions or not, and thus cannot directly be used to identify nondeterministic instructions.

Program slicing. Program slicing identifies a set of instructions (slice) that may affect (backward slicing) or be affected by (forward slicing) the values of arbitrary variables at a given point of time [173]. In order to filter nondeterministic instructions (short for: instructions that depend on nondeterministic input), one can use (forward) slicing to compute those instructions that may be affected by input data. This, however, requires marking all the input variables in a program, for all of which slicing needs to be undertaken. Moreover, slicing does not segregate data and control-flow dependencies, which is necessary for our protection scheme.

User-input dependency detection. Dependence graphs capture dependency relations (both data and control-flow) among program instructions. Given the dependence graphs, identifying instructions that depend on nondeterministic data is solvable by a graph reachability analysis. We found two techniques for this sort of analyses [159, 163]. The technique proposed by Scholz et al. [159] also takes the call-site dependency status (of a function) into consideration in its so-called *call-sensitive analysis*, which in turn makes it the best candidate for our nondeterminism detection. However, their tool is not publicly available[1], which is the reason we had to re-implement it from scratch.

3.3. Design

Throughout this section, we use a sample electricity meter application as our running example, which uses varying rates depending on the time of day. The meter in Listing 3.1 charges users twice the rate for peak hours, half the rate for off-peak hours, and the normal rate for holidays and non-peak hours. The sensor calls the `meterUsageCycle` function on an hourly basis to report consumptions (`*kwMinute` argument) in kilowatt minutes, their count (`size`), and the period of usage (`rate` argument).

[1]We contacted the authors on May 4, 2017 asking for a copy of their code, but we have not yet received a reply.

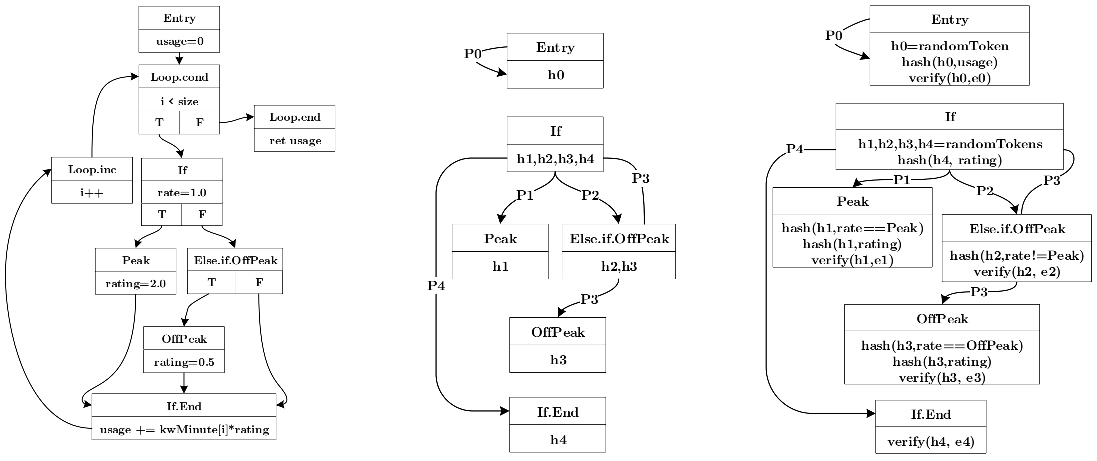
Listing 3.1 Fictional electricity meter application with varying rates

```
enum period {Peak, OffPeak, Normal};
float computeUsage(float *kwMinute, int size, enum period rate){
  float usage = 0;         //III
  for(int i=0;i<size;i++){         //DDI
    float rating = 1.0;         //DII|CFDI
    if(rate == Peak){         //DDI
      rating = 2.0;         //DII|CFDI
    } else if(rate == OffPeak){         //DDI
      rating = 0.5;         //DII|CFDI
    }
    usage += kwMinute[i]*rating;         //DDI
  }
  return usage;         //DDI
}
...
meterUsageCycle(float *kwMinute, int size, enum period rate){
  kwHour = 0;         //III
  if(isHoliday()){         //DDI
        enum period normalRate = Normal;         //DII|CFDI
        kwHour = computeUsage(kwMinute, size, normalRate);         //DDI|CFDI
  } else {
        kwHour = computeUsage(kwMinute, size, rate);         //DDI|CFDI
  }
  ...
}
```

3.3.1. Segregation of input data/control-flow dependent instructions

Our protection requires identifying all values that are tainted (from any input sources) with nondeterministic data (data-dependent) as well as branches whose execution relies on such data (control-flow dependent). As discussed in Section 3.2.2, the problem of nondeterministic behavior identification essentially is a reachability problem in graphs. Therefore, we base our segregation technique on the call-sensitive user-input dependency detection technique [159]. Because a detailed discussion of the analysis is out of the scope of this work, in the following we abstractly discuss the high-level idea only and refer to the original publication [159] for details.

The first step is to construct dependency graphs [80] for a given program. Nodes are program instructions and edges indicate data and control-flow dependencies among them.



(a) `computeUsage`'s CFG representation

(b) `computeUsage`'s paths

(c) `computeUsage`'s short range hashes & verifications

Figure 3.1.: Applying SROH to the `computeUsage` function of the sample electricity meter program

That is, assignments yield data dependency edges while calls and branches yield control flow edges. The location where functions are called (call site) together with the arguments that are actually passed to them affect the input dependency of their instructions. To adequately propagate call-site dependencies, a strongly connected component analysis is run on the program call graph. Then, multiple dependence graphs are constructed for each strongly connected component. These graphs are then analyzed (in a reverse topological order) to propagate dependencies, starting from functions that are called at the bottom of the call graph. The outcome of the analysis indicates data and control-flow dependencies. In Listing 3.1, comments indicate the input dependency of instructions: *III*, *DII*, *DDI* and *CFDI* denote input-independent (both data and control flow), data-independent, data-dependent, and control-flow dependent instructions, respectively. *III* instructions can directly be protected using OH, e.g., the right-hand operands of the assignments at lines 3 and 17 in our sample program (Listing 3.1). Clearly, hashing memory references of both *DDI* and *CFDI* yields different hashes for different inputs.

3.3.2. Short Range Oblivious Hashing (SROH)

To enhance the explicit protection of OH, we propose a novel *Short Range Oblivious Hashing (SROH)* mechanism to cover nondeterministic control flows by OH. As seen above, hashing both DDIs and CFDIs may lead to inconsistent hash values. The incorporation of data-dependent variables into hashes yields unverifiable hashes. Similarly, covering instructions in input-dependent branches by OH may lead to two types of inconsistent hashes for different nondeterministic values: **i)** the expected hash cannot be precomputed because nondeterministic values lead to the execution of different branches, and **ii)** the expected hash cannot be computed as a result of taking some branches for an indefinite number of times depending on nondeterministic values.

To cope with these issues, we propose the SROH technique comprised of a path-specific set of hash variables and a tailored verification. Our technique can protect DIIs residing in nondeterministic branches. Moreover, it captures a holistic control flow integrity by reflecting the evaluation results of nondeterministic branches in its hashes. The idea is to realize OH using distinct hash variables for every ordered sequence of basic blocks (OSBB) that are strictly executed in the identified order, in *every* execution. That is, for a given block in the sequence, all the preceding blocks are necessarily executed before reaching the block. Another requirement is that hashes, in such sequences, are verified in one of the blocks within the sequence—conveniently, the last block. Of course, some blocks may appear in multiple sequences. Such blocks in the intersection of multiple sequences contain multiple hashes, and thus verifications.

The SROH technique effectively addresses the two aforementioned issues with hashing input dependent branches because - **i)** hashes are computed in the scope of identified sequences and, therefore, not taking a branch does not affect the expected hashes, and **ii)** the verification of hashes takes place *within* the identified sequences, and thus the number of times a branch is taken does not corrupt hashes.

We use two strategies for loops—*input-dependent* and *input-independent*, depending on whether a loop (condition) is input-dependent. Our SROH technique consists of four steps.

Step i. OSBB discovery. Dominator tree relationships [121] can be utilized to identify sequences of basic blocks that have the desired property, i.e., immediate dominance, from a control flow graph (CFG). We construct OSBBs at the granularity of functions; however, the same idea can be applied at the program module level.

Input-independent loops, the looping condition of which does not rely on any nondeterministic input, do not require any specific action. Input-dependent loops, however, need to be handled differently. Since the number of iterations of an input-dependent loop is determined by (nondeterministic) inputs, hashing loop-variant instructions—the value changes of which are propagated to other loop iterations—leads to inconsistent hashes. We ensure hash consistencies for such loops by abstracting their (indefinite) number of iterations. That is, we only consider their (loop) bodies; and, subsequently, loop condition, iteration and end blocks are excluded from OSBB computations. Moreover, we require every OSBB to end immediately before an input-dependent loop begins.

As an example, we apply the discovery technique on the computeUsage function of our sample program, whose CFG representation is depicted in Figure 3.1a. Figure 3.1b illustrates the identified OSBBs, namely *P0: Entry*, P1: If → Peak, *P2: If → Else.if.OffPeak*, *P3: If → Else.if.OffPeak → OffPeak*, and *P4: If → If.End*. Nodes indicate program basic blocks while edges capture the order in which those blocks are executed. Since OSBBs for input-dependent loops start from the beginning of loops, in Figure 3.1b the *Entry* block is treated as a standalone OSBB, and hence not as a part of the loop sequences.

Step ii. Distinct hash variables. Next, we create a distinct hash variable for every identified OSBB in its first block. These hash variables are used to protect the instructions residing in the basic blocks of a particular OSBB, which we refer to as available hash variables in the block. In our `computeUsage` example (Figure 3.1b), we identify five paths and therefore create five hash variables, *h0, h1, h2, h3, h4*. These hash variables are further mapped to the basic blocks to clearly indicate the available variables in each block. Some blocks (e.g., *If* and *Else.if.OffPeak* blocks in Figure 3.1b) are in common between different sequences and thus can have multiple available hash variables, one per each sequence that pass through them. In such blocks, SROH can randomly utilize a random subset of available variables, or all of them.

Step iii. Hash verifications. In this step, all the DII|CFDIs are protected by hash variables that correspond to their OSBB. That is, DDIs as well as loop-variant instructions within input-dependent loops are skipped. In the case of input-independent loops, loop variants can also be covered as long as they are data-independent. Furthermore, to reflect the control flow of the program into available hashes of each block, we incorporate the evaluation result of branching conditions.

After incorporating instructions into hashes, we need to ensure that OSBB-specific hashes match expected values. However, the verification of hashes with instructions that were *not* previously executed results in false alarms. For example, in Figure 3.1b verifying *h1* in the *If.End* block will raise a false alarm if the *Peak* block is not executed before. Hence, each block can only verify those hash variables that are available in it. Figure 3.1c demonstrates the OSBB-specific hashing and verifications for the `computeUsage` function.

Step iv. Precomputing expected values. All expected hashes (*e0, e1, e2, e3 and e4*) have to be precomputed and adjusted in protected programs before shipping them. One way to achieve this is by triggering the protected branches, for instance using targeted symbolic execution [130] in a monitor mode to capture the expected values. Alternatively, we can precompute OSBB hashes by slicing and further emulating instructions that are involved in the hash computation of each OSBB. A standalone emulation of paths is possible as no DDIs are protected in paths. We use the emulation-based approach as it performs faster than symbolic execution.

In Section 3.4, we present a full utilization of SROH together with OH on the electricity meter program for which SROH increases the instruction coverage of OH from 13% (2 instructions) to 66% (10 instructions).

3.3.3. Data-Dependent Instructions (DDIs)

For a resilient protection scheme, in addition to protecting III (by OH) and DII|CFDI (by SROH), we need to also protect DDIs. For instance, in our electricity meter example in Listing 3.1, line 11 indicates an unprotected DDI that computes the actual usage. Similarly, the `isHoliday` function plays an important role in the usage computation by enforcing the holiday tariff. These program segments can be seen as sensitive operations in our smart meter program that are, nonetheless, left unprotected because of their dependency on nondeterministic data, e.g., system time and data provided by the meter's sensors.

Our evaluation in Section 3.6 confirms that such instructions constitute a high percentage of program instructions. Hence, failing to protect them enables perpetrators to violate system integrity despite the added (partial) protection.

We hence propose to use a complementary protection technique for those instructions that OH/SROH cannot protect. Unlike OH, such an integrity protection technique must be unaffected by nondeterministic input. We refer to this layer of protection as *data dependent protector (DDP)*. Since SC protection monitors program code, not the execution, it is not affected by any input data. Therefore, we can use it to build the DDP layer. Applying the DDP protection to our sample program enables us to cover both the `isHoliday` function and the DDIs in the `computeUsage` function.

We base our DDP on self-checksumming protections similar to those discussed in [22, 47]. Our DDP is comprised of *protection guards*, *ideal hash functions*, a *response mechanism*, and a *hardening method*:

Protection guards. Code hashes are computed by a set of overlapping guards, each of which is dedicated to a consecutive set of DDIs. These guards are seeded with *offset* and *size* of those instructions of the program in memory. Hash functions are then utilized to hash the machine code bytes residing between *offset* and *offset+size* into an accumulator variable.

Ideal hash functions. The hash function itself need not have properties of a cryptographic hash function, e.g. collision-resistance is not as important in this context. Instead, it is crucial that the utilized hash function is stealthy and fast. A stealthy hash function must have a covert computation routine as well as low entropy outcomes. Since the expected hashes have to be incorporated into the binary, using cryptographic hash functions entails incorporating high entropy expected values. Linear time attacks can find such values and potentially their guards as well [54]. Therefore, we resort to a simple *XOR* hash function.

Response mechanism. In case of mismatches, a covert response mechanism is triggered by the respective guard. Based on the nature of a protected program, this response may vary from performance degradation, temporally delayed termination or a signal to a command & control server [47, 58, 64]. In some cases, no intrusive response whatsoever is desired; instead an evidence of violation of integrity shall be maintained for postmortem analyses. In our scheme, we leave the response routine itself completely configurable/customizable to meet different program requirements.

Hardening method. As a hardening measure, we craft a network of interconnected guards; i.e., there are guards that protect other guards, possibly in a cyclic manner [22]. Under the assumption that guards are diversified, i.e., do not syntactically resemble each other too much, in order to counter pattern matching attacks, one factor in measuring the difficulty of tampering with guarded segments is the number of guards that need to be defeated by attackers. This is known as the *connectivity* of a protection network [47]. Higher connectivities increase the resilience, but harm performance (more checks takes more time). In our protection, we let users define their desired connectivity via a parameter.

3.3.4. Intertwined protection

For additional resilience, we intertwine the DDP layer with the OH layer. A direct consequence of interleaving the two is an added resilience against tampering attacks on self-checksumming guards. In Section 3.6.5, we discuss the security enhancements in detail.

The idea is to have OH/SROH protect SC guards, which can also be seen as the implicit protection of nondeterministic segments by OH. The protection incorporates all (invariant) attributes of guards (i.e. offset, size, and expected hash) into OH/SROH hashes, possibly with the exception of the DDP's computed hashes, if one desires to mitigate taint-based attacks, see Section 3.6.5. This works because self-checksumming guards do not depend on inputs: they compute hashes over certain process memory blocks and compare them with expected values, which remain unchanged, independent of the program inputs. Listing 3.2 illustrates this layered protection. For simplicity's sake, in our example we demonstrate the response mechanism as a function call (i.e. `response()`). This is rather easy for attackers to detect and subsequently disable such calls. In reality a set of covert and diversified response routines shall be utilized.

Listing 3.2 Fictional electricity meter application with varying rates

```
...
OH_hash(OH_hash_var, offset);
OH_hash(OH_hash_var, size);
OH_hash(OH_hash_var, SC_expected_hash);
SC_computed_hash = SC_hash(offset, size);
if(SC_computed_hash!=SC_expected_hash){
  response();//tampering detected
}
...
OH_assert(OH_hash_var, OH_expected_hash)
```

It is worth noting that SC guards that reside in input-dependent branches in principle cannot be covered by OH. However, because of our SROH extension, OH can protect control-flow dependent branches, and thus all the SC guards.

3.4. A full example of OH+SROH utilization

We carry out our transformations in the LLVM IR representation. However, for the sake of clarity, we demonstrate our protection (OH+SROH) on the electricity meter sample program in C. Listing 3.3 presents the applied protection on the sample program.

Listing 3.3 Snippet of the fictional electricity meter application after the utilization of OH+SROH

```
enum period {Peak, OffPeak, Normal};
float computeUsage(float *kwMinute, int size, enum period rate){
long h0 = <random token>;
float usage = 0;                         //III
OH_hash(OH_hash_var, usage)
SROH_hash(h0, usage)
SROH_verify(h0, <expected value>)
for(int i=0;i<size;i++){                 //DDI
    long h1 = <random token>, h2=<random token>,
    h3 = <random token>, h4 = <random token>;
    float rating = 1.0;                  //DII|CFDI
    SROH_hash(h4, rating);
    if(t= rate == Peak, SROH_hash(h1, t), t){                        //DDI
       rating = 2.0;                          //DII|CFDI
       SROH_hash(h1,rating);
       SROH_verify(h1, <expected_value>)
    } else if (t= rate != Peak, SROH_hash(h2, t), t){
    //dummy if clause capturing the missing branch in the high-level representation
       SROH_verify(h2, <expected_value>)
    } else if(t = rate == OffPeak, SROH_hash(h2,t), t){          //DDI
       rating = 0.5;                          //DII|CFDI
       SROH_hash(h3,rating);
       SROH_verify(h3, <expected value>)
    }
    SROH_verify(h4, <expected_value>)
    usage += kwMinute[i]*rating;         //DDI
 }
return usage;                          //DDI
}
OH_hash_var = <random token>
meterUsageCycle(float *kwMinute, int size, enum period rate){
        kwHour = 0;                    //III
        OH_hash(OH_hash_var,kwHour);
        long h0 = <random token>, h1=<random_token>;
        if(t = isHoliday(), SROH_hash(h0,t), t){          //DDI
                enum period normalRate = Normal;         //DII|CFDI
                SROH_hash(h0,normalRate);
                kwHour = computeUsage(kwMinute, size, normalRate);//DDI|CFDI
                SROH_verify(h0, <expected_value>)
        } else if (t= !isHoliday(), SROH_hash(h1,t), t){ //DDI
                SROH_hash(h1,normalRate);
                kwHour = computeUsage(kwMinute, size, rate);         //DDI|CFDI
                SROH_verify(h1, <expected_value>)
        }
        OH_verify(OH_hash_var, <expected value>)
}
```

3.5. Implementation

We implemented the entire protection tool chain for LLVM 6.0, aiming for a cross-platform protection. Our entire tool-chain is publicly available at `https://github.com/tum-i22/sip-oblivious-hashing`. In this section, we briefly discuss some highlights of our prototype.

3.5.1. Protection process

The process takes as input the program that we wish to protect. To limit the scope of the protection application, a set of sensitive functions can be specified in the configuration, for instance, to induce less performance overhead. If such functions are provided to the protection tool, all protections (OH, SROH, SC) are applied to them only, with the exception of some additional functions that are picked by SC to satisfy the desired connectivity level (see Section 3.5.5). Additionally, users can provide their desired response routine to the tool chain. This could be easily extended to supply a set of diversified and covert response routines.

Next, the program is analyzed to identify data and control-flow dependencies. This information is required in all subsequent steps. The program is then subjected to the SC pass, where a set of interconnected SC guards is created to protect DDIs.

An OH transformation pass then protects the SC guards and the application's IIIs. The former accomplishes part of intertwining OH and SC. That is, SC guards residing in input independent segments get protected by OH. SROH further extends the OH coverage by protecting DII|CFDIs. It effectively covers the remaining SC guards and thus completes the process of intertwining OH and SC.

The `SC-Post-Patching` process then adjusts SC guards accordingly (see Section 3.5.5). Finally, the `OH-Post-Patching` process runs the binary and makes the final adjustments by fixing expected hashes in the OH and SROH guards (see Section 3.5.3). The outcome of the process is a protected binary in which its DDIs and IIIs as well as the newly created SC code is protected.

3.5.2. Input dependency detection

Our analysis pass, besides identifying input dependencies, distinguishes between data and control flow dependencies. This is necessary for the implementation of OH, SROH, and even SC.

The analyzer relies on function definitions to identify sources of input dependencies. This is problematic as many functions may come with no definitions, e.g., call sites of external functions. In such cases, the analyzer makes conservative assumptions, i.e. marks instructions as input-dependent if it cannot prove otherwise. The same holds true for the converse situation, if a local function is passed as a callback argument to an external function. In this case, we miss the external caller's input dependency information.

To cope with these limitations, users can fine tune these assumptions in configuration files. Such files primarily contain information about the input dependency of external functions and call-back arguments. Users will have to create them once, but may use them repeatedly for different programs that are based on the same protected code base.

3.5.3. Oblivious hashing (OH)

Our OH prototype has a parametrized number of hash variables which are then randomly used throughout the program. We also implemented two different hash functions (XOR and CRC32) to be randomly utilized, which can easily be extended.

To realize OH, we create hash calls with hashable instructions/values to available hash functions and incorporate outcomes into hash variables randomly. We consider the `CMP`, `Load`, `Store`, `Call`, `GetElementPtr` and `Return` LLVM instructions as our hashable candidates. To incorporate the `CMP`, `Load`, `GetElementPtr` and `Store` instructions, we hash their right-hand-side operands. For `Call` instructions, we take both arguments and return values. `Return` instructions are protected by incorporating their operand (return value) into hashes.

Bear in mind that we ensure consistent hashes by skipping all instructions and operands that are DDI or CFDI. These instructions are marked by the input dependency analysis.

After each hash call, OH places a verification guard that checks if the value of the hash variable matches an expected value.

Statically precomputing expected hashes may require backtracking a chain of values in the program. We instead resort to placeholders that will later be patched by a dynamic patcher. That is, the OH pass leaves expected hashes to be filled later by a patcher.

Since OH only is applicable in deterministic program regions, running a program with any given input should execute all its checks. Exploiting this characteristic of OH, we developed a debugger-based patcher that replaces placeholders with expected hashes according to computed values. We refer to this dynamic patcher as the `OH-Post-Patching` process.

3.5.4. Short Range Oblivious Hashing (SROH)

As discussed in Section 3.3.2, in order to realize SROH, we need to implement path discovery, distinct hash variables, hash verification and precomputation of expected values.

LLVM natively supports both dominance tree generation for functions as well as loop block detection. We exploited these features to compute individual short range paths for all the functions with control-flow dependency. The number of paths yields the number of required distinct hash variables.

SROH skips all DDIs in all paths. Within each short range path, we utilize our OH protection (see Section 3.5.3), with an enforced usage of the available hash variables in blocks. We also use LLVM's loop analysis along with a reachability analysis to detect loop-invariant instructions that can be incorporated into hashes within input-dependent loops.

It is important to ensure that the verifications of each path are strictly carried out in the blocks of that path only. The situation is slightly different for loops. Since input-independent loops incorporate loop-variants in their hashes, having verifications in the loop body entails supporting a set of expected values, instead of a single value. In our prototype, we solve this problem by verifying short range hashes of input-independent loops only in the loop end block.

Finally, we need to adjust verifications with the expected hash values. For this purpose, we use either of the LLVM interpreter (`lli`) or program slicing to emulate sliced instructions [163] in each path individually. Thereafter, we patch verifications with the correct expected hash values.

3.5.5. Self-checksumming (SC)

Since OH together with SROH is capable of protecting IIIs as well as DII|CFDIs, we desire to apply SC strictly to data-dependent segments of a program. This positively impacts the overhead of protections, without harming overall coverage.

The SC pass injects guards into the intermediate representation of a program according to the constructed protection network. These guards check if a function in memory hashes to an expected value. However, guard details (the beginning address of functions, their sizes, and expected hashes) can only be known after compilation. Therefore, in the pass, instead of the exact values we use a set of placeholders. Later, a post-compilation process (*SC-Post-Patching*) adjusts protected binaries with correct values of addresses, sizes, and expected hashes.

3.5.6. Response mechanism

Our response to any tampering is a covert termination by corrupting stack frames. This could be replaced with any desirable response mechanism by providing the desired behavior to the tool chain. The tool can easily be extended to support a set of diversified response mechanisms for better resilience.

3.6. Evaluation

In this section, we evaluate the efficiency as well as the effectiveness (coverage and security) of the proposed protection tool chain.

3.6.1. Dataset

To evaluate our protection, we use a subset of 26 programs in the MiBench dataset [93], which is a representative embedded benchmark suite comprising 33 programs. The reason for our restriction to the subset of 26 programs is crashes in the external library `dg` (`https://github.com/mchalupa/dg`) that we use to carry out tasks such as pointer and argument reachability analyses. We expect these issues to be fixed in the future. We also include 3 open source games gathered from Github, namely tetris, 2048, and snake.

Table 3.1 reports on various details of the programs in our dataset (LLVM Insts. column represents the number of instructions in LLVM). We employed our input dependency analyzer to compute the percentage of input-independent (III%), input-dependent (DDI%+ CFDI%), and finally data-independent (DII%) instructions. DII, in addition to III, also captures CFDI. III indicates the potential coverage of OH, while DII captures the same for SROH. However, besides non-hashable instructions, SROH inevitably has to skip loop-variants as well as instructions tainted by input-dependent arguments.

3.6.2. Preparation

To measure the impact of protection on a subset of program instructions, we repeat our experiments for random combinations of 10%, 25%, 50% and 100% of the program functions that we aim to protect. This mimics the user's specification of the sensitive functions to protect. For each program, we select twenty random function combinations of the aforementioned percentages and report the average values. We set the `SC connectivity level=1` constant throughout the experiment, i.e., there is one checker for each sensitive function. Since SC networks of checkers are randomly generated, we repeat the SC protection three times for the same combination of functions. Applying the protection for each program with different combinations and repetition of SC protection results in 240 variations of protections: 20 combinations of functions $\times$ 4 coverage percentages (10, 25, 50, 100) $\times$ 3 random SC networks. In the case of 100% coverage, we repeat the SC protection twenty instead of three times, because the 20 combinations of functions are all identical. All experiments were executed on a bare-metal machine with 16 GB of RAM and Intel Core i7 CPU running Linux 16.4.

3.6.3. Coverage

We are, firstly, interested in the coverage of each of the protection measures, i.e., how many instructions are protected by which measure. We report on the ratio of instructions protected by each of the OH and SROH primitives. We also measure the basic block coverage of these protections. Any block that contains at least one protected instruction is reported as a protected block. This is relative to the overall program, independent of whether we do 10, 25, 50, and 100 protections. According to our experiments, SC can protect all the instructions that OH/SROH fall short to protect. Table 3.1 reflects the following coverage values corresponding to our evaluation: the total number of basic blocks in programs (Blocks); the ratio of SROH protected blocks (SROHB%); the ratio of OH protected blocks (OHB%); the number of invariant-loop (unprotected) blocks (LB); the number and percentage of OH protected instructions (OH); the number and percentage of SROH protected instructions (SROH); the number of skipped instructions (SI) by SROH due to: incomplete (pointer) analysis, non-hashable instruction, loop-variant instruction,

Program	Instructions	III%	DDI+CFDI%	DI%	OHI	SROHI	OHI%	SROHI%	**OHI%+SROHI%**	SI	SROHDDI	Blocks	LB	OHB%	SROHB%	**OHB%+SROHB%**
qsort_small	92	2.2	97.8	17.4	2	2	2.2	2.2	4.3	12	0	17	5	5.9	11.8	17.7
qsort_large	147	1.4	98.6	10.9	2	6	1.4	4.1	5.4	12	4	19	5	5.3	26.3	31.6
crc	147	2	98	14.3	2	7	1.4	4.8	6.1	16	4	19	4	5.3	31.6	36.9
dijkstra_small	323	19.5	80.5	42.4	56	10	17.3	3.1	20.4	81	10	51	7	41.2	35.3	76.5
dijkstra_large	323	19.5	80.5	42.4	56	10	17.3	3.1	20.4	81	10	51	7	41.2	35.3	76.5
rawcaudio	418	0.5	99.5	12.9	1	28	0.2	6.7	6.9	47	22	63	5	1.6	44.4	46
rawdaudio	418	0.5	99.5	12.9	1	23	0.2	5.5	5.7	48	18	63	5	1.6	38.1	39.7
basicmath_small	532	49.4	50.6	51.7	176	0	33.1	0	33.1	99	0	62	12	67.7	0	67.7
tetris	629	20.7	79.3	44.7	33	65	5.2	10.3	15.6	207	24	108	19	12	53.7	65.7
basicmath_large	643	49.3	50.7	52.4	222	0	34.5	0	34.5	115	0	82	20	61	0	61
sha	657	0.3	99.7	23.4	2	58	0.3	8.8	9.1	100	6	56	8	3.6	67.9	71.5
bitcnts	664	0.5	99.5	9.9	3	11	0.5	1.7	2.1	60	8	69	9	1.4	15.9	17.3
fft	742	0.3	99.7	18.5	2	27	0.3	3.6	3.9	126	18	87	24	1.1	52.9	54
2048_game	749	0.9	99.1	35.4	4	115	0.5	15.4	15.9	180	34	124	12	0.8	72.6	73.4
search_large	827	10.2	89.8	26.5	55	0	6.7	0	6.7	164	0	147	6	10.9	2	12.9
search_small	827	10.2	89.8	26.5	55	0	6.7	0	6.7	164	0	147	6	10.9	2	12.9
snake	1065	2.8	97.2	23.3	14	92	1.3	8.6	10	178	36	148	19	5.4	73.6	79
patricia	1087	0.2	99.8	12.1	2	42	0.2	3.9	4	128	40	149	18	0.7	46.3	47
bf	3607	0.1	99.9	5	4	74	0.1	2.1	2.2	137	34	129	16	0.8	48.8	49.6
rijndael	5866	0.2	99.8	3.9	7	84	0.1	1.4	1.6	188	52	147	32	1.4	68	69.4
say	6859	0.3	99.7	18.6	12	591	0.2	8.6	8.8	1124	450	1026	159	0.5	67.1	67.6
susan	12656	0.1	99.9	5.6	12	588	0.1	4.6	4.7	602	498	698	104	0.1	77.2	77.3
ispell	13545	0	100	19.8	2	1580	0	11.7	11.7	2432	1328	2628	501	0	74.4	74.4
toast	13930	0	100	11.4	1	845	0	6.1	6.1	1334	594	1342	128	0.1	74.5	74.6
djpeg	52708	0.3	99.7	14	66	531	0.1	1	1.1	7168	384	5419	135	0.5	12.9	13.4
tiffdither	53162	0.1	99.9	13.7	8	1499	0	2.8	2.8	7028	1246	6855	170	0	30.1	30.1
tiff2bw	53463	0.1	99.9	13.7	5	1491	0	2.8	2.8	7046	1240	6882	187	0	29.8	29.8
cjpeg	54837	0.2	99.8	13.4	43	519	0.1	0.9	1	7177	366	5615	138	0.3	11.8	12.1
tiffmedian	55708	0.1	99.9	13.6	6	1619	0	2.9	2.9	7293	1340	7184	280	0	31.7	31.7
Mean μ	11608	6.6	93.4	21	29.4	342	4.5	4.4	8.8	1494.7	267.8	1358.2	70.4	9.7	39.2	48.9
Median	827	0.5	99.5	14.3	6	58	0.3	3.1	**6.1**	164	24	129	18	1.4	35.3	**49.6**
Std.Dev. σ	19744.1	13.2	13.2	13.6	51	531.8	9.2	3.8	8.6	2625.8	442	2376.6	108.8	18.1	25.1	23.5

Table 3.1.: Block and instruction coverage reports of OH and SROH protections. The III column describes the percentage of instructions effectively protected by OH; the DII column describes the percentage of instructions in principle protectable by SROH. Median increase in instruction coverage from OH to OH+SROH is by factor 6.1%/0.3%≈20; increase in median block coverage is by factor 49.6%/1.4%≈35.

argument reachable instruction; and the number of instructions in data-dependent branches protected by SROH (SROHDDI). Note DII=SROHI+SI-SROHDDI.

3.6.4. Performance analysis

Secondly, we are interested in the runtime and static overheads.

Runtime overhead

The `dg` library takes very long time (and crashes after a few runs) to complete its pointer analysis for tiff2bw, tiffdither and tiffmedian programs. This makes it impossible to generate protected versions for these programs. Also, ispell runs improperly as one of the provided input files (the dictionary) is apparently corrupted. Therefore, we excluded them from our performance analysis and also from Figure 3.2.

The protected binaries were each executed 100 times with the same inputs, in order to measure the runtime overhead and to weed out random factors. For programs with required input at runtime, e.g. games, we pipe the inputs by hooking system calls such as `getchar`. The baseline is computed by averaging 100 runs of the unprotected programs, which received identical execution parameters as their protected counterparts. Figure 3.2 illustrates the mean and standard deviation of the performance overhead of the protected programs in our experiment. One can see that some programs such as "crc" have a very low overhead (under 2%), while many others have a higher overhead (over between 32% and 64%). Moreover, the overhead generally increases with the coverage percentage of protected code. The irregularities are due to randomness in the implementation of the protection.

The form of the network of checkers plays an important role in the performance overhead of the protected binaries. That is, the overhead (of the protected binary) may vary, based on how hot the segments containing SC checkers are. For instance, if a function that is called frequently happens to carry out SC checks, this will cause a significant performance degradation. The large overhead variance on different instances (see the error bars in Figure 3.2) of the same program confirms the role of the checkers' locality.

Protection time

The protection process takes 30.16 seconds on average for our programs in the dataset. This time includes the input dependency analysis (12.12), SC (0.0019), SROH+OH (15.40), and the two post patching routines (2.61+0.29).

3.6.5. Security analysis

Thirdly, we are interested in the security of our scheme. One way to evaluate the security is to randomly tamper with protected programs and measure the detection rate. The assumption in such random attacks is that perpetrators possess no knowledge about the protection mechanism. However, we should assume that adversaries are fully aware of the internals of our protection scheme. To capture the attacker's possible actions, we hence decided to instead use the *attack-defense-tree* notation [117] as depicted in Figure 3.3 and refer an additional analysis with random tampering to future work. The ultimate goal of perpetrators is to tamper with sensitive code/data in a program, which is the root of our attack tree. They can do so if they manage to *disable OH guards which are guarding the*

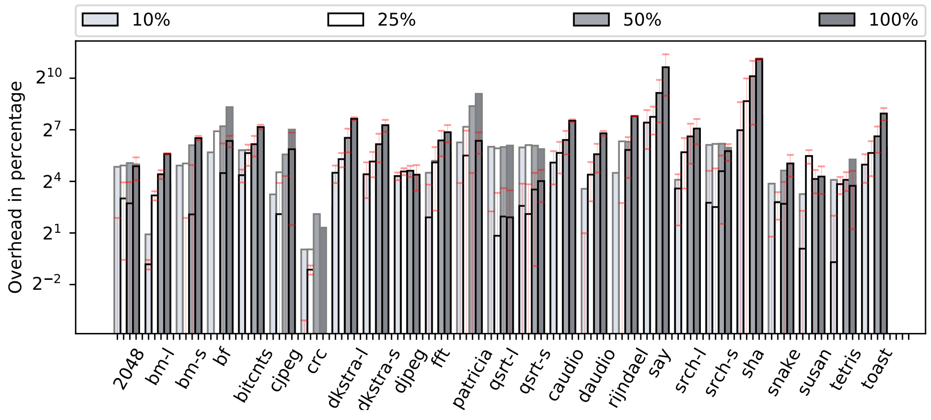


Figure 3.2.: Performance measurements using OH+SROH+SC for four (10%, 25%, 50%, and 100%) coverage levels. OH/SROH runtime overhead is depicted by black-margined bars, additional SC runtime overhead is shown with stacked bars with gray margins. Bars with no black margins indicate protected programs with negligible OH/SROH overheads.

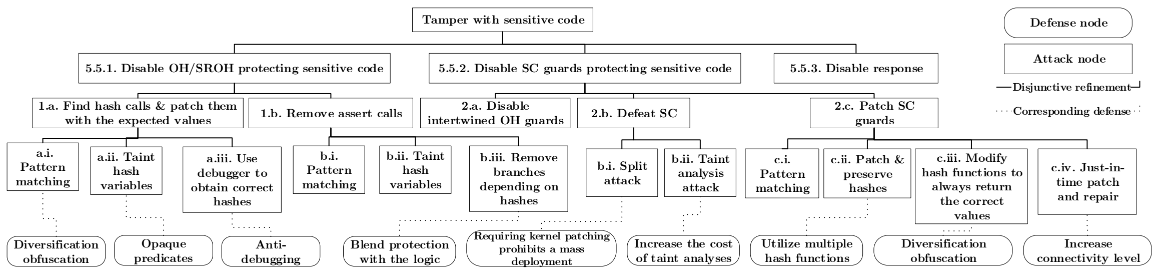

Tamper with sensitive code
5.5.1. Disable OH/SROH protecting sensitive code
5.5.2. Disable SC guards protecting sensitive code
5.5.3. Disable response
1.a. Find hash calls & patch them with the expected values
1.b. Remove assert calls
2.a. Disable intertwined OH guards
2.b. Defeat SC
2.c. Patch SC guards
a.i. Pattern matching
a.ii. Taint hash variables
a.iii. Use debugger to obtain correct hashes
b.i. Pattern matching
b.ii. Taint hash variables
b.iii. Remove branches depending on hashes
b.i. Split attack
b.ii. Taint analysis attack
c.i. Pattern matching
c.ii. Patch & preserve hashes
c.iii. Modify hash functions to always return the correct values
c.iv. Just-in-time patch and repair
Diversification obfuscation
Opaque predicates
Anti-debugging
Blend protection with the logic
Requiring kernel patching prohibits a mass deployment
Increase the cost of taint analyses
Utilize multiple hash functions
Diversification obfuscation
Increase connectivity level
Defense node
Attack node
Disjunctive refinement
Corresponding defense

Figure 3.3.: An attack-tree-based security analysis of the proposed protection scheme.

sensitive code, *disable SC guards* and/or *disable the response mechanism*.

Disable SROH/OH guards protecting sensitive code

To defeat OH protection attackers need to *1.a. find hash calls & patch them with expected values* or *1.b. remove all asserts* which verify the computed hashes. To replace calls to hash functions with expected values they first need to find these hash calls in the binary. *a.i. Pattern matching* and *a.ii. Tainting hash variables* are two techniques that an attacker may utilize. If hash calls in a protected binary follow recognizable patterns, attackers can quickly identify hash call sites. To raise the bar against such attacks, we suggest to resort to diversification obfuscation techniques, e.g. instruction(s) substitution [56]. Attackers can utilize taint analyzers [150] to backtrack involved instructions in hash computations. This, however, requires attackers to identify hash variables in the binary first, which is obviously harder.

One way to raise the bar against taint analysis is to extend the search space, for instance, by introducing a massive number of bogus data dependencies on hash variables using opaque predicates [62].

Another possibility is to *a.iii. attach a debugger* (on an untampered version of the program) and record expected values on each hash call site. Then, these expected values will be patched in the binary. However, attackers would need to identify all OH hash call sites. Therefore, such call sites must utilize strong obfuscations to prevent pattern matching. Another possibility is to add resilience against debuggers (dynamic analyzers). For instance, the anti debugger technique proposed by Abrath et al. [3] can add more resilience.

1.b. Removing assert calls is another possibility for attackers to potentially circumvent the protection. Again, they can utilize *b.i. pattern matching* and/or *b.ii. taint analysis* to achieve this goal, for which the aforementioned hardening measures apply. Another attack is to *b.iii. remove all the branches that rely on computed hashes*. For additional resilience, we can further blend the program logic with the protection, e.g., the execution flow can depend on the computed hash values, similar to the technique presented in [47].

Disable SC guards

Adversaries may opt to defeat SC guards if the sensitive code of interest resides in an SC-protected segment. Three sub-attacks can materialize here - *2.a. Disabling intertwined OH guards*, *2.b. Defeating SC* and *2.c. Patching SC guards*.

2.a. Disabling intertwined OH guards. As all the SC guards are additionally protected by OH/SROH protection. Disabling the intertwined protection boils down to attacking OH/SROH guards (see Section 3.6.5).

2.b. Defeating SC. There are two known attacks on self-checksumming based techniques: *b.i. memory split* [183] and *b.ii. taint analysis* [150]. The former requires a modification of the OS kernel, which potentially prohibits the massive deployment of tampered programs. However, the latter can potentially defeat our entire protection if SC hashes taint our OH/SROH hashes. The reason is that the taint-based attack can in theory identify all the branching conditions that are tainted by the parts of the process memory that contains the program's code. This not only includes SC guards but also OH/SROH guards. To mitigate this attack, we can ensure that SC's computed hashes (over process memory) are never incorporated into OH/SROH hashes, and instead OH/SROH covers the routine that computes such hashes. Also, we must place OH/SROH guards spatially distant from SC guards, otherwise a scan near detected SC guards may reveal the overlapping OH/SROH guard(s). This adds resilience against tampering attacks targeting SC guard arguments, hash and response routines. However, even with these measures, the taint-based attack can still detect SC's branching condition. Given that the taint tracking process of the attack is computationally very expensive [22], carefully crafting programs to increase the cost of such taint analyses could potentially render the attack impractical. For instance, one could introduce a large number of (side-effect-free) read/write operations from/to SC variables.

2.c. Patch SC guards. In order to disable SC guards attackers could use *c.i. pattern matching* (already addressed in *1.a.i*), *c.ii. patch and preserve hashes*, *c.iii. modify hash functions to always return the expected values* and *c.iv. just in-time patch & repair* attacks.

c.ii. Patch and preserve hashes. Since the hash function that is used by SC is not collision-free, it is possible that attackers manage to patch some instructions in a way that hashes remain unchanged. To mitigate this threat, we can utilize a multitude of overlapping hash functions similar to those proposed in [22]. Another option on the x86 architecture is to interleave program blocks such that tampering with protected blocks affects other blocks [99].

c.iii. Modify hash functions to always return the expected values. Attackers can make hash functions to always return the correct value, if they manage to relate a guard's branching condition (containing an expected hash) to the underlying hash function that computes hashes. To tackle this attack, we shall utilize diversification obfuscation on instructions that evaluate hash results [25, 56].

c.iv. Just-in-time patch and repair. Another possibility for attackers is to patch a protected segment and fix it before it is checked. Increasing the connectivity level raises the bar against this attack.

Disable response

Finally, attackers can target the response mechanism itself, for instance, by using pattern matching or taint analysis. For example, an immediate termination after tamper detection could be traced back to the protection guards. For this purpose, covert responses like those discussed in [47, 58, 64] could be utilized.

It is worthwhile to note that the resilience of our protection guards (both SC and OH/SROH) as well as their response routines heavily relies on obfuscation. Therefore, we believe further empirical studies need to be conducted to measure the effectiveness of the applied obfuscation techniques on our protection scheme.

3.7. Discussion

3.7.1. Coverage

The potential coverage of OH is limited by the number of instructions that deterministically process—constant—data for all program inputs (III). Our programs contain a median of only 0.5% IIIs (μ=6.6%, σ =13.1%), which severely limits the application of OH. In fact, we observed a median instruction coverage of only 0.3% (μ=4.5%, σ=9.2%) for OH. Given that instructions with no data dependency on nondeterministic data constitute a median of 14.3% (μ=21%, σ =13.6%) (see DII% column in Table 3.1) in our dataset, SROH has the potential to improve coverage of protected instructions. This is, however, limited by factors such as hashable instructions, incomplete analysis results and argument reachable variables. Our SROH added a median instruction coverage of 3.1% (μ=4.4%, σ=3.8%). *This is an increase w.r.t. pure OH protection by a median factor of 20.* It is worth noting that our prototypical input dependency analyzer relies on conservative pointer as well as reachability analyses provided by the `dg` tool. More advanced pointer analysis techniques (such as those presented in [168]) can potentially enhance the coverage. Despite the immediate benefits, we see the extension of the analysis tool beyond the scope of this work, given that all the presented protection techniques remain unchanged.

Regarding block coverage, SROH yields even more encouraging results. Our experiments show that SROH yields a median additional block coverage of 35.3%. When contrasted with the median block coverage of 1.4% for OH alone, *this corresponds to an increase of the number of protected block by a factor of 35.*

Of course, the external validity of our results is limited by the programs in our dataset.

3.7.2. Implicit protection with OH/SROH

Apart from the protection coverage that SROH offers, it has another major benefit on the security of the system. SROH enables us to cover all the SC guards regardless of the branches in which they reside. This coverage makes it harder to tamper with such guards and thus turns our scheme into a more resilient measure (see Section 3.6.5). Whether taint-based attacks can help identify, and subsequently disable, OH/SROH guards for SC guards, depends on the efficiency of this attack in practice (see Section 3.6.5 and [22]). Our experiments confirm that all the SC-protected segments benefit from the implicit protection of SROH/OH.

3.7.3. Performance

For the bf, bm (both large and small), cjpeg, patricia, qsrt (both large and small) programs, a significant portion of the overhead stems from SC protection, as shown in Figure 3.2. This can be mitigated if resilience of the protection is less important than performance, by simply not applying SC to those parts of the code that are deemed less critical. In case of programs like dijkstra (large and small), djpeg, rijndael, say, search (large and small) and sha, we observe a higher overhead being imposed by OH/SROH. We believe that SROH hash verifications (asserts) in the loop body of such programs are the main reason behind the slowdowns. Employing the composed protection (SC+OH/SROH) on CPU-intensive programs (particularly with a high number of instructions located in loops) inevitably imposes high overheads. Therefore, to better analyze overheads, we shall make a distinction between computationally expensive and computationally cheap programs. It is fair to mention that most of (if not all) MiBench programs are rather computationally expensive and hence our results tend to capture a worst case scenario.

We consider bf, dijkstra (both), patricia, rawaudio (both), basicmath (both), search (both), bitcnts, fft, rijndael, say, sha, and toast as computationally expensive programs, and consider the rest as computationally less expensive. For the computationally less expensive programs, our experiments indicate acceptable overheads (median of 17.6% (μ=25%, σ=23%) and 29.7% (μ=35%, σ=21%)) for a partial protection of 10% and 25%, respectively. The 50% protection induces higher slowdowns of 37.1% (μ=39%, σ=22%). A full protection imposes an overhead of 52% (μ=62%, σ=49%). Note that the large variance is due to the nature of the protected programs.

Although the application of resilient protection on CPU-intensive programs yields a high overhead, we believe these overheads are acceptable for non-CPU-intensive programs. For instance, 10, 25, 50, and 100 protection coverages impose no noticeable slowdowns in the games in our dataset (namely tetris, 2048, snake). Therefore, whether the overhead is acceptable or not depends on the specific program at hand. For instance, commonly checked program parts include license checks which usually are not CPU intensive.

3.8. Conclusions

We have designed, implemented, and evaluated an end-to-end tool chain for practical integrity protection with oblivious hashing and self-checksumming. In order to protect input-data dependent instructions, we proposed SROH as a technique for protecting input dependent control flows by localizing hash computations.

We evaluated our proposed protection with 29 real-world programs, 26 of which are taken from the MiBench dataset, plus 3 open source games. In this sample set of programs, increased the median of protected instructions by a factor of 20 and the median of protected blocks by a factor of 35 (SROH+OH when compared to OH only). The remaining instructions were protected by a complementary self-checksumming technique hardened by a network of checkers. We then showed how to interleave both protection mechanisms, aiming for a more resilient protection.

Our results indicate that the imposed overhead is 52% with a complete protection (for non-CPU-intensive programs). However, in real use cases it is often necessary to protect only a subset of the program (i.e. sensitive functions). This, we argue, makes our protection a practical scheme.

4. Virtualized Self-Checksumming

In this chapter, we present a novel approach to address the problems of self-checksumming integrity protection concerning applicability and composability. This chapter was previously published in a work [7] co-authored by the thesis author.

4.1. Introduction

Self-checksumming (SC) enables programs to detect changes in their process memory using so called *guards* [47]. Upon detections, SC calls a response mechanism. However, since these checks have to be executed at runtime, the expected checksum values need to be pre-computed and, after compilation, inserted into the executable. Not only does this approach require knowledge of the underlying system's architecture, it also mandates a post-patching step to put these expected checksums (hashes) into predefined places. This is an extremely tedious and error-prone process [22]. This process also limits the use of other obfuscation techniques, as one needs to maintain a set of placeholders (i.e. contiguous sequences of 4 or 8 bytes in the code segment) for the pre-computed checksums at known offsets in the executable. Applying obfuscation would likely change the offsets and contiguous layout of the placeholders.

To eliminate this post-patching step, we leverage the compiler framework LLVM [119] to implement self-checksumming atop virtualized instructions. For this, we also implement virtualization obfuscation. LLVM already implements backends for different system architectures, which removes the required architecture knowledge and post-patching. Applying the guards at a higher abstraction level removes the post-patching process at binary level entirely. This is achieved by first applying virtualization obfuscation [87] and then adding the guards in the custom interpreter bytecode.

Contributions. This work makes the following contributions:

- A novel design for combining self-checksumming and virtualization obfuscation (Section 4.3).
- A performance evaluation of the implementation, using a dataset of 24 real-world programs (Section 4.4).
- An attack-tree based security analysis of the design and implementation (Section 4.4).

The rest of this chapter is organized as follows. Section 4.2 presents the necessary background knowledge for this chapter and related work. Section 4.5 discusses performance and security tradeoffs. Section 4.6 presents the limitations of this approach and implementation. Finally, the conclusions are presented in Section 4.7.

4.2. Background and Related Work

In this section we first present the two essential parts of our approach, i.e. *virtualization obfuscation* and *self-checksumming*. Afterwards, we present related work about combining the two techniques.

4.2.1. Virtualization Obfuscation (VO)

VO's primary goal is to transform a program's control flow to a semantically equivalent, yet less comprehensible version. Given a program, this technique lifts all instructions to a new, random *Instruction Set Architecture* (ISA). Thereafter, an interpreter specific to the newly generated ISA is created. Lifted code along with the interpreter are what the end user receives.

The interpreter's job is to fetch, decode and dispatch execution to the original instructions' *handler*. Each handler emulates the original instructions' behavior. Furthermore, all program memory allocations are done via a *virtual memory* (VM). A *virtual program counter* (VPC) keeps track of the last executed instruction at runtime. This obfuscation technique can be applied at different representations of programs such as source code [57] or binary level [126]. The level at which the technique is applied plays an important role in its composability with other protections. To the best of our knowledge, there is no publicly available implementation of VO as a compiler pass particularly in LLVM.

VO intuitively reduces the comprehensibility of protected programs for attackers. VO can resists automated attacks imposing additional cost [12, 21, 44] at the perpetrator's end. For instance, the utilization of opaque predicates [62] or range dividers [20] can significantly hamper attacks based on symbolic execution [156].

4.2.2. Self-Checksumming (SC)

Self-checksumming is a software tamper-proofing technique [47]. The idea is to equip a software with a set of interconnected guards. Each of these guards carries out hash calculations over the code segment (in the process memory) during runtime to detect potential code manipulations. Guards need to be pre-seeded with the expected hash values of the code that they are protecting. Upon detection of code tampering (i.e. hash mismatch), a response mechanism is triggered [139].

Since all calculations are done at runtime over the binary (machine code), the expected hashes can only be known after compilation of programs. That is, if an SC protection is to be developed as a compiler pass, the expected hashes need to be adjusted once the binary representation of the program is finalized. For this purpose usually placeholders along with post-patching mechanisms are utilized [4]. Alternatively, a backend pass (similar to the one described in [104]) can be used to adjust placeholders.

Both of the mentioned adjustment approaches have a tight dependency on the underlying architecture. That is, the adjustment shall be tailored for each and every architecture for which the binary is compiled. The architectural dependency imposes extra cost in terms of development and maintenance of protections.

Furthermore, SC guards are susceptible to pattern matching attacks [4, 9, 22]. Therefore, without proper utilization of obfuscations, SC is rather easily identifiable, and perhaps defeatable. However, obfuscation inherently alters the syntactical representation of programs. SC imposes two limitations on the application of obfuscations: **i)** the placeholders need to be preserved (not obfuscated); otherwise, some adjustments may fail; and **(ii)** applying further obfuscations (after SC hashes are adjusted) could break SC guards (as expected hashes may no longer hold). These setbacks may result in having no obfuscation on overlapping guards and expected hash values (placeholders before adjustments), which negatively impacts the resilience of SC. Consequently, the composability of SC with other protections is heavily limited, and thus, the overall security. In this chapter, we propose a technique to overcome the identified drawbacks and to fix the composability problem of SC.

4.2.3. Combining Virtualization and Integrity Protection

Since our proposal is based on combining VO and SC, we reviewed existing literature on protection schemes that use VO to add resilience to their schemes.

Ghosh et al. [87] proposed a protection technique that combines process virtualization (comparable to VO), encryption, and self-checksumming. In their technique, program instructions together with their protection guards are encrypted and subsequently shipped into the binary. The decryption key is also placed in the binary but protected using white-box cryptography [53]. The process virtualization is rather used as a proxy to decrypt protected instructions. Since the SC protection is directly applied on the instructions, utilizing further protections (after the application of SC) could break the SC protection. In another similar work Ghosh et al. [88] used SC to protect dynamically generated (cached) instructions of the virtualized programs.

Gan et al. [82] proposed a technique that uses protected virtual machines acting as the root of trust. Several hardening techniques including white-box cryptography, obfuscations, etc. are also utilized to protect the virtual machine. The authors indicate such a virtual machine can be relied on to carry out integrity checks.

In contrast to the related work, our proposal rather aims at improving composability of protections and removing architectural dependencies and thus reducing costs.

4.3. Design

In this section we present the architecture of our approach, the design decisions and the reasoning behind them. We end this section describing the optimizations that were added to the design.

4.3.1. High-level Architecture

The architecture of a protected program using our technique is presented in Figure 4.1. Every program utilizes its own *Random (Virtual) Instruction Set Architecture* (RISA). The RISA (depicted in the top-left corner of Figure 4.1), is a volatile data structure capturing a set of random instruction mnemonics and their actual semantics. The RISA only exists at the time of transformation and it is discarded afterwards. The RISA is created sequentially while visiting program instructions.

Our technique can be applied on either the entire application or a subset of its functions. Users can annotate the desired functions with a *sensitive* tag. By iterating over instructions (of annotated functions) a `Virtual Program Array` (VPA) containing the translation of the original instruction to the new RISA is created (top-right of Figure 4.1). That is, if an instruction does not have an equivalent mnemonic in RISA, a corresponding mnemonic and a reference to its semantics are inserted into RISA. Next, all memory accesses (constants, variables and registers) are altered to use a sequential `Virtual Memory` (VM), which is depicted in the middle-left side of Figure 4.1. In addition to mnemonics, which are captured as random opcodes, the VPA keeps a reference to the indexes at which corresponding data of instructions are persisted in the VM.

At this stage, self-checksumming guards according to the specified protection requirements are injected into the VPA (center of Figure 4.1). Guards, similar to the original program instructions, use the VM to carry out their calculations and subsequently verifications. Simply put, the computed hash as well as the expected hash values are stored in and retrieved from the VM. It is important to note that SC guards protect the VPA; i.e. they compute hashes over the VPA (not the original program instructions).

For the lifted instructions to be executable an `Interpreter` is needed (bottom of Figure 4.1). The Interpreter fetches instructions from the VPA. It decodes the RISA's semantic of mnemonics to execute the fetched instructions. Within the interpreter a `Virtual Program Counter` (VPC) keeps track of the last executed instruction (see middle-right part of Figure 4.1).

4.3.2. Detailed Design

This section presents several architectural components from Section 4.3.1, and their inner workings in more detail.

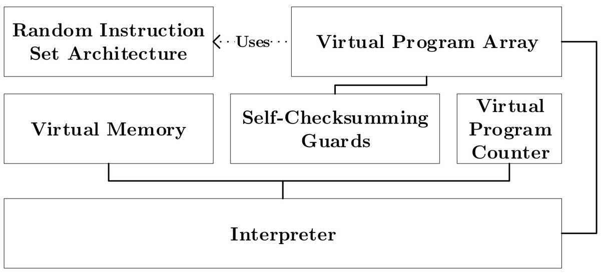


Figure 4.1.: Solution's high level architecture

Random Instruction Set Architecture (RISA)

The RISA is a program/function specific contract comprised of *Opcode*, *Operand(s)*, and *Semantics*. The opcode, in essence, captures mnemonics of the RISA. Opcodes are randomly generated 16-bit integers. Operands refer to indexes in the VM where either the input or output of the computation should be "retrieved from" or "stored to".

Virtual Memory (VM)

The VM holds all operands needed by the translated instructions. Values are stored contiguously, and accessed by index. As types are known beforehand, we can deterministically compute the amount of elements we need to read or write upon load and store operations, respectively. To do so, we simply set a pointer to the index corresponding to the "beginning of" and "operand of" interest followed with a cast to the known type. After dereferencing this pointer, the value can be used by the interpreter. Persisting changes to VM values includes splitting the data into chunks of 8-bit integer values.

Virtual Program Array (VPA)

The VPA is a static array containing the lifted instructions. As we are using an interpreter per-function (see section 4.3.2), each function has its own VPA and its checksum can be verified by any other function in the same module. In the case of a control-flow changing instruction the next VPC value is provided in the bytecode, which the interpreter will assign to the VPC. Since we are using 16-bit integers as *Opcode*, all elements in VPA are 16-bit integers. This includes indexes to the VM. The VPA is a constant, global array. This allows any function to calculate the hash of any other virtualized function. Being declared as constant in LLVM also means we cannot generate code that modifies the array at runtime.

The decoded instruction's result will be written back into the VM to be used by subsequent instructions. The VM is allocated on the heap. When handling an instruction that returns execution to the caller, we clean up the allocated memory before returning (also see Section 4.6.2).

Instruction Lifting and Translation

In our design, we generate an interpreter for each virtualized function. Function level virtualization enables us to maintain the program's original function symbols and thus no linking problem will be incurred. As our solution is implemented as a transformation pass based on LLVM, the maximum number of instructions we have to translate in our VO is limited by the maximum of instructions in LLVM, i.e. 58 distinct instructions. `PHINode` is the only instruction that we cannot directly translate. To cope with `PHINode` instructions, we resort to LLVM's `reg2mem` pass yielding replacement of `PHINodes` with corresponding allocations, load and store instructions. Exceptions are not supported in the current implementation (see Section 4.6.1).

Based on the user's list of sensitive functions, we sequentially virtualize each one individually. The virtualization step first iterates over all instructions in the current function. For each instruction visited, we generate a random 16-bit integer value to be used as the *opcode*. If an instruction of this type has already been added, (granted that operands' types match) we reuse that instruction's *opcode*. Next, for each operand, we add the value to VM and append the index to the VPA. For instructions that change the control flow, such as branch instructions, we append a placeholder instead.

After all instructions have been visited, we iterate over the translated instructions and, for instructions that change the control flow, replace the placeholders with correct indices of the branch destinations. Despite the flat nature of the VPA, we can still translate branch instructions. Given that LLVM branch instructions point to target basic blocks, we need to translate the target to an index in the VPA. The index should refer to the first instruction of the original destination block in the VPA. This is as simple as assigning the target instruction index to the VPC. Such assignments mimic jumps in programs.

For conditional branches, the situation is slightly different. Each conditional branch has two destinations depending on whether the condition holds true or false. Therefore, we need to translate both target blocks to their corresponding indexes in the VPA and subsequently set the VPC accordingly. The branch translation step needs to be done after all instructions have been virtualized to guarantee that the index is correct. Otherwise, we might not find the correct instruction, if it has not yet been virtualized.

After all instructions have been virtualized, we add the current function with its hash to a temporary list. This list is later used while crafting SC checkers (see Section 4.3.2). If the current function is a checker (dictated by the randomly created network of checkers Section 4.3.2), we insert a guard for each checkee at random locations into the VPA of the function.

Interpreter

An interpreter is generated for each individual function. Interpreter generation starts with allocating a VM and storing all needed constant values and function arguments into it. The VPC is initialized to zero. Next, a loop is added, which fetches the next instruction, decodes it and dispatches execution to the corresponding handler. The interpreter starts with the instruction at index zero, and then increments the VPC by one for each fetched *opcode* from the VPA. In case of control-flow changing instructions, the VPC is set to indexes of target destinations in the VPA.

The main body of the interpreter is in fact a switch statement that takes the current *opcode* as an argument. Each case in the switch body corresponds to a distinct opcode in the RISA. Cases are *handlers* of virtualized instructions. Handlers contain code that emulates the original instruction(s). This involves loading operands from the VM, executing the decoded instruction, and writing the result back into the VM.

Several instructions coded with the same opcode in the program will end up using the same handler. In addition to reducing the binary size, reusing these handlers has several advantages. For one, there is no one-to-one mapping of handler and original instruction. This also implies that if an attacker wants to change how one specific instruction is handled, for example changing a jump destination, it will result in side-effects at different locations of the function. Therefore, it is intuitively harder to tamper with the program control-flow or, generally, instructions.

Using function-level virtualization has the advantage that once a function's interpreter has been successfully reverse engineered, the attacker cannot transfer this knowledge to the other functions; instead, each function's interpreter has to be reverse engineered individually. Moreover, having virtualization at the level of functions yields a simpler linking process.

Since the `ret` (return instruction in LLVM IR) can be emulated in the function-level virtualization, passing the return values between functions becomes trivial. However, a module-level interpreter would need to read and write the return values from a shared memory, which entails addressing challenges such as *race conditions* and *memory management* issues.

In our design, the default case of the interpreter's switch statement will invoke the response mechanism. That is we treat invalid opcodes as well as VPC issues as tampering attacks.

Self-Checksumming Guards

One way to tamper with the program behavior is to manipulate VPA values. Our self-checksumming protection utilizes a set of code snippets, referred to as *guards*, that verifies the VPA values of different (virtualized) functions. In a nutshell, a guard hashes a function's VPA to ensure that it matches the expected value at different intervals during program execution.

SC protection is added after virtualization. Guards obtain a unique opcode similar to other instructions. Each guard has a target function (checkee) as well as an expected hash value. That is, for every guard opcode in VPA there exists two operands in VM. Another memory slot is reserved in VM for the runtime hash value of guards. During program execution, upon fetching a guard opcode the interpreter loads the address of the target function's (checkee's) VPA along with the expected hash from VM. For the given target VPA a cumulative byte-by-byte hash is computed that is persisted in the preserved runtime hash in VM. The runtime hash is subsequently matched with the expected hash. In case of mismatches the response mechanism is triggered. Bear in mind that depending on the desired protection configuration (see Section 4.3.2) more than one guard might be injected into the VPA of a given function.

As per the hash function, we use the binary `XOR` operator which comes with two clear benefits. First, the operation occurs quite often in normal programs, e.g. to clear a register value. So, it offers a higher stealth. Second, `XOR` is fast as it requires only a single operation on many X86 processors [81]. Nonetheless, our implementation can easily be extended to use different hash functions, if needed.

Guards Connectivity

If a guard in function A checks the code of function B, then we call function A the *checker* and function B the *checkee*. Connectivity refers to the number of checkers per each sensitive function. In the generated network, this is given by the number of incoming edges to a (sensitive) node. Since in our implementation we do not support cyclic checks, the connectivity is restricted by the number of functions in the module.

Network of Checkers

We generate a random network of checkers in the form of a directed acyclic graph. For each function in the sensitive set, we pick a number of functions equal to the desired connectivity to be checkers to the function of interest. In cases where the desired connectivity value is simply not achievable, for example because it is higher than the number of functions in the module, our pass will use the highest possible connectivity. This results in each sensitive function being checked by all other non-sensitive functions in the module.

Response Mechanism

Hash mismatches in our solution are redirected to the default case in the interpreter's switch statement. The response mechanism in our solution is straight-forward termination of the process by calling the C library function `abort`. This can be extended to use a stealthier response (such as stack pollution) or a multitude of them, one of which is randomly selected at runtime. Since the intrusiveness and hostility of the response mechanism depend on the use case (e.g. termination is not an options in browsers [19]), we let the users of the tool develop their desired responses.

4.3.3. Optimization

The machine code that will be generated by our scheme is likely to be quite inefficient. The switch statement can, in the worst case (with no reuse of handlers), get as large as the number of instructions of the original function. This results in a multitude of comparisons and conditional branches. In some cases, the value that is used in the switch has to be compared against each case value until a match is found. Therefore, the generated code, in the worst case, has to iterate through plenty of cases and compare them to the given opcode value. Only on a match, will it jump to the specified block. Thereafter the same routine is repeated until no more instructions are left in the function's VPA.

It is noteworthy to mention that some compilers reduce the number of comparisons by translating switch statements to indirect jumps through jump tables [29]. This optimization can improve the performance at the cost of adding a multitude of jumps in the program binary.

Despite handler reuses and indirect jumps, the performance impact particularly for instructions within loops can be significantly high. To cope with this limitation, we extend our interpreter to support indirect threading [24]. This optimization enables directly connecting the handlers with each other, instead of having to iterate through the switch cases. The cornerstone of this technique is to compute the successor handler for each handler. Afterward, a direct jump to the successor is placed at the end of handlers. If there exists more than one successor for a given handler, we compute the list of possible targets and insert an indirect branch. Listing 4.1 illustrates (using pseudo-code) how the control-flow looks like after applying this optimization. Note the `goto` instructions at the end of each handler.

In LLVM, indirect branches require a list of possible destinations and an address to jump to. While we can statically determine the list of destinations, we only know the correct addresses during execution. Not only are the actual addresses not calculated at the point of our pass, we are also using the handler for more than one instruction, so we also have more than one successor and need to account for multiple addresses. Since each instruction by itself only has one successor, we can simply use an index into the list of possible destinations and resolve the correct destination address at runtime. This is shown in listing 4.1 on lines 8 and 9.

Despite the potential performance improvements, we believe the optimization negatively impacts the security of the protection. Connecting the handlers with each other also makes the resulting control flow more linear. Linear control flows makes the resulting machine code easier to analyze both for a human and static analysis tools. As a switch in machine code is nothing more than a sequence of comparisons and conditional jumps, symbolic execution tools quickly run into path explosion problems. Linear control flow for a human makes the sequence of executed instructions more intuitive than a series of comparisons

Listing 4.1 A simple demonstration of an interpreter utilizing the indirect threading optimization

```
int some_func() {
  uint8_t *DataArray = ...; // store constants and function params
  uint64_t PC = 0; // program counter
  goto handler_1;
  handler_1: // this handler is being reused
    handle_op_1();
    PC += ...;
    next = CodeArray[PC]; // dynamically determine next handler
    goto handlers[next];
  handler_2:
    handle_op_2();
    PC += ...;
    goto handler_3;
    ...
}
```

and conditional jumps. Furthermore, the handlers without reuse can readily be spotted by attackers.

Therefore, we keep this optimization optional, to be used in cases where large overheads are not tolerable. In the remainder of this chapter we refer to the unoptimized implementation as the *secure* implementation.

4.4. Evaluation

In this section we conduct a set of experiments to measure the performance and to evaluate the security of our proposed scheme. As pointed out earlier, our optimization is set to enhance the performance. However, the actual performance improvement as well as security implications of the *optimized* approach are rather unknown. Throughout this section we run our evaluations on both approaches to precisely capture advantages and disadvantages of each approach.

4.4.1. Dataset

To carry out our evaluations we use a subset of 22 programs from the MiBench suite [93] along with three open source CLI games, namely `tetris`[1], `snake`[2], and `2048`[3], summing up to a total of 25 programs. It is worthwhile to mention that technical difficulties in compiling some MiBench programs to a single LLVM IR bitcode file (e.g. use of a mix of assembly and c code) forced us to exclude them from our evaluations. The first four columns in Table 4.1 show details (#Instruction: number of instructions, #Function: number of functions, and #Block: number of basic blocks in LLVM IR) for each program in our dataset.

4.4.2. Coverage

The goal of this experiment is to determine the effectiveness of the protection scheme. One way is to measure the protection coverage. Coverage of protection in this case refers to the number of instructions that are virtualized as well as the number of instructions (in the VPA) that are protected by SC guards.

We generated a set of protected binaries (VO+SC) with a function coverage of 100%. A coverage of 100% indicates that the protection deems all the program instructions as security sensitive and thus tries to protect them all. Since the checkers of SC guards are randomly selected, we repeat the binary generation 20 times to weed out potential noises yielding a total of 480 programs, 20 $\times$ 24 (programs). Bear in mind that utilizing optimizations has no impact on the coverage of SC. Columns #Prot. Inst. and Prot. Inst.% of Table 4.1 capture the number and percentage of SC protected instructions, respectively. Section 4.5.1 discusses the impact of the protection coverage on the scheme security.

4.4.3. Performance

The goal of this experiment is to measure the overhead of the added protection in both secure and optimized modes. We intend to capture the actual overhead of VO in both modes. Then, we repeat the same set of experiments for the combination between VO and SC. This way of measurement enables us to single out the overhead of each scheme separately.

To measure the impact of our protection, we generate protected binaries with a range of partial protections (i.e. 10%, 20%, and 50% of functions) as well as a complete protection (i.e. 100% of functions). For every protection level we generate 20 random combinations of functions to be deemed as sensitive. Thereafter, if the SC protection is enabled, we generate 10 protected programs for each combination to weed out the noise of SC's random network of checker creation. That is, for the VO+SC benchmarks we end up with 800 (20 [function

[1]`https://github.com/troglobit/tetris`
[2]`https://github.com/troglobit/snake`
[3]`https://github.com/cuadue/2048_game`

Program	#Inst.	#Function	#Block	#Prot. Inst.	Prot. Inst. %
qsort_s	107	2	18	74	69
crc	152	4	20	135	89
qsort_l	166	2	22	128	77
dijkstra_l	338	6	59	304	90
dijkstra_s	338	6	59	304	90
rawcaudio	437	3	78	389	89
rawdaudio	437	3	78	389	89
basicmath_s	538	5	63	186	35
basicmath_l	649	5	83	186	29
sha	666	8	57	619	93
tetris	669	13	129	564	84
bitcnts	705	15	82	669	95
fft	760	7	91	483	64
2048	803	17	146	699	87
search_l	873	10	159	741	85
search_s	873	10	159	741	85
snake	1124	13	172	1071	95
patricia	1201	6	169	867	72
bf	3667	8	168	3383	92
rijndael	5924	7	147	5074	86
say	7447	75	1302	7135	96
susan	12996	19	916	12760	98
toast	15374	94	1542	15268	99
djpeg	54496	379	6518	53204	98
cjpeg	56735	391	6788	55396	98
Mean	6699.00	44.32	761.00	6430.76	83
Median	760.00	8.00	129.00	669.00	**89**
Std	15253.74	104.82	1817.06	14937.38	18

Table 4.1.: Protection coverage of instructions including the network of checkers using SC atop VO (*Prot. Inst.*% column)

combinations] $\times$ 4 [protection levels] $\times$ 10 [network of checkers]) protected instances for each mode (secure and optimized). However, when only VO is applied we generate 80 (20 $\times$ 4) protected instances per mode. It is noteworthy that we set `connectivity=2` for

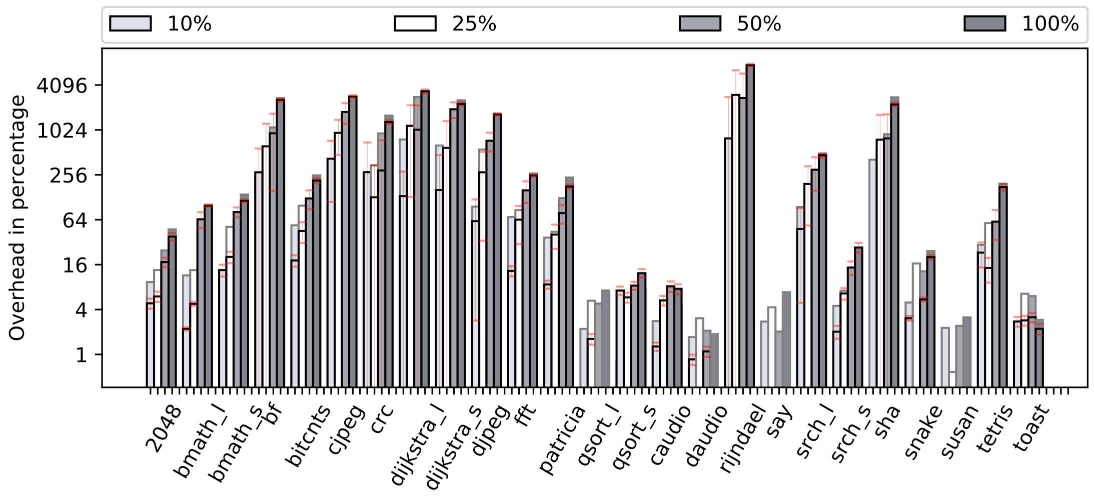


Figure 4.2.: Overhead of VO vs. VO+SC protection in the secure mode; the gray-margined bars capture VO+SC protection while the black-margined bars depict VO protection results

SC throughout our experiments. That is, every sensitive functions is checked by 2 other functions, if enough functions are available in the module.

We use `benchexec`[4] [30] to precisely measure the performance of our programs before and after applying protections. We run each program 100 times with the exact same inputs and measure the overhead. For games we pipe in constant input by intercepting the necessary library calls (e.g. `getch`). According to our experiments all the protected programs execute correctly with respect to the provided inputs.

Secure Mode

Figure 4.2 illustrates the overhead of VO as well as VO+SC in the secure mode . The average overhead of VO (applied single handed) is 89.34%, 314.95%, 445.41%, and 1018.46% for protection levels of 10, 25, 50 and 100, respectively. When VO and SC are combined the average overhead rounds to 109.74%, 193.64%, 468.72%, and 1055.51% for protection levels of 10, 25, 50 and 100, respectively. Note that the average of VO+SC overheads for the 25% protection level are smaller than the corresponding overheads when only VO is applied. We believe this is due to the randomness of function selections, the frequency of check executions, and potential LLVM optimizations.

[4] `https://github.com/sosy-lab/benchexec`

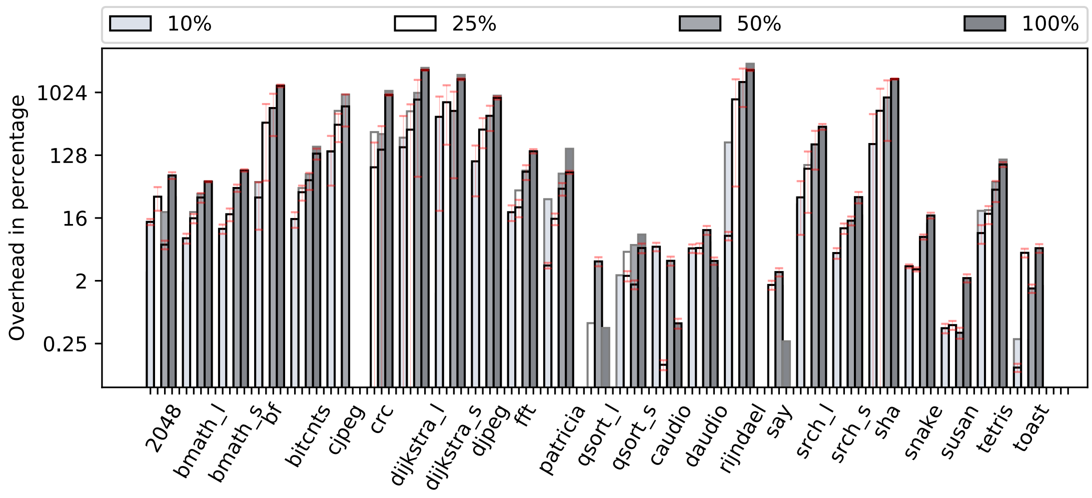


Figure 4.3.: Overhead of VO vs. VO+SC in the optimized mode; the gray-margined bars capture VO+SC protection while the black-margined bars depict VO protection results

Optimized Mode

In this section we report on the average overhead of our protection in the optimized mode. Figure 4.3 captures the results of our experiments. In sum, the induced overhead for VO-only is 48.27%, 151.59%, 231.65%, and 458.42% for protection levels of 10, 25, 50 and 100, respectively. For VO+SC protection, the captured results indicate an average overhead of 32.64%, 150.54%, 231.65%, and 501.04% for protection levels of 10, 25, 50 and 100, respectively.

It it noteworthy to mention that some protected (VO+SC) instances (in both secure and optimized modes) perform better than the unprotected version (VO only). We investigated those binaries manually but were not able to find any problems in them. They appear to benefit much more from the LLVM optimizations compared to other protected binaries.

4.4.4. Security

We analyze threats to the security of our protection from three perspectives: threats to SC protection, threats to VO, and threats to the combined protection. The utilization of *secure* or *optimized* approaches naturally impacts the protection level. Since our optimization is merely applied on the interpreter of VO, the other components are not impacted. Therefore, we make a distinction between the two approaches only in the security analysis of VO.

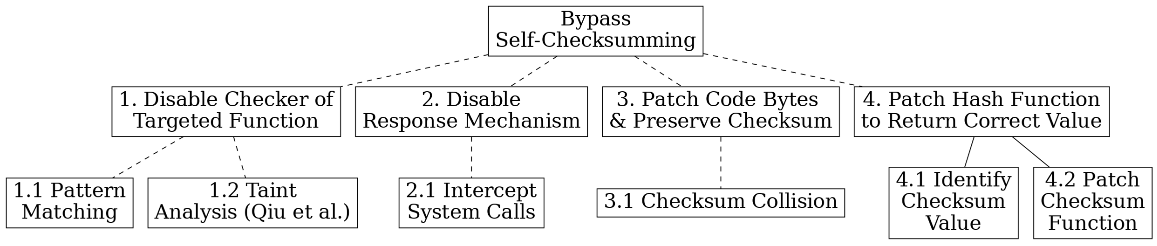

Bypass Self-Checksumming
1. Disable Checker of Targeted Function
2. Disable Response Mechanism
3. Patch Code Bytes & Preserve Checksum
4. Patch Hash Function to Return Correct Value
1.1 Pattern Matching
1.2 Taint Analysis (Qiu et al.)
2.1 Intercept System Calls
3.1 Checksum Collision
4.1 Identify Checksum Value
4.2 Patch Checksum Function

Figure 4.4.: Attacks on SC depicted in the form of attack tree notation [22]

Threats to SC protection

Disabling the self-checksumming is one way to tamper with a protected program. Figure 4.4 represents attacks on SC protection [22] in the form of an attack tree. Dashed lines and solid lines stand for disjunction and conjunction refinements. The following paragraphs describe each of the nodes at depth one in more detail.

Disable Checkers An attacker may disable the checkers for the targeted function. The first difficulty in this case is identifying all guards in functions that are checkers. If attackers successfully identify such checkers, they still might need to identify checkers covering the target checkers (overlapping guards). Since we utilize VO at the function level, attackers have to analyze and reverse engineer the interpreters of all checkers individually. Therefore, the complexity and the amount of work needed for a successful attack is considerably higher.

Qiu et al. [150] present a generic approach for defeating SC protection using taint analysis. However, as also pointed out by the authors, such attacks fail to affect our protection due to the utilization of VO. Their attack identifies the verification of SC hashes (conditional branch) in a given program. Thanks to VO multiple branches (not only SC verifications) may reuse the same handler rendering the attack ineffective.

It is worthwhile to mention that we consider attacks that require a modification in the OS (such as [183]) limited for mass distributions and therefore less critical.

Disable Response Mechanism Disabling the response function is unlikely to be successful, but possible. After detecting a hash mismatch, the interpreter/guard branches to a specific handler and calls the `abort` function. If an attacker were to disable this function call, the program will likely run into a segmentation fault or, at the very least, undefined behavior. Mainly because the value of *VPC* in such cases is not properly set.

Completely removing the call to the response function from the handler or changing the jump address after the comparison of hash values are considered as an attack against the interpreter. These attacks are explained in section 4.4.4.

Patch Code Bytes & Preserve Checksum VO tends to significantly hinder this attack. For machine code it is possible to find a sequence of instructions that perform a useful task, while hashing to the same expected hash. However, the fact that the changes need to go through the interpreter further limits the range of possibilities.

Patch Hash Function An attacker can modify the hash function to always return the correct value for the current checkee. The actual hash value is stored in the function VM. Therefore, reverse engineering VO, particularly the interpreter's accesses to VM, is a prerequisite for the success of such attacks. To find the correct location where the precomputed checksum is stored, it is also necessary to reverse engineer the interpreter, which we discuss in the following section.

Threats to VO

It is important to make a distinction between the approaches (namely *optimized* and *secure*) when discussing threats to VO. The optimized approach (without further hardening) stands worse chances against attacks. Mainly because VPC is replaced with direct jumps to the next block(s). This on the one hand enhances the performance. On the other hand, the optimization eases the control flow recovery of the obfuscated code using static analysis. Utilizing resilient opaque predicates can enhance this situation. We believe the rest of security analysis remains indifferent for both approaches. Therefore, we refer to both approaches as VO in the remainder of this section.

Attacks on VO can be classified into two categories: **i)** generic attacks on VO; and **ii)** manual attacks on the interpreter.

Generic attacks on VO Table 4.2 summarizes a few state-of-the-art attacks on VO along with their advantages and disadvantages.

The approach by Kinder [116] requires knowledge about the VPC used in the interpreter. They assume there exists a function to reliably detect the storage location of the VPC. After our transformation (in the program binary representation) we cannot reliably make assumptions about the location of the VPC. Compiling with different levels of optimizations (e.g. `-O3`) might reserve a register value for it, but it might also end up being written to and loaded from memory.

Coogan et al. [67] propose to use taint analysis in order to identify the relevant instructions, i.e. instructions that affect the values of program outputs in a system trace. This approach is able to identify and remove interpreter instructions from traces. However, it does not remove the SC guards. Moreover, the output of this approach is a simplified trace for each different input, which does not tell the attacker how to identify and disable the SC guards from the protected program.

Yadegari et al. [188] significantly improve the attack by Coogan et al., by more advanced trace simplification techniques, symbolic execution and recombining the simplified traces

	Kinder [116]	Coogan [67]	Yadegari [188]	Salwan [156]
Attack Type	SA[1]	TA[2]	bit-level TA[2]	SA[1] & TA[2], formula simpl., code simpl.
Attacker Goal	approx. data values	significant trace	*CFG*	extract original code
Output	*CFG* and invariants	simpl. trace	simpl. *CFG*	simpl. code
Drawback	assumptions on interpreter structure	equation to *CFG* conversion	large input space	path explosion, symb. reasoning

[1] Static Analysis
[2] Taint Analysis

Table 4.2.: Known attacks on virtualization obfuscation [21, 156]

into a simplified control flow graph (CFG). The drawback of this approach is the large input space that the protected program may accept as input, because in order to reconstruct an accurate CFG, all possible paths in the code must be exercised first. Even if this is achieved, the attacker still has to detect and remove the SC guards, which requires significant effort overall.

Salwan et al. [156] developed a novel approach to defeat VO by combining taint analysis, symbolic execution and code simplification. Compared with known attacks that target the control-flow graph, the output of their script [5] is expected to be a clean devirtualized executable. Their technique also symbolizes the VPC. We were unable to successfully use their tool against executables protected with our protection, due to an error during execution. We have already reported the bug but have not yet heard back from the authors. Nevertheless, we believe that attacking our approach will run into the path explosion problem for the real-world programs in our dataset.

Attacks on the Interpreter In addition to known attacks against VO, it is also possible to attack the interpreter itself without reverse engineering the executed bytecode. In the executable, it is possible to change the jump address of a specific block to another location. This can happen in functions with few branches, where the handler for the branch instruction is not reused. We try to mitigate this by reusing handlers and thus increasing side effects when a handler has been tampered with. Changing the address of a jump will then be changed for *all* branches using this specific handler. This effect can further be increased by adding opaque predicates, generating even more branches and increasing the number of instructions using the same handler.

[5] `https://github.com/JonathanSalwan/Tigress_protection`

Identifying a handler that is only used by a single instruction in a function not being checked seems to be one of the most promising attack scenarios. However, this means that the instruction is not a sensitive instruction and may not be useful for the attacker to tamper with.

In general, the interpreter can be hardened by other protection techniques like any other piece of software. This is also one of the main reasons we have decided to implement our solution using LLVM: to increase the composability with other, existing software protection techniques.

Threats to the Combined Protection

As mentioned in section 4.4.4, attacking SC requires reverse engineering the virtualization obfuscation. SC is implemented on top of VO and thus protected by it. Changing *opcodes* in the generated bytecode can be identified by having SC guards in place. We can conclude that it is necessary to first reverse engineer our VO, in order to defeat SC guards. Ignoring these protections and attacking the interpreter itself, as outlined earlier generates side-effects across each function due to the reuse of handlers.

4.5. Discussion

This section discusses tradeoffs between the performance and the security of our approach. We conclude this section by discussing how the security of our approach could be improved.

4.5.1. Protection Coverage

Our results confirm that our implementation is capable of virtualizing all instructions in the given programs. The SC protection on top of VO is capable of protecting a large portion of instructions (a median of 89%) including the network of checkers. Some instructions however were left unprotected. Our investigations indicate that such instructions reside either in the root of the network of checkers or in the nodes with no incoming edges. Since we do not support cycles, some nodes might be left unprotected. Bear in mind that we can easily achieve 100% coverage over the original program instructions by not checking the guards themselves to avoid cycles. We believe not checking guards downgrades the resilience of the protection. Nonetheless, supporting cycles is a matter of engineering efforts.

4.5.2. Performance vs. Security

As with every software protection, there is a trade off between performance and security. Adding security, for example in the case of our secure interpreter, results in additional code to be executed. The extent to which added protections impact the performance depends on various factors such as the number of comparisons, loops, etc.

As showed in the performance evaluation 4.4.3, our secure implementation yields on average an overhead of 1055.51%. Being $\approx$ 10x slower than the unprotected executable is on the upper end of worst case performances for regular interpreters. However, this approach is also hard for symbolic execution engines to analyze, due to path explosion. An important remark here is the low overhead that SC protection, with a connectivity of 2 checkers per checkee, imposes on the protected binaries. For the 25% protection level the VO+SC protected binaries outperformed VO-protected binaries. As pointed out earlier, the randomness of the network of checkers appears to be the cause of this phenomenon. It is worth mentioning, that the interpreter in this solution has no optimizations whatsoever applied to it.

Further research needs to be conducted to develop and subsequently utilize a set of potential optimizations that do not downgrade security.

The optimized implementation, which utilized indirect threading in the interpreter, resulted in far more efficient protected binaries. Our results indicate an approximate decrease of $\approx$ 50% in the overhead of protection. More importantly, in the optimized mode, the contribution of SC protection to the overhead is significantly decreased. That is, in 10, 25, and 50% protection levels SC did not impose any extra overhead but overheads dropped. However, one can not conclude that utilizing SC yields better performance as other factors such as randomness of checkers and LLVM optimizations need to be taken into account. For the 100% protection level SC constitutes only $\approx$ 43% of the protection overhead (501.04% VO+SC - 458.42% VO only). It appears that LLVM manages to significantly optimize the SC protection. A large portion of the reported overheads stem from LLVM's inability to optimize VO. The use of VM introduces a complexity in the framework's analysis as it transforms direct variable accesses into dynamic array lookups. This makes it difficult for LLVM to analyze control and data flow and thus hinders further native optimizations in IR. Besides, the VM itself imposes indirect accesses by storing and subsequently retrieving values from memory instead of registers, which substantially slows down the execution. In the case of the secure version, the switch statement is translated to multiple comparison and jump instructions, which decreases the effectiveness of the branch prediction. As the value to switch over is loaded dynamically, this can not be improved by static analysis nor compiler optimizations. Instruction prefetching in these cases is also impacted due to the number of jump instructions generated by the switch statement.

We believe as the optimization removes the switch statement and connects each handler with its successors, it becomes easier for LLVM to analyze and run additional optimizations. Despite the performance gain, the indirect threading optimization makes the generated code easier for a human to analyze. Removing the switch also reduces the amount of possible paths allowing more efficient symbolic execution based attacks.

Lastly, the introduced performance overheads heavily depend on the nature of the program to protect. The structure of the program's call graph, number of nested loops, and the degree of IO dependency play important roles in the overall overhead of the protection.

4.5.3. Security

By using a switch statement to dispatch all opcodes, we were able to force symbolic execution engines to run into the problem of path explosions. Even if the original program did not contain many different paths, after transformation every case in the switch statement corresponds to one path. The highest possible number of cases in the generated switch statement is equal to the number of instructions in the original function. This happens when we cannot reuse a single handler. Usually, certain instructions appear more than once, for example to load a value. We try to reuse as many handlers as we can to increase side-effects of attacking the interpreter (see Section 4.4.4). This heavily depends on the structure of the original module, for example the number of instructions in functions. The amount of possible paths can be further increased by introducing opaque predicates [78] in our interpreter.

Our protection can fall short when an attacker can identify a handler that is neither reused, nor protected by SC. Supporting cyclic checks in SC can mitigate this kind of attack. One great benefit of our approach is that any type of LLVM-based software protection can be employed on top or before our transformation to increase the resilience of the protection.

4.6. Limitations

This section presents limitations of our approach and implementation. Several of these limitations can be overcome and we plan to address them in future work.

4.6.1. Unsupported Instructions

There are a few Windows-specific instructions regarding exception-handling that we do not support. This is in part due to the fact that Windows does not allow for loadable modules, and our infrastructure is almost exclusively based on *nix. Since our pass is loaded via LLVM's `opt` tool, we would need to integrate the pass into LLVM's internal optimization pipeline. Doing so increases development time significantly, because every change made to the pass results in having to rebuild a lot of LLVM's tools. We have decided to move support for Windows to future work.

4.6.2. VM Restrictions

Calling `free` on VM can be problematic, when exceptions are involved. This is a well-known source of memory leaks. If a called function were to throw an exception that is not caught, the VM will never be cleaned up. In C++, this is typically solved with a technique called *RAII* [151], but we do not have access to this in LLVM IR. Therefore, in these special cases, our implementation is leaking the memory of the VM. This is an issue that we hope to resolve in future work.

There are instructions in LLVM that require their operands to be constant, meaning they have to be known at compile time. Consequently, we are unable to load those dynamically from the VM. An example for this is the intrinsic LLVM `memcpy` function. Two of its parameters, *alignment* and *volatile*, are required to be constant. Due to the fact that values loaded from the VM at runtime are not constant, in these cases we reuse the original instruction's operands.

Another restriction of using a VM is that we cannot add global variables to it. Since all of our generated instructions operate solely on the VM, we would not assign to the global variable. Calling another function that depends on the updated value would change the program's behavior. This could, in theory, be solved by updating both the global variable, as well as the VM. However, calling a function that updates the global variable would require us to update the value in the VM. This requires performing non-trivial control flow analysis to determine read and write accesses to the global variable. A simple solution to this issue is to not add global variables to the VM, but instead simply use the global variable itself.

4.7. Conclusions

We designed, implemented and evaluated an LLVM-based protection that effectively combines virtualization obfuscation (VO) and self-checksumming (SC). Our approach removes the need for binary post-patching of SC pre-computed hash (checksum) values. As a direct consequence, SC is completely architecture-agnostic and better yet, fully composable with other protection techniques. More importantly, our SC scheme benefits from the hardening added by VO and thus exhibits a higher resilience.

Regarding VO, we presented and later evaluated two implementations, namely *secure* and *optimized*. We conducted a set of performance as well as security evaluations, where a set of 25 real-world programs (MiBench and CLI games) were used. The secure version stands better chances against symbolic execution attacks. However, it imposes an average overhead of $\approx$10x for full protection (i.e. 100% of instructions are protected), of binaries. In contrast, the optimized version imposes an average overhead of 5x for the same protection level. The SC protection itself imposes only 43% overhead for full protection with a connectivity of two. We believe such overheads are acceptable when protection is applied on a subset of sensitive segments of programs (e.g. license checking).

In the security evaluation we discussed attacks on SC and VO. Possible mitigations were also explained in our evaluations. Since our solution adds SC on top of VO, an attacker has to first break the obfuscation, in order to be able to attack SC. To the best of our knowledge, there exists no tool that can automatically carry out such attacks. Therefore, a combination of tools need to be manually applied by attackers to break the scheme.

As future work we plan on implementing cyclic checks for SC. Another interesting area is to further investigate means to optimize VO without (or with less) security sacrifices.

Part III.

Software Integrity Protection Composition

5. Composition Framework

This chapter presents the design and implementation of our software integrity protection composition framework. The content of the chapter was previously published in work [8] co-authored by the thesis's author.

5.1. Introduction

One way to achieve a higher degree of resilience against a wider range of attacks is to utilize a multitude of protections simultaneously. We refer to this process as *composition of protections* in the remainder of this document. Composing protections not only mitigates more attacks but also forms multi-layer defenses, which protect each other and are hence harder to be defeated by perpetrators. However, due to the brittle nature of integrity protection schemes, composing such protections is an open problem in literature. On the one hand, composing oblivious hashing (OH) [49] and self-checksumming (SC) [22] could lead to a cyclic dependency between the two and ultimately could produce faulty binaries (we discuss different conflicts in detail later in Section 5.3). On the other hand, non-cyclic dependencies are desired because code segments that are directly and/or indirectly checked (protected) by multiple integrity checks are more resilient to code patching attacks. Furthermore, protections have different capabilities when it comes to protecting program assets. For instance, OH is unable to protect nondeterministic program segments, while SC does not have such a restriction [5]. Moreover, the locality of the checks can have a huge impact on the imposed overhead [40]. Simply put, carrying out expensive checks in frequently executed code segments can induce large overheads. Therefore, deciding upon where and how to apply different protections in the given program can severely impact security and performance of the protected programs. To sum it up, we believe composing a range of protection techniques can strengthen the software resilience against MATE attackers.

Problem. Naively composing protections is ineffective and can potentially lead to conflicts, which can severely limit the application and coverage of protections.

Gaps. To the best of our knowledge (see Section 5.2), the gaps in literature are:

- The potential conflicts arising from composing protection schemes were not thoroughly studied;

- The reviewed literature makes conservative assumptions as to whether protection schemes are composable or not, i.e. conflicts are prevented by forbidding the composition rather than handling them upon composition; and
- The existing compositions do not offer a concrete optimization methodology for better performance and security w.r.t. the program at hand.

Contributions. This chapter closes the aforementioned gaps by:

- Identifying three representative types of conflicts in the integrity protection composition (see Section 5.3);
- Proposing a graph-based technique for protection composition, enabling the detection and handling of the conflicts at the program level (see Section 5.4);
- Utilizing ILP to optimize the composition for better performance and security w.r.t. the program at hand (see Section 5.4);
- Implementing the proposed technique as an (open source) end-to-end framework for a conflict-free composition of protection schemes (see Section 5.5);
- Empirically evaluating the effectiveness and efficiency of the proposed technique using a set of real-world programs (see Section 5.6);

Section 5.7 describes threats to validity. Section 5.8 presents conclusions and ideas for future work.

5.2. Background and Related Work

This chapter contributes to two areas: software protection composition and protection optimization. To introduce the area of software protection, we first review a set of representative integrity protection techniques that not only motivate the whole idea of the composition but are also used in our empirical evaluations.

5.2.1. Integrity Protection Schemes

Self-checksumming (SC)

SC [22, 47, 87, 88, 104] operates by injecting a set of guards that hash or checksum the desired (sensitive) segments of program code in memory (at runtime) and verify that the hashes match to expected values. Because the expected values can only be known after compilation a set of placeholders are used during compilation. An adjustment process is executed after compilation, to patch the placeholders with the actual expected values. Consequently, if perpetrators tamper with the program code, the guards can detect them. Once inconsistencies are detected, a *response mechanism* is triggered [139], e.g. by injecting a fault into the stack that eventually causes a crash in the application. SC guards can further be hardened interconnecting them such that they protect each other [47]. SC guards add atypical behavior to a program, i.e. the self memory access, which exposes the guards to taint-based attacks [150]. SC is also susceptible to the memory split attacks [183]. One advantage is that SC is capable of protecting any program segment with no restriction whatsoever.

Oblivious Hashing (OH) and Short Range Oblivious Hashing (SROH)

OH computes hashes of the program's execution trace (including memory values), by incorporating the memory reference of instructions at runtime [49] and subsequently verifying them (i.e. matching them to expected values) at random intervals. One way to adjust expected values is to resort to a technique similar to the SC post-patching technique [5].

OH captures the effect of the program execution, not its actual code and thus cannot be deceived by attacks such as the memory split attack. However, OH can only be applied to deterministic segments of a program, which are executed for any given input. This restriction severely limits the application of OH to only small parts of programs. SROH [5] partially removes this restriction by extending the coverage of OH to constant data residing in nondeterministic branches as well as all the branching conditions. The input-data-dependent instructions in a program cannot be covered by SROH nonetheless.

Call Stack Integrity Verification (CSIV)

CSIV aims to guard the access to sensitive functions in a given program [19]. That is, only authentic sequences of calls can reach such functions. Any attempts to illegitimately jump to them triggers a response mechanism. To do so, a shadow stack can be created in which all (or part of) function calls on the path to sensitive functions are reflected into a hash variable. This can be achieved in two steps: **i)** accumulatively hashing random tokens [2] in functions leading to a sensitive function; and **ii)** verifying the expected hashes in sensitive functions. As opposed to SC and OH which protect the code bytes, respectively execution memory, CSIV only protects the intended control flow.

Code Mobility (CM)

CM aims at thwarting reverse engineering by mitigating static and dynamic analyses [77]. It partitions a given program into a client part and server part. CM moves the sensitive code to the server section and places corresponding calls to the server in the client. The server upon request delivers the sensitive code in pieces. This limits the amount of the code that an attacker can get access to at once. CM imposes a constant connection between client and server.

The Need for Obfuscation

When it comes to integrity protection schemes obfuscation is no longer an additional layer of protection but a must. The reason being that protection guards are commonly injected into programs by instrumentation techniques. As a direct consequence, they end up having the same syntactical resemblance. This makes them extremely vulnerable to pattern-matching attacks. Plenty of proposed schemes unanimously agree on the gravity of applying obfuscation to protections [9]. Theoretically speaking, applying obfuscation shall not cause a conflict in protections. However, the application of obfuscation in practice could lead to failures, particularly in the adjustment processes (see Section 5.3.1).

5.2.2. Composition of Protections

The problem of protection composition is comprised of two subproblems: (1) *deciding on the order in which transformations shall be applied* and (2) *deciding upon which program segments shall be protected with which protections to avoid conflicts while maximizing security and minimizing the cost*. We discuss each of these problems in the following paragraphs.

Computing the optimal order of protections is equivalent to the problem of phase ordering in optimizing compilers and hence is undecidable [176]. Since the amenability of protection to each other can be known upfront, we need not to compute an optimal order of protections. Instead, we resort to a set of heuristics as we discuss in Section 5.4.8.

The problem of applying which protections on which segments of the given program was studied in the context of software obfuscation. Heffner et al. [96] proposed a technique for composing obfuscation transformation by encoding dependency relations among transformations in a probabilistic *Finite State Automaton (FSA)*. The probabilities capture different metrics as to which obfuscation is more desired to be applied. The composition is then achieved by a random walk on the FSA. Similarly, Liu et al. [127] presented a technique that uses an iterative search-based algorithm to identify an optimal sequence of obfuscations over particular partitions of a given program. Guelton et al. [92] identified a set of arising conflicts amongst obfuscations provided in two open source engines (namely Tigress and O-LLVM) and clang optimization passes. Su et al. [167] proposed the design of a composition technique using Petri nets and inhibitor arcs to compute a sequence in which protections should be applied, in order to avoid conflicts, to a program. However, as we show in this chapter the type of conflicts and metrics presented for obfuscation are not transferable to the software integrity protection composition problem.

ASPIRE project (`https://aspire-fp7.eu/`) is a framework that enables chaining of integrity protection techniques. Nonetheless, not all of the protections can be composed due to the potential rise of conflicts [185]. For instance, in their tool chain, code mobility and static remote-attestation shall not be utilized simultaneously. Such preventions are hard coded in the framework within the tool configurations. In this work, we aim to utilize potentially conflicting protections at the same time by detecting and further handling such conflicts. Furthermore, we optimize both conflict handling, i.e. the decision on which protection shall be removed from a conflicting segment, and locality of protections in the program for better performance and security (coverage) of a protected program. To achieve these goals, we model our composition problem as a graph problem and subsequently use ILP to optimally solve it. Our transition from expressing nodes and their relations in a graph to an ILP problem is inspired by the techniques introduced by Boulle et al. [33] and Wrighton et al. [181].

5.2.3. Protection Optimization

Performance (Overhead)

The locality of the guards that carry out the checks plays an important role in the imposed overhead. That is, if guards are placed in very frequently executed code (also called "hot code"), e.g. code inside nested loops, the overhead is potentially high. Cappert et al. [40] proposed a metric to measure the *hotness* of a function based on its call frequency captured in a dynamic profiling analysis. In their protection, they favor colder functions as checkers in an attempt to induce less overhead. In this work, we analyze code hotness at a lower level, namely basic blocks, along with other factors to optimize the locality of checkers of different types.

Security (Coverage)

An important metric in measuring the security of protection schemes is their *connectivity* [72, 87, 88] that captures the number of unique guards that protect a particular program element (instruction, basic blocks, function, etc). Understandably, achieving higher connectivity entails having more guards which (in-)directly check other guards, yielding larger overheads. In this work, we aim at reaching a desired connectivity level, while minimizing the overhead. Since composed protections can potentially protect each others' guards, we propose a new metric as *implicit coverage* that captures the program elements guards of which are protected by other guards, i.e. indirect checking.

5.3. Requirements

5.3.1. Conflict Detection

To understand the arising conflicts when combining protections, we attempted to apply the protection schemes described in Section 5.2.1, in various orders on a representative dataset of programs. In this section, we briefly report on the conflicts that we encountered in our naive composition attempts. From the identified conflicts we then derive a set of requirements for our composition framework.

OH and SC

Both OH and SC use the concept of hashing and verification. OH hashes memory references, while SC hashes the process memory corresponding to a range of instructions, basic blocks or functions, at runtime. One can apply each of these schemes once or multiple times in arbitrary orders. Both schemes require a finalization step (post-patching) in which the expected hash values are adjusted in the protected binary. When OH and SC protected segments cyclically overlap, the adjustment of OH hashes invalidates previously computed SC hashes and vice versa. Consequently, binaries containing such conflicts would trigger false tamper-detection alarms. This conflict in essence can occur between any pair of overlapping schemes that rely on a post-patching step.

One solution for this conflict is to resort to cross-protection placeholders (pivot bytes) like those presented in [22] to adjust values without invalidating others. However, the pivot bytes themselves can not be protected by any of the composed protections and thus they remain at risk of tampering attacks. Alternatively, we could keep cycles intact by resorting to commutative (reversible) hash functions, which are intuitively easier to be defeated by attackers. Therefore, we rather want to detect and subsequently prevent such cyclic conflicts in our composition. Simply put, *deterministic finalization order* is the first requirement of our system.

SC/OH and CM

Chaining CM with SC yields a different kind of conflict. CM mobilizes protected functions to a server from which they are retrieved only when needed and cached in a random memory address, such that the attacker does not know where to find the function. Since the actual location of the mobilized functions shall be unpredictable, SC guards will be unable to find such functions at runtime, hence SC cannot protect CM protected functions. To avoid this type of conflict, SC needs to mark protectee segments in the code such that they cannot be removed from the code by the CM protection. Conflicts of such can be detected by *safely tracing reference changes*. Unlike SC, OH needs not to know the address of mobilized functions. Applying CM on OH-protected functions as well as the inverse case impose no conflicts.

Obfuscation and SC/OH

Obfuscation protects the confidentiality of data, logic and the location of code in many protection schemes. Without obfuscation plenty of protection measures can readily be defeated by pattern matching. Obfuscation can conflict with protections, too. If it is applied after inserting the SC and/or OH guards, but before the final adjustments (when SC and/or OH are utilized), it may change the resemblance of placeholders introducing faults in post-patching steps. If obfuscation is applied after final adjustments, it can break SC checks as hash and length of obfuscated blocks may differ from what SC guards expect.

Syntactical changes of basic blocks, their location, and lengths cause conflicts in composition of obfuscation with schemes that rely on syntactical values. In our system, we are in need of a *safe value change tracing* mechanism to prevent such conflicts. In other words, we need to mark certain pieces of code as unobfuscatable.

It is worthwhile to state that the identified conflicts are a complete set of possible conflicts for the given set of protections. That is, any other composition of the mentioned protections does not cause conflicts.

5.4. Design

5.4.1. Running Example

To illustrate our design and improve readability, we use a fictional vehicle mileage counter program (Listing 5.1) as our running example throughout this section. The program has two global variables: `RotationCount` and `MILEAGE`. The `onWheelRotationCompleted` function is an event handler that is triggered on every complete wheel rotation. After 100 rotations, a mile is added to the car's mileage counter, which is an increase-only variable. The `MILEAGE` is merely altered in the `incrementMileage` function. In this example perpetrators aim to undermine the integrity of the mileage counter.

Listing 5.1 Fictional car mileage counter program

```
int RotationCount = 0, MILEAGE = 0;
void onWheelRotationCompleted() {
  RotationCount += 1;
  if(RotationCount >= 100) {
    incrementMilage();
    RotationCount = 0;
  }
}
int incrementMileage() {
  MILEAGE += 1;
  return MILEAGE;
}
```

5.4.2. High level Process

To achieve low coupling amongst the different protections and the composition transformation, each protection transformation stays completely unaware of the composition taking place in the framework. However, we do not want protection transformations altering the program-to-protect in one go, because conflicts might need to be settled before actually applying protections. To achieve this goal, we refactor transformations into *two-step transformations*. In the first step, the transformation generates a set of proposals as to which segments it can protect in the given program (regardless of other protections that might be part of the composition). Additionally, each protection submits a set of constraints (Section 5.4.4) defining under which assumptions the protection will function properly. The second step, carries out the transformation for the requested proposals on demand. In order to be composable, all the protections need to be refactored to comply with this two-step transformation requirement.

Our composition framework executes the first step of protection transformations and collects their protection proposals as well as their constraints. The collected proposals and constraints are then merged to form *protection manifests* (Section 5.4.3). Subsequently, the created manifests are projected on a *defense graph* (Section 5.4.5) upon which conflict analysis (Section 5.4.6) and optimizations (Section 5.4.7) are performed.

A possible outcome of applying SC, SROH, and CSIV protections on the running example is illustrated in Listing 5.2. In the following sections we will explain how this outcome has been obtained using our composition framework.

5.4.3. Protection Manifest

A *Protection Manifest* is a data structure comprising of a protection guard, a set of instructions protected by that guard, and constraints under which the protection behaves as intended. Simply put, manifests reflect how and where the protection is going to be applied on the code. Each protection technique like SC, OH/SROH, and CSIV proposes a multitude of manifests in the first step of a *two-step transformation*.

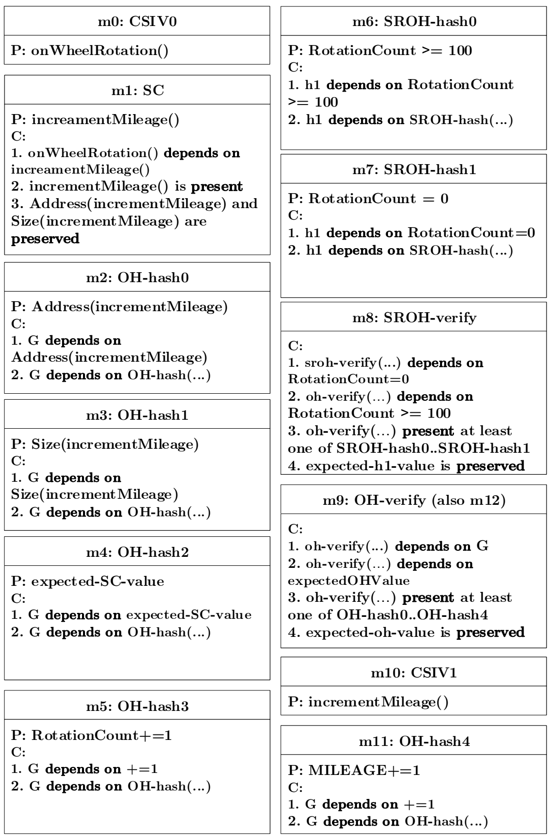
Figure 5.1 depicts 12 protection manifests output by our approach on the running example from Listing 5.1. Note that manifest m12 is the same as manifest m9, but they are placed in different functions. The name of each manifest includes the protection technique acronym and optionally an appended ordinal number for the sake of unique names. Inside the rectables representing manifests we see one or more lines prefixed by "P"(indicating the protected instructions) and/or "C" (indicating the constraints, described in Section 5.4.4). SC and CSIV protections create stand-alone manifests, while OH and SROH protections represent each guard as a dependent pair of *hash* and *verify* manifests. The *verify* manifests do not contain any "P", just "C", because they are verifying the "P" from *hash* manifests. This split of *verify* and *hash* manifests, facilitates the cyclic dependency analysis of manifests. Each manifest keeps a reference to the protection instructions, which are not included in Figure 5.1 due to space limitations. As a direct consequence, our composition framework, can easily decide to drop a manifest (e.g. due to an arising conflict) resulting in the removal of the corresponding guard instructions.

Listing 5.2 Protected fictional car mileage counter program, note that all protections are applied at the LLVM IR level, however for the sake of simplicity our snippet is presented at the source code level

```
int RotationCount, MILEAGE = 0;
long G = <random token>; //OH hash variable
void onWheelRotationCompleted(){
  long h1 = <random token>;
  //overlapping SC and OH protection
  long computedSCHash=
      SC_hash(t=<AddIncrementMileage>,OH_hash(G,t), t),
      (t=<SizeIncrementMileage>, OH_hash(G,t),t);
  //overlapping SC and OH protection
  SC_verify(computedSCHash, (t=<expected_SC_value>, OH_hash(G,t), t));
  RotationCount += (t=1, OH_hash(G,t),t);
  if(t= RotationCount >= 100, SROH_hash(h1,t),t) {
    CSIV_Register();
    m = milageIncrement();
    RotationCount = (t=0, SROH_hash(h1,t),t);
    SROH_verify(h1, <expected_h1_value>);
  }
}
int incrementMileage(){
  CSIV_Register();
  CSIV_Verify(); //Sensitive funcions include a control flow verification
  MILEAGE += (t=1,OH_hash(G,t),t);
  OH_verify(G,<expected_oh_value>);
  G = <token> //the same token as in line 2
  return MILEAGE;
}
```

5.4.4. Constraints

As pointed out earlier, different protections can be applied individually to a program. However, each technique may yield different types of conflicts, if composed. To be able to cope with those conflicts, we resort to a set of protection constraints, which are bundled up in the manifests (indicated by "C:" in Figure 5.1). Constraints are a set of rules that each protection scheme adds to their manifests. That is, protections will work as intended if and only if all the constraints are satisfied throughout the composition process. Having protection schemes specifying constraints in their manifests yields better encapsulation as the composition itself need not to hold any scheme related knowledge.



m0: CSIV0
P: onWheelRotation()

m1: SC
P: increamentMileage()
C:
1. onWheelRotation() **depends on** increamentMileage()
2. incrementMileage() is **present**
3. Address(incrementMileage) and Size(incrementMileage) are **preserved**

m2: OH-hash0
P: Address(incrementMileage)
C:
1. G **depends on** Address(incrementMileage)
2. G **depends on** OH-hash(...)

m3: OH-hash1
P: Size(incrementMileage)
C:
1. G **depends on** Size(incrementMileage)
2. G **depends on** OH-hash(...)

m4: OH-hash2
P: expected-SC-value
C:
1. G **depends on** expected-SC-value
2. G **depends on** OH-hash(...)

m5: OH-hash3
P: RotationCount+=1
C:
1. G **depends on** +=1
2. G **depends on** OH-hash(...)

m6: SROH-hash0
P: RotationCount >= 100
C:
1. h1 **depends on** RotationCount >= 100
2. h1 **depends on** SROH-hash(...)

m7: SROH-hash1
P: RotationCount = 0
C:
1. h1 **depends on** RotationCount=0
2. h1 **depends on** SROH-hash(...)

m8: SROH-verify
C:
1. sroh-verify(...) **depends on** RotationCount=0
2. oh-verify(...) **depends on** RotationCount >= 100
3. oh-verify(...) **present** at least one of SROH-hash0..SROH-hash1
4. expected-h1-value is **preserved**

m9: OH-verify (also m12)
C:
1. oh-verify(...) **depends on** G
2. oh-verify(...) **depends on** expectedOHValue
3. oh-verify(...) **present** at least one of OH-hash0..OH-hash4
4. expected-oh-value is **preserved**

m10: CSIV1
P: incrementMileage()

m11: OH-hash4
P: MILEAGE+=1
C:
1. G **depends on** +=1
2. G **depends on** OH-hash(...)

Figure 5.1.: Generated protection manifests for the example program; `P` and `C` refer to protected segments (instruction/function) and constraints, respectively

In our framework, we define three types of constraints, which directly correspond to the three requirements introduced in Section 5.3.1:

1. *Order* constraints between manifest m_A, and manifest m_B, denote that m_A can only be finalized after m_B. For example, in the m1:SC manifest from Figure 5.1, `c1` indicates that `onWheelRotation` depends on `incrementMileage` and therefore the latter has to be finalized before. In a broader sense, we use the order constraints to capture dependency relationships between protection and program instructions. OH/SROH hash manifests use the very constraint to capture the dependency of their hash variables to hashed instructions.

2. *Preserve* constraints indicate that the value of certain marked instructions shall not change. As depicted in Figure 5.1 preserve constraints are used by both SC and

OH/SROH verify manifests to mark their placeholders as unmodifiable.

3. *Present* constraints between m_A and m_B denotes that m_A can only exist if m_B is present in the program. SC manifests mark their protectee functions with the present constraint so they stay in the binary, e.g. `incrementMileage` is marked with such constraint in our example. We use another variation of the present constraint to capture the relation between split OH/SROH hash and verify manifests. Hashes that are not verified are effectively useless as they add no protection. To prevent the existence of hash manifests without the corresponding verify manifests, we use present constraints such that verification manifests have to choose $n \geq 1$ out of the total number of relevant hash manifests.

5.4.5. Defense Graph

To enable automated composition of protections, we first need an abstract representation of: protection guards, protected segments, and the overlaps between protections. Casting protection guards, their corresponding protectees, and constraints into a unified representation aids detection of conflicts. Moreover, such a representation supports reasoning about the resilience of the composed protections, for instance, by measuring the number of guards that protect other guards. Additionally, the representation needs to be flexible enough to accommodate further information about program at hand. Profiling information about program basic blocks can support optimization decisions regarding the location of protection guards.

We propose to use a graph-based representation as it effectively meets all our needs. The composition pass populates the graph from the proposed manifests and protection protectee relations between program sub-structures and protections. Since protections are applied at different level of granularity (e.g. SC is applied at the function level, while OH/SROH is employed at the instruction level) graph nodes can refer to functions, instructions, or manifests. Solid black arcs represent *dependency* (order) constraints. Dashed arcs represent *present* contraints over multiple elements, i.e. constraints comprised of two or more elements are translated to arcs between their corresponding nodes in the graph. Whilst, constraints over single elements (namely preserve and present constraints) are added to their corresponding nodes as attributes. It is important to bear in mind that constraints are transitive. That is, if a function has incoming/outgoing arcs, all of its instructions will inherit the same relation (arcs). Figure 5.2 depicts a simplified graph for the running example. The gray arcs capture the transitive arcs, i.e. inherited dependency relations from functions to their instructions. For the sake of brevity, we omitted arcs that are deemed to be less interesting. As illustrated in the figure the SC manifest leverages from an additional protection by three OH hash manifests (namely `OH_hash0`, `OH_hash1`, and `OH_hash2`).

We extend our graph nodes with a set of properties to accommodate for all the necessary meta data. For example, `<ExecFreq(N)>` where `N` refers to the execution frequency of

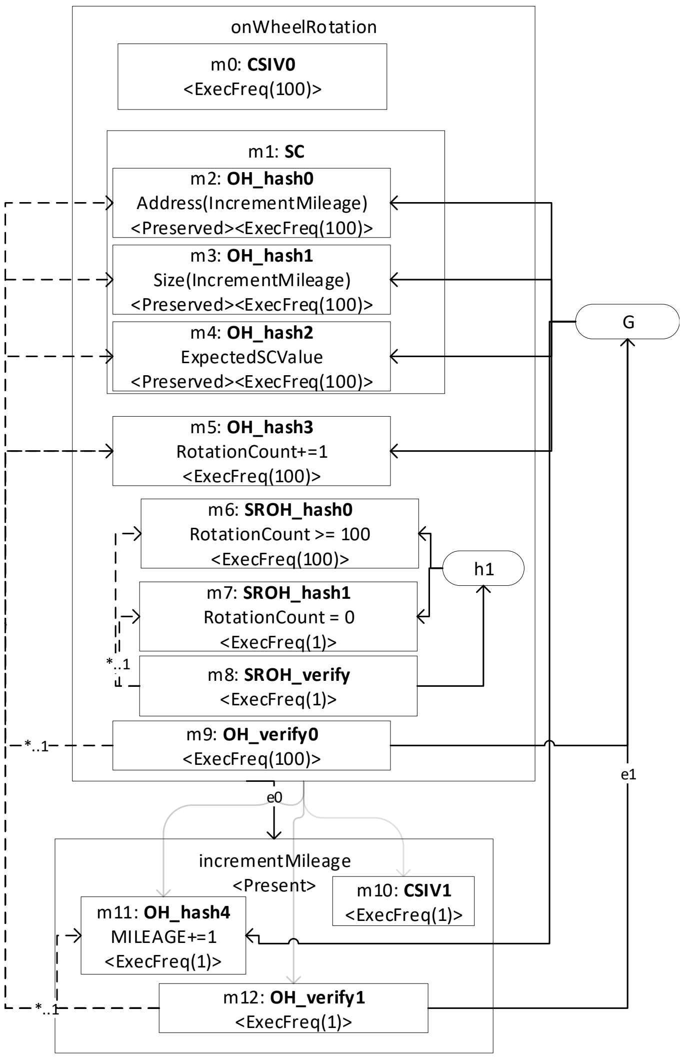


Figure 5.2.: Defense graph for the manifests of the example program; node attributes are depicted in the form of `<Attribute>`; Manifests are shown as sharp-edged rectangles while program instructions (`G` and `h1`) are shown as round-edged rectangles.

the block in which the manifest resides. We collect such data by profiling binaries and subsequently normalizing them.

5.4.6. Conflicts and Conflict Handling

Depending on the way that manifests are composed, *order* and/or *present* constraints may hold *true* or *false* values. When these types of constrains are *false*, the corresponding manifests are not applied for protecting the program. However, *preserve* constraints must conserve segments from modifications, which is important to be honored by the obfuscation passes. Therefore, no selection of manifests violates *preserve* constraints. Manifests carrying *present* constraints can only be selected if and only if their required manifests are selected as well.

Honoring *order* constraints requires a delicate detection technique as a set of order constraints can collectively result in ambiguities. For example, assume the defense graph: $m_A \to m_B \to m_C \to m_A$, where $\to$ denotes the order constraint indicating that the left-hand manifest must be applied before the right-hand manifest. In the this example, a cyclic dependency materializes that renders the finalization order of the manifests impossible. That is, finalizing in any order results in invalidation of already finalized nodes. In the defense graph from Figure 5.2, the transitive edge `e0` together with `e1` form a cycle. Our strategy is to break cycles by removing one or more manifests in the cycle. However, our framework needs to detect cycles in the first place. To efficiently detect cycles in the defense graph, we utilize the strongly connected component (SCC) analysis proposed by Nuutila [142]. After cycle detection, there is more than one solution to the manifest removal problem. One out of all the manifests contributing to a cycle may randomly be removed, which would break the cycle. Since manifests in the graph correspond to protection mechanisms, the decision as to which manifest shall be removed entails different security and performance trade-offs. To balance those trade-offs, we are in need of a set of security as well as performance metrics. Therefore, we resort to a set of metrics to measure the impact of each manifest on the security and performance overhead. To find an optimal solution, we employ ILP to handle conflicts as described in Section 5.4.7.

5.4.7. Optimization

Our goal is to minimize the overhead of protections while attaining a set of security requirements (i.e. explicit and implicit coverage of protections). Running protection passes yields a set of candidate manifests, $M_{candidates} : \{m_0, m_1, ..., m_n\}$, where n corresponds to the total number of protection manifests, and $m_i \in \{0, 1\}$ for $0 \leq i \leq n$. Precisely put, our goal is to select a subset of manifests, i.e. a concrete assignment of each $m_i \in M_{candidates}$, which satisfies our security requirements with the minimum possible overhead.

Security Requirements

For every manifest, the composition pass calculates the values for the following metrics: (1) *explicit instruction coverage*, which measures the instructions explicitly protected by manifests and (2) *explicit block coverage*, which measures the basic blocks explicitly protected by manifests. Thereupon, desired security constraints can be written in the form of inequalities: $\sum_{i=1}^{n} P_{j_{m_i}} \geq D_j$ or $\sum_{i=1}^{n} P_{j_{m_i}} \leq D_j$, where $0 \leq j \leq p$; p, P_j, and D_j correspond to the total number of scores computed for manifests, the j^{th} score of a manifest, and the desired value for the score of interest, respectively. For example, `m1:SC` will be assigned with explicit instruction coverage of 6 (lines of code in `incrementMileage`), and explicit block coverage of 1 (there is only one basic block in `incrementMileage`). This generic inequality can be used to have constraints over any metric value of manifests, except for the implicit coverage metrics (both block and instruction).

Implicit coverage rather depends on two (or more) manifests: a protector and a protectee. Therefore, it is impossible to measure the implicit coverage of a manifest without prior knowledge of other selected manifests in the solution. For instance, `m2, m3,` and `m4` are implicitly protecting the `incrementMileage` function as they protect the guard (`m1:SC`) that is protecting that function. However, their implicit coverage is equivalent to the explicit coverage of `m1` if and only if `m1` is selected in the final solution, otherwise it is 0. Furthermore, there could exist multiple arcs implicitly protecting the same set of instructions. For example, there are three OH hash manifests (`m2`, `m3` and `m4`) that are implicitly protecting `m1`. To precisely calculate the implicit coverage, we need to detect duplicates and subsequently consider them in our computation.

To detect duplicates, we introduce two sets of binary auxiliary variables: $e_{i,j}$ and $f_0, ..., f_n$, where $0 \leq i, j \leq n$ and n corresponds to the total number of manifests. Every $e_{i,j} \in \{0, 1\}$ indicates whether there exists an arc between m_i and m_j in the defense graph or not. It is important to note that an arc ($e_{i,j}$) can only exist if and only if both of its manifests (m_i and m_j) are present in the solution. We formulate the desired relation by introducing a constraint such as $0 \leq m_i + m_j - 2e_{i,j} \leq 1$ for all arcs. To deal with possible duplicate arcs, we ensure that every manifest's implicit coverage is counted only once for all the duplicate arcs. For this purpose, we first constrain auxiliary variables f_i as a disjunction of arcs to m_i, i.e. $E_i = \{e_{j,k} | i = k\}$, $0 \leq |E_i| \times f_i - \sum_{c=e_{j,i}}^{E_i} c \leq 1$. Note that E_i is a list capturing all arcs ending in m_i (sink). In our runinng example, for m_1 three duplicate arcs exist: $e_{2,1}$, $e_{3,1}$ and $e_{4,1}$, which are not visible in Figure 5.2, because m_2, m_3 and m_4 protect m_1's guard code. This yields a constraint such as $0 \leq 3 \times f_1 - e_{2,1} - e_{3,1} - e_{4,1} \leq 1$. Subsequently, for a precise calculation we use f_i as m_i's coefficient for the implicit coverage, i.e. $\sum_0^n f_i \times implicit(m_i) \leq K$. Note that the $implicit(m_i)$ function can either return the size of implicitly protected instructions or blocks by m_i. At this point we can compute solutions with the desired implicit coverage by setting K to the value of interest.

Conflict Constraints

We formulate our 3 protection constraints in the form of inequalities. To break a cycle, an arc, from the set of arcs taking part in the cycle, needs to be removed. We can formulate this as $\sum_{i=1}^{C} m_i \leq C - 1$, where C denotes the count of manifests in a cycle. In our example, `m1`, `G`, and `m12` form a cyclic conflict. A corresponding avoid-cycle constraint will limit the solver's choice to either of `m1` or `m12` manifests. Since the objective function is subject to other security as well as performance constraints, the manifest with the least contribution to the desired properties will be removed.

For every set of manifests that form a cycle, we add the afore mentioned conflict constraint to the set. Since the SCC analysis may fail to detect all cycles, we reanalyze the outcome of the optimization to identify further (sub-)cycles. All the newly identified cycles are repeatedly added as constraints to the system till no more cycles are detected.

For *present* constraints, we add a constraint $m_i \leq m_j$, where i is the index of the manifest that cannot exist without the manifest at the index j. It turns out that *preserve* constraints do not impose any limitations on the choice of manifests as they are rather restrictions on the utilization of obfuscation on preserved placeholders. Therefore, we exclude them from our optimization process.

5.4.8. Order of Protections

Some protections can protect the guards of other protections. In such cases, a layered protection is recommended. The instructions explicitly protected by the protectee guards are implicitly protected by the protector guards. To defeat such chained guards, attackers may need to find and disable various guards of different types, which intuitively hardens the composed protection. Such chains of protections can only materialize when protections are applied in particular orders. Therefore, the order in which protections are applied is important.

We believe OH shall be applied after SC as SC guards can benefit from the additional protections of OH [5]. Both SC and OH can protect CSIV guards, while only OH can cover CM protected functions. Precisely put, the order of protection is SC $\rightarrow$ OH/SROH $\rightarrow$ CSIV $\rightarrow$ CM. It is worth noting that all obfuscations should be applied after integrity protection passes, but before finalizations, to maximize their coverage over integrity protection guards.

An interesting take is that protections can be applied multiple times. In such cases, guards can potentially have a higher number of overlaps. Although our design supports a multiple application of schemes, we consider it out of the scope of this work.

5.5. Implementation

All the mentioned protection schemes have an LLVM-based open source implementation. For compatibility and portability reasons, we also decided to develop our composition framework in LLVM. Our entire tool chain is open source (see **??**).

Figure 5.3 illustrates the modules in our framework along with their relation to protection and LLVM modules. The green boxes indicate the newly developed modules in the the system. Our optimizations are implemented as LLVM transformation modules. Protection passes (CSIV, CM, SC, and OH/SROH) are first executed which results in a set of manifests. The existing protection passes (CSIV, CM, SC, and OH/SROH) were re-factored to to

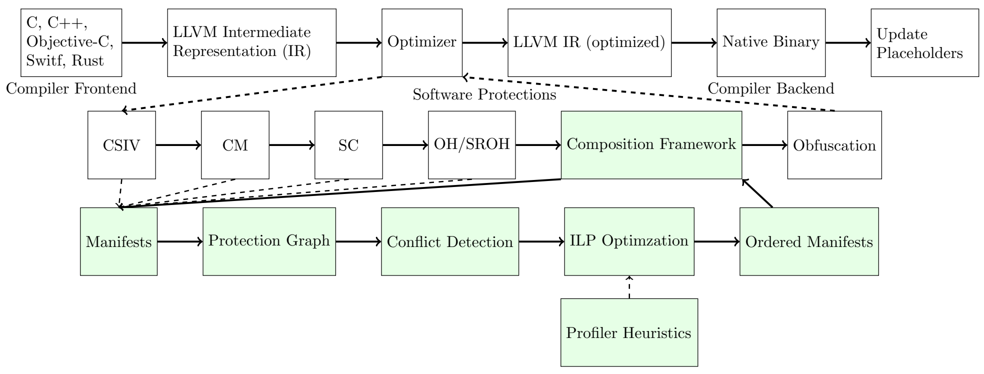


Figure 5.3.: Module representation of our solution in LLVM framework; green boxes distinguish composition modules from LLVM and protection modules.

support the two-step transformation according to Section 5.4.2. It is worthwhile to mention that supporting further protections in our framework is rather straightforward. To do so, a protection pass needs to be refactored to conform to our framework's two-step *propose* and *apply* interfaces.

The composition pass collects the generated manifests from which it populates a defense graph. We use the lemon graph library (`https://lemon.cs.elte.hu`), due to its performance and flexibility, to build our defense graph with specialized arcs and nodes. Thereafter, a set of ILP constraints from present and order manifest-constraints are generated. Further security requirements are also added to the problem space in the form of integer linear constraints. We use `glpk` library (`https://www.gnu.org/software/glpk/`) to solve the linear problem. Once the optimal solution is found (if security constraints are satisfiable), a conflict handling pass is engaged that remove all the manifests which are not selected by the solver. The last step is to define the finalization order of the manifests. This is important as the correctness of the post patching step relies on this order. In our framework we unified the adjustment step to follow the reverse topological patching order to avoid inconsistencies in placeholders.

5.6. Evaluation

In our evaluation, we run experiments to measure the security as well as performance of our composition. For performance, we benchmark the protection overhead for different coverage levels. Moreover, we conduct an experiment in which we compare our composition optimization to a benchmark in the literature. Regarding security, we measure protection coverage and we perform a threat analysis using attack trees.

5.6.1. Dataset

To evaluate effectiveness and efficiency of our solution, we carry out a set of empirical experiments on a dataset comprised of 25 real-world programs; 22 of these programs are taken from the MiBench dataset [93], which is a representative embedded benchmark comprised of 33 programs. It is worthwhile to mention that we cannot benchmark 11 out of the 33 programs from MiBench 10 of which is due to failures and extremely slow computations in the external pointer analysis library that SROH scheme uses [5]. The 11th program (i.e. `ispell`) appears to be faulty [5]. The other three programs in our dataset are open-source games `snake` (`https://github.com/troglobit/snake`), `tetris` (`https://github.com/troglobit/tetris`) and `2048` (`https://github.com/cuadue/2048_game`) taken from GitHub.

Generating protected instances for six particular programs, namely `cjpeg`, `djpeg`, `say`, `susan`, `tetris` and `toast`, forced us to adopt a different tool. The solver that we use in our toolchain (i.e. `glpk`) takes more than 5 minutes to find solutions for some problems. It becomes problematic when programs contain a large number of conflicting manifests due to subcycles. In such cases the composition is repeated until no cycles are detected. For the mentioned programs the number of cycles (and subcycles) and thus the repetitions exceeds 1000 times, which makes it impractical to generate all binaries. To cope with this problem, in the generation of protected instances of the aforementioned programs, we utilize the commercial `Gurobi` solver (`http://www.gurobi.com/`) under a free academic license.

5.6.2. Configuration of Testbed

All performance measurements are recorded on a machine with an Intel i7-6700k processor and 32 GB of memory running the Ubuntu 18.04 LTS operating system. For all the experiments, we use constant parameters for the SC pass (*connectivity=1*) and default parameters for other protections, namely OH, SROH, CSIV and CM.

5.6.3. Coverage Optimization

In this evaluation we intend to measure the effectiveness of our ILP-based optimization on the coverage of the composition. The goal is to achieve higher coverage results with lower overheads. For this purpose, we base our analysis on two types of coverage, namely *explicit* and *implicit*. The former captures the ratio of protected instructions by the composed protections. The latter, however, represents the ratio of protected instructions, the guards of which are also protected by other guard(s). Since there exist no benchmark of the composition of our protections of interest (i.e. SC, CSIV, OH/SROH, CM), we build a baseline from our dataset. For this purpose, we use the maximum possible manifests as the baseline of our evaluation. In this mode, the composition is steered to select all manifests that yield no conflicts in the programs. We then collect the explicit and implicit coverage values achieved by the composition for every program in the the dataset. These values capture the maximum attainable coverages. Subsequently, we feed those coverage values as constraints (requirements) and set the solver to find solutions with (possibly) lower overheads.

Table 5.1 presents the experiment results; *LLVM Inst. count*, *Manifest all*, *Explicit*, and *Implicit*, *Overhead% median*, *Manifest after*, and *Decrease% median* columns denote number of program instructions in LLVM IR, maximum number of applicable manifests, explicit instruction coverage, implicit instruction coverage, median of the actual overhead (after execution), number of manifests after optimization, and percentage of the decrease over the median, respectively. It is worthwhile to mention that the implicit/explicit instruction coverage may be larger than the total number of instructions. This is due to the fact that we include protection instructions (guards) in our coverage calculations.

The five columns following the program name column capture the results of the maximum manifest protection (i.e. the baseline). The other three columns starting from the *Manifest after* capture the results corresponding to the ILP optimization aiming for minimizing the overhead while attaining the same explicit and implicit coverages as the baseline (maximum security).

Our experiments indicate that our optimization yields an overhead decrease of 38.9% on average.

5.6.4. Performance

In the performance evaluations, we are interested in measuring the performance improvements of the optimized composition as opposed to no-optimizations. One way to conduct this experiment is to use randomly selected manifests as a baseline. Our results indicate that the presented optimization technique unsurprisingly beats such a baseline. However, we believe random selection is unrealistic. In practice, compositions rather use some heuristics [5].

In order to capture possible improvements, we resort to a heuristic-based composition of SC and OH/SROH (presented in [5]) as our baseline. Our aim is to measure the overhead difference between the optimized composition versus the heuristic one. For this purpose, similar to the previous experiment, we feed their coverage values as constraints to our framework. Thereafter, we use the optimization to minimize the overhead.

To obtain comparable results, we follow their partial protection methodology in which combinations of 10%, 25%, 50%, and 100% are protected. Each coverage level indicates the percentage of program functions that must be protected by the composition framework. To weed out the noise from our measurements, for each coverage level we generate 20 random combinations of functions. Our setup yields 4000 protected binaries in total, 2000 ($= 25 \times 4 \times 20$) protected instances for the baseline and 2000 instances as the optimized-protected binaries.

To measure runtime overheads, we run each instance 100 times with the exact same input to the protected and unprotected instances of a program. For the programs in which user input is required, we pipe constant inputs by intercepting library/system calls.

	Max Manifest (all possible protection)					Min Overhead		
program	LLVM Inst. count	Manifest all	Explicit	Implicit	Overhead% median	Manifest after	Overhead% median	Decrease% median
qsort_s	92	37	143	133	117.38	29	107.29	8.59
crc	147	105	569	561	12101.88	100	11454.04	5.35
qsort_l	147	35	120	110	109.91	27	99.46	9.50
djkstra_l	323	376	1490	1480	10155.87	370	7733.33	23.85
djkstra_s	323	376	1490	1480	7424.65	370	3373.06	54.57
caudio	418	111	1578	1571	128.24	106	104.91	18.20
daudio	418	105	1556	1549	131.75	100	86.90	34.04
bm_s	532	384	1284	1216	484.69	353	190.75	60.64
tetris	629	847	761	539	861.10	280	94.41	89.04
bm_l	643	489	1395	1319	221.63	408	202.76	8.51
sha	657	1855	5752	5744	2581.26	1849	2401.35	6.97
bitcnts	664	157	1803	1793	1558.30	150	966.02	38.01
fft	742	431	2896	2885	2074.47	399	897.10	56.76
2048	749	677	3109	3094	345.51	592	288.95	16.37
srch_l	827	259	2254	2229	497.44	245	451.20	9.30
srch_s	827	259	2254	2229	203.45	245	97.69	51.98
snake	1065	1826	6726	6719	52.17	1815	49.57	4.99
patricia	1087	431	2827	2814	1290.82	297	812.04	37.09
bf	3607	280	5489	5471	1558.30	210	473.18	69.63
rijndael	5866	418	7292	7269	316.90	345	88.12	72.19
say	6859	28148	10020	7902	30536.44	2698	4593.18	84.96
susan	12656	101599	6315	5088	9634.79	1513	2933.76	69.55
toast	13930	91387	13294	11501	17598.29	2215	3344.37	81.00
djpeg	52708	4358	23055	0	11427.80	1533	8534.31	25.32
cjpeg	54837	4560	23109	0	17394.82	1507	11101.80	36.18
Mean	6430.12	9580.40	2289.26	2987.84	5152.31	710.24	2419.18	**38.90**
Median	742.00	418.00	1578.00	1793.00	1290.82	353.00	473.18	**36.18**
Std.Dev	14730.39	26786.87	1859.21	2960.85	7714.51	784.68	3546.48	28.04

Table 5.1.: Overhead decrease of optimized solutions with maximized attainable protections (explicit and implicit coverage) with minimized overhead; columns 2-6 correspond to the maximum possible protection (applicable manifests) while columns 7-9 capture the results of optimized solutions

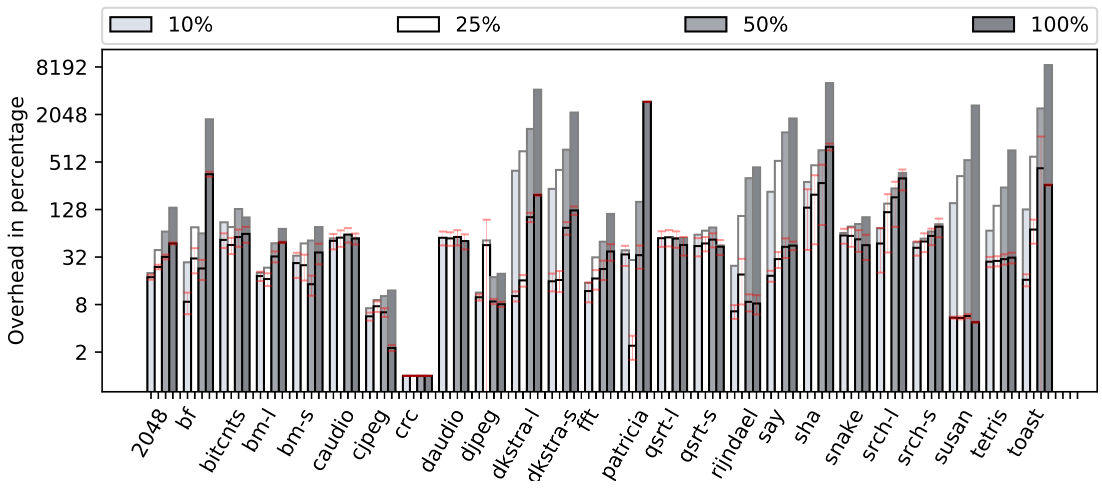


Figure 5.4.: Performance comparison of compositions of SC and OH/SROH using heuristic-based approach vs. our optimization technique for partial protections of 10%, 25%, 50%, and 100% of program instructions; the gray-margined stacked bars capture the average overhead of the heuristic-based approach, while the black-margined bars represent the same numbers corresponding to our technique

Figure 5.4 illustrates the average overhead of the composed protection for different coverage levels. The black-margined bars represent the average overhead induced by our composition optimization technique. The gray-margined bars behind the black-margined bars, represent the average overhead of the heuristic-based compositions. Our results show that our technique on average induces 31.20%, 41.77%, 68.93% and 226.98% overhead for coverage levels of 10, 25, 50 and 100% as opposed to 86.53%, 166.85%, 351.48% and 1270.96% (respectively) that is induced by the heuristic approach in [5]. Our optimization yields a five-fold decrease in the overhead in the case of 100% protection. It is noteworthy that our optimization fails to outperform the heuristic approach for some programs (or coverage levels). Such cases indicate that there simply is no room for improvements with the requested coverage constraints.

It is important to note that preventing conflicts was one of our main goals in this work. Since a majority of conflicts yield corrupted binaries, proper execution of the protected programs adds confidence to the correctness of our approach. In our experiments all the protected binaries executed correctly (i.e. no crashes or false alarms were detected) for the given set of inputs.

5.6.5. Security Analysis

To evaluate the security of the composed protections, one has two options: **i)** injecting random faults (tampering) and measuring detection rates per each scheme; and **ii)** using attack-defense tree notation to capture the steps that an attacker has to take in order to defeat the protection. The assumption in the former is that perpetrators have no knowledge about the protection measure. Subsequently, the goal is to measure what ratio of the random attacks can be detected, which is unrealistic due to 2 reasons. Firstly, attackers do possess or acquire knowledge about the protection techniques and thus random fault injection does not reflect the behavior of attackers in real life, as indicated in user studies [45]. Secondly, coverage metrics capture the percentage of protected instructions by each scheme. Simply put, we already know which instructions are left unprotected or protected with weaker protections. Therefore, one can already know which random faults are going to go unnoticed depending on the location where they are injected.

For the sake of simplicity, in our security evaluation we use a fictional program so-called `cool` that comes with a *license check (LC)* function. The goal of the attacker is to use the program without any usage limitations imposed by the `LC`. The `cool` program in various execution traces consults the `LC`. Any suspicion to violation of integrity, will be entertained with a stealthy program termination. Now let us assume that the `cool` program is protected using our protection composition framework. Figure 5.5 captures the steps that an attacker needs to successfully undertake in order to defeat the protected `LC`.

The root of the attack tree is *defeat LC*. Our attacker has 3 means to achieve this goal: i) tamper with the `LC` code, e.g. to have a license which is always valid; ii) tamper with the callsites (usages) of `LC` function such that, for instance, they call a forged function instead of `LC`; iii) subverting the program control flow to completely bypass the `LC`. All three require the attacker to first identify the `LC` function and/or its callsites in the protected program. The identification of the `LC` can be hardened by utilizing obfuscation techniques such as CM.

To tamper with the logic of `LC`, the attacker (in addition to finding `LC` in the first place) may need to defeat SC and OH guards protecting `LC`. This, however, requires to detect SC and OH guards in the program. Since SC and OH guards are very amenable, it is more difficult to tackle them one by one ([5] discusses the difficulty of this task in depth). These guards are further hardened by obfuscation techniques that are applied on top of them.

Tampering with the `LC` callsites potentially includes detection and disabling of OH/SROH/SC/CM protections in the program. The detection is further hardened by means of applied obfuscations.

To protect the integrity of the control flow, our composition utilizes CSIV protection. For an attacker to *defeat the CSIV protection*, it is first necessary to identify CSIV guards. Subsequently, the attacker needs to defeat all the potential overlapping SC/OH/SROH/CM protections.

5.7. Threats to Validity

Although we have integrated the most representative integrity protections in our framework, we may have overlooked some conflicting dependency between these protection

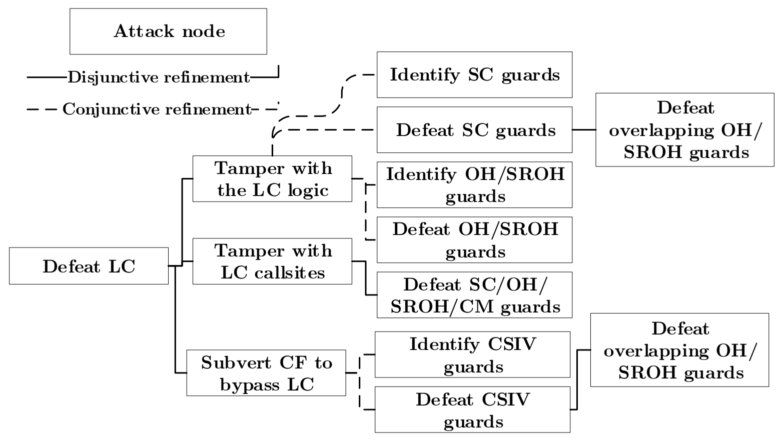


Figure 5.5.: Security analysis of the composed protection of SC, OH/SROH, and CSIV using attack tree notation

schemes. Therefore, we are only confident with the scope of the evaluated schemes. Adding further constraints may be needed for full correctness. We believe the graph-based representation is sufficient for the detection of conflicts between protection schemes.

The performance optimizations are based on the hotness of the code blocks w.r.t. the constant inputs that were passed to the programs in the dataset. Further inputs, for instance, generated using symbolic execution, can be generated for the programs in an effort to visit more representative performance profile of the program after protection.

5.8. Conclusions

Composing protections introduces layers of guards that if chosen and placed wisely can form an interconnected network of protection. This forces attackers to defeat multiple types of protection to mount their attacks, which adds more resilience to the entire protection. Conflicts amongst protection and catering for the program at hand are two problems of the software protection composition. In this work with the help of a defense graph and integer linear programming, we presented a technique that not only handles conflicts but also composes protections in a way that layered protections are formed. We further optimized the protection w.r.t. the program at hand for better performance and security. Our evaluation results on a dataset of 25 real-world programs indicate that our composition framework decreases the average overhead of selected composed protections by 38.9% while providing the maximum security (coverage). Furthermore, our approach yields a five-fold decrease on the average overhead in comparison to the relevant heuristic-based approach.

Part IV.

Software Integrity Protection Resilience Quantification

6. Resilience

In this chapter, we present a methodology along with two metrics to estimate the resilience of (composed) integrity protection techniques. The resistance is captured as a conjecture of localizability and defeatability of protections w.r.t known automated attacks.

6.1. Introduction

Software Protection encompasses a variety of protections including *integrity checking*, *obfuscation*, and *hardening* (e.g., anti-debugging [3]) measures. Depending on the use case a composition of SP protections should be utilized to mitigate potential attacks [8]. Since protections are part of the program, they are subject to attacks themselves. Perpetrators can try to localize and then defeat integrity checking routines, just like any other logic in protected programs. Applying obfuscation can **i)** introduce software diversity leading to syntactical dissimilarity amongst various usages of the same protection routines throughout the program [5]; and **ii)** impair the comprehensibility of the program and hence harden against reverse engineering attacks [17].

Intuitively, a combination of obfuscations raises the bar for a MATE more than using such code transformations individually. However, to the best of our knowledge, prior work has not determined which combination of obfuscations is suited for integrity protections, and how reliable the protection is. The impact of obfuscations in delaying localization of integrity protections via pattern matching or machine learning is unknown.

Furthermore, some attacks (e.g., taint-based) may not scale in a practical sense. It is essential to identify which obfuscations add more computational complexity for such practical attacks.

Problem. Given that integrity checking (together with obfuscations and hardening) techniques can only postpone the inevitable breach in the system, it is crucial to estimate how high the bar is set for perpetrators. Without a way to measure the resilience of protections, estimating the risk is subjective, and the claims on the security offered by integrity protection are merely speculations.

Research questions. RQ1: How can we estimate the resilience of integrity protection measures? **RQ2:** How do obfuscation and other hardening measures impact the resilience?

Challenge. The MATE's ingenuity coupled with unlimited access to the host on which the program is executed, hinder a complete enumeration of all possible attacks.

Idea. In this work we confine the analysis to a catalog of *automated* attacks and their corresponding defenses, similar to [23]. We believe the degree of defenses' effectiveness against automated attacks can capture a first-order proxy quantifying resilience. Since any such catalog will be incomplete in the beginning, we open-source our catalogs and enable extensions through pull requests from the community. Nonetheless, the automated attacks and defenses currently in our catalogs represent the state-of-the-art.

Gaps. Section 6.2 captures related work. We identified the following gaps in the literature:

- No previous study was done on *how* resistant state-of-the-art integrity protections are against automated attacks (i.e. pattern-based, machine-learning-based, and taint-based attacks);
- The impact of obfuscation on the resilience of integrity protections was not previously studied;
- There exists no methodology to evaluate the resilience of composed protections at the function level and at the program level.

Contributions. This chapter closes the above gaps by:

- proposing an empirical evaluation methodology of resilience, agnostic to the techniques used to protect programs and the automated attacks applied to such programs;
- proposing a machine-learning-based measure to localize integrity protections even when combined with obfuscations;
- proposing a metric (Estimated Patch Steps) to capture the effort needed for (a first-order proxy of) patching protections;
- proposing a resilience metric as the fusion of localizability and defeatability of protections;
- empirically evaluating the effectiveness of state-of-the-art attacks (viz. pattern-based, machine-learning-based, and taint-based) in localizing protections on the MiBench data set;

Section 6.3 discusses the methodology. In Sections 6.4 to 6.6 we present attacks, defenses, and metrics of our methodology. Section 6.7 captures our evaluations. Section 6.8 discusses our results. Section 6.9 describes threats to validity. Section 6.10 presents conclusions and ideas for future work.

6.2. Background and Related Work

6.2.1. Attacker Model

We assume that our attackers possess full knowledge about the internals of the potentially utilized protections, but we believe they do not necessarily know which ones in which order are used. Assuming that attackers do not possess knowledge about the internals of protections would mean that we rely on security by obscurity.

6.2.2. Protection Quantification and Metrics

Schrittwieser et al. [160] conducted a survey to study the effectiveness of obfuscations against published de-obfuscation attacks. Their survey gathers and classifies attacks on obfuscations into four categories, namely pattern matching, automated static and dynamic analysis, and human-assisted analysis, that are utilized to locate code fragments in obfuscated programs. Thereafter, they analytically estimate the strength of obfuscation techniques w.r.t. the gathered attacks.

Ahmadvand et al. presented two coverage metrics, *explicit* and *implicit* coverage, which are used in selecting integrity protections in their composition framework called *sip-shaker* [8]. Explicit coverage refers to the ratio of program instructions that are directly protected. Implicit coverage captures the ratio of instructions whose guards are protected by other guards, i.e., they are indirectly protected by guards that protect their immediate protections. Their framework and benchmarks focus on coverage metrics as the security criteria.

Ghosh et al. [88], Chang and Atallah [47] used the *connectivity* metric as a way to tweak their protections to reconcile security and overhead. Myles and Jin [138] proposed a set of metrics to quantify the capability of integrity protections that are primarily built upon coverage metrics. We use similar coverage metrics in our evaluations. However, coverage metrics on their own do not capture the resilience of protections, as some attacks [150, 183] are not affected by the protection coverage.

Kanzaki et al. [107] developed a technique to identify obfuscated code fragments in a given program. Their technique can potentially be used as a metric to flag poorly obfuscated code. In this work, we obfuscate the entire program (not only the protected fragments), and thus identifying obfuscated fragments does not help the perpetrators in detecting protections. Banescu et al. [20] formulated deobfuscation as a search problem for which symbolic execution is utilized to reach to particular obfuscated statements in the program. The time needed by symbolic execution to successfully attack programs was then used as a metric to quantify the resilience of obfuscation techniques against automated attacks.

The presented techniques from the literature aim to quantify the resilience of software protections by measuring the offered security. We believe such metrics do contribute to estimating the security of software protections but fail to capture the actual resilience against automated attacks on integrity protections.

6.2.3. Empirical Evaluations

Ceccato et al. [45, 46] presented an empirical study on how attackers (professional as well as amateur pen testers) conduct their attacks on real programs. Their work sheds light on the steps that attackers take to analyze, identify, and craft necessary tools, as well as circumvent and undo protections. In this work, we take a different perspective. Instead of human attackers, we aim at characterizing protections by empirically studying the resilience to known automated attacks.

Garba and Favaro [84] developed a framework to de-obfuscate (simplify) obfuscated code using compiler optimizations in combination with a SMT solver. Their benchmarks indicate compelling results in the area of de-obfuscation. Their work can be seen as a potential extension to our attack catalog.

Salem and Banescu [154] utilized machine learning to classify obfuscation techniques. In their technique, learners pick up the patterns of four obfuscation techniques (control flow flattening, instruction substitution, bogus control flow, and virtualization) in training and later effectively detect them in programs. Similarly, Tofighi-Shirazi et al. developed a machine learning-based opaque predicate obfuscation detection [63, 175]. Ben-Nun et al., Chen and Monperrus show promising techniques for source code embedding [28, 52]. However, our attackers of interest (MATE) do not have access to the source code of programs but their binaries. Therefore, we do not use these embeddings in our experiments.

To capture the relation between obfuscations and attacks, Kelly et al. [111] measured the effectiveness of six diversification obfuscations against two classes of remote attacker exploits (e.g., buffer overflows) provided in the Cyber Grand Challenge (`https://www.darpa.mil/program/cyber-grand-challenge`). They obfuscated the programs given in the challenge and studied which combinations of obfuscations generate less exploitable programs than others. Their attacker model does not match our MATE attacker model.

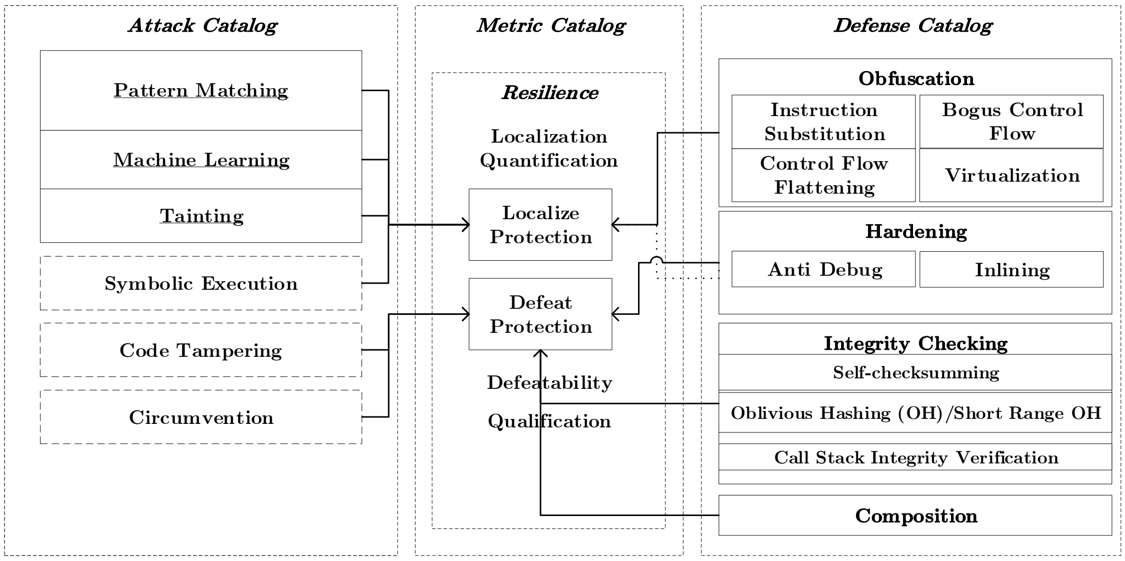
Ding et al. [75] proposed a technique to detect code clones using the lexical-semantic relationships that tends to learn robust patterns of semantically equivalent code, despite obfuscations and optimizations, in the assembly representation.

To the best of our knowledge, no previous studies address all our identified gaps. However, they provide puzzle pieces as to how the resilience of obfuscations should be measured. The presented related work inspires our experiments, but in addition to related work, we provide new attacks, metrics, and experimental results.

6.3. Methodology

Our goal is to characterize the resilience of state-of-the-art integrity protection techniques in combination with other SP techniques against known automated attacks, in order to facilitate selecting the strongest protection combination for a given protection target. We consider two roles in our methodology: *attacker* and *defender*. The defender aims at countering, impeding, or delaying attacks. To quantify resilience, we refine the battle between the defender and the attacker into two hypothetically disjoint fronts, *localize* and *defeat*. In the *localization battle*, the attacker wants to distinguish the protection logic from program logic. The defender employs countermeasures to make the detection as hard as possible. The *defeat battle* generally requires the *localization battle* to be successful. The attacker actively changes the behavior of the program, e.g., by patching the code. A patch is defined as a sequence of bytes that are overwritten by the MATE in order to neutralize exactly one integrity check (guard) and maintain the same input-output behaviour of the original program. The defender attempts to raise the bar by introducing layers of protection (e.g., cyclic checks) to checkers and the response mechanisms. Through our methodology, we enable the defender to estimate the resilience of their protections. Figure 6.1 depicts our evaluation methodology. The left box in the figure captures our catalog of automated attacks. The right box depicts the defense catalog. The middle box shows the metrics used to measure the effectiveness of attacks (localizability and defeatability).

Figure 6.1.: Integrity protection resilience evaluation methodology encompassing attack, defense, and metric catalogs along with the their relation to the localize and defeat battlefronts; the attack categories containing automated attacks are underlined



We consider three *living* catalogs in our methodology: *attack*, *defense*, and *metric*. These catalogs are meant to be maintained and extended as new attacks, defenses, and metrics are discovered. The attack catalog captures automated techniques that the attacker utilizes in either or both battles. The defense catalog includes measures that the defender applies to resist attacks. The metric catalog provides means to measure the effectiveness of attacks vs. defenses.

We empirically benchmark the effectiveness of different permutations of attacks and defenses in the catalog, on a representative set of programs. Our ultimate goal is to characterize the resilience as a conjecture of two values: *localizability* and *defeatability*.

6.3.1. Localizability

We define localizability as the probability of successfully identifying integrity checking routines in programs. Localizability also is a property of protections and combinations of protections (i.e., integrity checking scheme together with obfuscation permutations). Our idea is that localizability w.r.t. certain attacks can be pre-computed for a representative set of programs. Henceforth, these values can be generalized and subsequently reused to estimate the localizability of the same protections in other programs that are comparable to the representative programs. Simply put, identical compositions of integrity checking techniques with identical combinations of obfuscations are expected to have comparable localizability scores and *are less dependent of the individual program they protect.*

6.3.2. Defeatability

We define the generic term of "defeatability" as the effort needed to break protections. Specifically, we are after quantifying two aspects: **i)** the number of patches that the attacker has to perform to defeat protections successfully; and **ii)** the effort needed for each of the patches. While we can quantify the number of required patches, the involved effort remains hard to quantify objectively. It is challenging to automate defeat attacks without committing to unrealistic assumptions. A minor tweak of the code can render automated patching attacks defeated. Factors such as the type of response mechanism(s), the degree of intertwinement of the response(s) with the program logic, attackers' skills, the size of programs, and the employed hardening techniques (e.g., emulation detections, code packing, and debug/trace preventions) can significantly affect the difficulty of patches. Most of these factors (particularly response mechanisms) are very use-case specific. We consider quantifying such defeatability factors, despite their pertinence, beyond the reach of this thesis. Therefore, we settle for the number of patches as a first-order approximation of defeatability.

The number of required patches is affected by the applicability of protections on different segments of a particular program. Some protections cannot protect certain parts of programs, e.g., oblivious hashing can only be applied to input independent instructions [5]. Specific applications of protections have a direct impact on the formation of layers of protections of different types, as well as their overlaps. Such layers and overlapping protections affect the number of required patches. Multi-layered and cross-checking protections are expected to have higher defeatability scores. In this work, *higher defeatability* scores indicate a higher number of required patches and thus *more resilience* against MATE attackers.

6.3.3. Methodology Summary

To sum up, our methodology for resilience quantification takes the defenders' perspective. The method includes the use of large sample data sets for computing localizability scores that are less dependent of a program P that we want to protect. We suggest updating these scores when new sample programs are available, when there are further protection or hardening mechanisms, and when there are new attacks. The method then suggests updating defeatability scores for P.

6.4. Attacks

The attack catalog comprises six categories. When possible, we develop automated attacks for each of these categories. To honor the attacker model in Section 6.2.1, we use our knowledge about the internals of protections in the development of attack instantiations. Simply put, our attacks target potential shortcomings of protections.

6.4.1. Pattern Matching

Attackers can use recognizable patterns in protected programs to detect integrity checking guards (or responses). At first, we assume integrity checking techniques do not utilize obfuscations. With this assumption in mind, attackers can develop detection patterns for every integrity checking technique in our defense catalog. They can use syntactical as well as structural features to detect protection signatures. However, to mimic a realistic attack environment, in our benchmarks, no hardcoded names nor addresses will be used as signatures. We provide further details of these attacks in Section 6.7.2.

6.4.2. Machine Learning

Obfuscation is an immediate remedy to pattern-based attacks. It makes pattern identification harder. Identifying protection patterns and subsequently handcrafting matcher for them fail to scale, given the number of possible obfuscation permutations. Machine Learning is a tool used in the literature to automate the pattern identification on obfuscated code [75, 154].

We develop a machine-learning-based attack for detecting integrity checking schemes hardened with obfuscations as an instantiation of this category of attacks. Our goal is to localize integrity checking guards at the granularity of basic blocks in protected programs. To this end, we fuse three machine learning approaches: term frequency-inverse document frequency (TFIDF) [155] and neural networks on syntactical features; the GraphSage algorithm [95] for the structural features. We provide further details in the evaluation section.

6.4.3. Tainting

Analyzing relations amongst instructions can help identify structural patterns in integrity protections [9]. For instance, the response mechanism is usually triggered when a specific variable is set in the program [150]. Performing taint analysis can potentially reveal instructions contributing to tamper detections. For this category of attacks, we instantiate the attack proposed by Qiu et al. [150]. The attack targets the SC protection. It relies on program traces on which a set of forwarding and backward taint analyses are undertaken, leading to the localization of SC checks. We utilize (develop) the exact attack to measure the resilience of the SC protection (with other defenses) against taint-based attacks. Since this attack operates on the program traces, we study the impact of two trace polluting obfuscations (namely, control flow flattening and virtualization) on the attack effectiveness.

6.4.4. Symbolic Execution

Attackers can use symbolic execution to lift code to a higher abstraction on which localization attacks (e.g., pattern matching) can be more effective as it abstracts away from syntactic differences [116, 152, 156, 188]. A good candidate for this class of attacks is the symbolic de-virtualization technique proposed by Salwan et al. [156]. However, due to difficulties in the symbolic execution of system calls used by the programs in our data sets, we were unable to use their tool. We believe employing symbolic attacks is independent of the rest of the attacks. That is, attackers can first use symbolic execution and subsequently proceed with attacks on integrity protection. Hence, we exclude symbolic attacks from our empirical evaluations.

6.4.5. Code Tampering

This category refers to the actual patching attacks. As pointed out earlier, we do not attempt to automate patching attacks as simple code tweaks can break such attacks. In this work, we consider such attacks to be done manually.

6.4.6. Circumvention

This class of attacks can be seen as the side-channel equivalent class of attacks against software-based protection. There exist some attacks (more might be discovered) that can potentially defeat protections with no localization whatsoever of their guards [183]. There is not much we could do in terms of resilience characterization for such attacks. The mitigation of such techniques typically includes employing another layer of protection (see how [131] mitigates [183]). We urge defenders to beware of such attacks and to seek measures to mitigate them continuously.

6.5. Defenses

6.5.1. Integrity Checking

We include 4 representative schemes in our analysis, namely self-checksumming (SC) [22, 47], oblivious hashing (OH) [49], short range oblivious hashing (SROH) [5], and call stack integrity verification (CSIV) [8, 19]. We always apply OH and SROH together for better protections and refer to them as one scheme OH/SROH in the remainder of this chapter.

6.5.2. Obfuscation

We instantiate four obfuscation transformations for the obfuscation category of the defense catalog. For the first three techniques, namely instruction substitution (IS), bogus control flow (BC), and control flow flattening (CFF), we use the implementation provided by Junod et al. [104]. In the case of virtualization (Virt), we use the transformation provided in [7].

IS

is a technique in which particular instruction sequences are replaced with relatively complicated yet semantically equivalent sequences.

BC

reduces the comprehensibility of programs by introducing structural complexity in the form of adding bogus blocks. The program's control flow remains the same as the unobfuscated version by preventing the control from taking bogus paths using opaque predicates.

CFF

also aims at introducing structural complexity (and diversity) by unfolding nested branching conditions into a single conditional branch.

Virt

lifts program instructions into a random instruction set architecture (ISA) execution of which is only made possible via a specially tailored interpreter. Virtualization introduces structural as well as functional complexity in the obfuscated programs.

6.5.3. Hardening Transformation

Defenders typically use further hardening measures such as anti-debugging [3], dynamic code encryption [15, 87, 180], anti-symbolic [20, 143], runtime packing, etc. Hardening transformations can increase the difficulty of both localization and defeat attacks. Hardening measures' effectiveness can be evaluated similarly to the obfuscations. Benchmarking hardening measures falls beyond the focus of this thesis.

6.6. Metrics

We instantiate three metrics, one per category plus one combined metric for capturing resilience. We interpret localizability as a property of the applied protection mechanisms—and not the individual program under consideration—that can be empirically determined using existing data sets. Defeatability is a property of the individual program under consideration because, in general, there will be varying numbers of protections of relevant fragments of the program.

6.6.1. Localization Quantification

The *f-measure* captures the detection accuracy by means of a harmonic average of precision and recall. Under the assumption that recognizable patterns proportionally increase the localizability, the ratio of protections with identifiable patterns essentially yields localizability of protections. Therefore, we resort to the *f-measure* metric to quantify the recognizability of protection patterns in protected programs. This localization metric is dependent on, and hence relative to, the considered attack techniques.

6.6.2. Defeatability Quantification

In estimating defeatability, it is important to understand that the defeatability may differ for different segments of the same program. Intuitively, a function protected by a range of different protections exhibits higher defeatability (i.e., harder to defeat) than a function with only one or no guards. Therefore, we need to quantify the defeatability for every sensitive segment in the program individually, which we will denote by segmental defeatability, and subsequently aggregate them to capture a collective defeatability score.

We pick the function level as the smallest unit (segment) in the program for which we calculate a defeatability score. The function defeatability is calculated based on the number of protections, their types, and the order in which they are applied to a program.

Estimated Patch Steps (EPS). Attackers are required to disable tamper detections or their response (a.k.a. reaction) to tamper with the original program code or data. The required steps depend on how protections are chained together and hence shall be computed for every program individually. *Estimated Patch Steps (EPS)* captures the number of alteration steps that attackers need to take to achieve their goal. EPS corresponds to the number of guards or responses that need to be circumvented. That is, if another guard protects the guard protecting a segment, the EPS number shall capture this.

Remember that the effort needed in each attack step entirely depends on the difficulty of disabling the targeted guards or their response mechanisms. To calculate the EPS for a given program, we use the defense graph [8] in which nodes correspond to the protection guards and protected segments. At the same time, edges indicate protection relations between guards and program segments. For estimating defeatability, we need to compute the number of distinct edges that directly or indirectly contribute to the protection of each guard and each original program code fragment. Direct contributions can be translated to all edges from immediate adjacent nodes of a guard. Indirect contributions refer to any edge that is on a path to any edge with direct contribution to the protection of a particular node. We define $EPS(u)$ as $\sum_{u \neq v} |d(u,v)|$, where u corresponds to a protection guard while $d(u,v)$ is the set of (acyclic) paths (edges) between u and v nodes computed using the *Dijkstra* algorithm (d).

Weighted EPS. Although EPS captures the number of steps, the actual difficulty of a defeat largely depends on the difficulty of disabling the employed response mechanisms. That difficulty varies considerably from one response mechanism to another. A simple program termination response can be defeated more easily than stealthy stack manipulation. Therefore, it is beneficial to let defenders estimate the difficulty of disabling their use of case-specific response mechanisms a priori. This estimation is done by assigning a value between 1 and ∞; higher values indicate greater difficulty.

To take into account the effort, we assign weights to edges according to the difficulty of defeat. For example, with the weighted metric experts may weigh a particular implementation of *SC* guards lower than a variation of response utilized in *OH* protections. We assign a default weight of 1 to all protection guards with no explicit weight assignments. Bear in mind that edge weights $d(u, v)$ are intrinsically taken into account in the path computations.

Once the EPS value of each node (guard) is computed, we need to map the nodes' EPSs to a desired granularity level as segments. Remember that we consider functions as the granularity of interest. Assuming acyclic dependencies only, let $EPS(f) = \sum_{u \in p(f)} EPS(u)$, where f is the function of interest; N is the set of all protection nodes; $p(f) = \{x \in N \mid target(x) = f\}$ where $target(x)$ returns the function of the segment that is protected by node x.

6.6.3. Resilience

To capture resilience at the program level, we need to aggregate localizability and defeatability of functions (segments). Due to the diversity of obfuscated protections, the localizabilities (probabilities) of functions are roughly independent of each other. Therefore, we multiply the localizability of functions for an aggregate localizability value. In the case of defeatability, we compute an average over functions' scores. We define the resilience score of a program π as a pair consisting of the product f-measure for localizability and the average EPS defeatability: $res(\pi) = \langle \prod_{f \in \pi} loc(f), \frac{1}{|\{f|f \in \pi\}|} \cdot \sum_{f \in \pi} EPS(f) \rangle$, where loc returns a localizability f-score of the given function based on the applied obfuscations. Bear in mind that localizability for different permutations of defenses is *calculated* once for the representative programs in our data set. From the *defender's* perspective, we now want to estimate the localizability of *one concrete program* π. Because we take the defender's perspective, we know which obfuscations were applied to which function, and can thus look up the respective localizability scores in a table that we provide in Section 6.7. Localizability values only need to be updated and recomputed as new defenses, or attacks are discovered. However, the EPSs need to be computed for each program individually.

6.7. Evaluation

IIn this section, we undertake a set of experiments to evaluate the effectiveness of attacks against a combination of defenses.

6.7.1. Test Programs

We use a subset of 22 programs from the MiBench data set [93], the simple data set [154] containing 40 programs, and three open-source CLI games, namely tetris, snake, and 2048, totaling 65 programs. In the remainder of this chapter, whenever we refer to the MiBench data set, the three games are included as well. We were unable to use all the samples from the MiBench data set due to failures in the external analysis tools that are required by OH/SROH protection [5].

6.7.2. Integrity Checking Localizability

Our objective here is to quantify the ease of localizing protections. Previously, we described two attacks, namely pattern-based attacks, and machine-learning-based attacks (see Section 6.4) that adversaries can potentially mount on protections in order to localize them. Since our methodology is from the perspective of the defender, we use LLVM IR as the representation on which we conduct our evaluations. The availability of precise static analysis in LLVM is the reason behind our decision to use IR instead of the assembly representation. In reality attackers might be able to retrieve (parts of) the IR using tools such as *remill* (`github.com/lifting-bits/remill`), *mcsema* (`github.com/trailofbits/mcsema`), and *reopt* (`github.com/GaloisInc/reopt`).

To measure the effectiveness of localization attacks, we resort to the f-measure metric (see Section 6.6.1). From our experiments, we noticed that compositions of integrity checking techniques do not impact the localization of these techniques. That is, composing integrity checking techniques did not cause any significant difference in the *detection effectiveness* of their guards in protected programs. Therefore, we do not compose different integrity checking techniques in the pattern-based, as well as machine-learning-based attacks evaluations presented in the following subsections.

Pattern-Based

We develop signature-based attacks against SC, OH/SROH, and CSIV. To simulate a real-world environment, we do not use variable names or function names. This is mainly because a simple obfuscation pass can break such detection techniques rendering our pattern-based attack defeated. We mainly use instruction sequences that are commonly used by protections.

Pattern-Matching Data set. For this experiment, we generate three protected instances (SC, OH/SROH, and CSIV) of each program in the test set amounting to 195 programs.

Attacks. We are able to develop pattern matchings for all the protections due to their obvious footprints in the programs. We mount the pattern attacks on the pattern matching data set and subsequently compute the average localization effectiveness by means of the f-measure metric for each protection. Our experiment indicates an f-measure of 100%, 93.5%, and 100% for SC, OH/SROH, and CSIV, respectively.

Machine Learning

In this experiment, our objective is to estimate the localizability of integrity checking techniques when used in combination with obfuscations. We use protected programs in LLVM IR bitcode format as input artifacts to the machine learning attack. This experiment includes SC, OH/SROH, and CSIV protections.

Machine Learning Data sets. For this purpose, we experiment with the simple and MiBench data sets separately. The reason that we do not combine the sets is to study the impact of data set size on the effectiveness of classifications. We generate distinct protected-obfuscated data sets, one per each order-sensitive permutation of obfuscations, namely IS, BC, and CFF, yielding 16 combinations (including no obfuscations, i.e., None). Throughout our experiments, we realized that the default obfuscation configuration fails to substantially affect ML classifications in the case of MiBench. Consequently, we extend our obfuscation permutations with more aggressive configurations for the MiBench data set. IS and CFF can be applied multiple times according to an input parameter (the default value is one). In the case of BC, a probability value between 1-100 (the default is 30) can be set via an input parameter. We benchmark values between 2-10 for IS and CFF obfuscations, and 40, 60, 100 for BC. However, due to space limitations, we only report on 25 configurations with the highest impact on the effectiveness values amounting to 41 (16+25) distinct protected-obfuscated data sets of MiBench.

Each uniquely obfuscated data set contains (obfuscated) basic block samples of all programs (in the MiBench or simple data set) with four flavors of protections: No-Protection, SC, OH/SROH, and CSIV. It is important to reiterate that integrity checking logics are rather constant regardless of the program to which they are applied. Obfuscations are the primary source of diversity in the samples. The left column of the two tables in Table 6.1 and Table 6.2 capture the number of samples (DS) for the MiBench and simple data sets. The obfuscation parameters are denoted as subscript values.

Feature Engineering. We incorporate the syntactical and structural properties of the programs into our features. Syntactical patterns refer to the syntactic resemblance of instructions corresponding to the protection of interest. Structural patterns refer to similar constructs (i.e., call-graph, control flow, and data flow) of two distinct programs protected with the same set of protections.

For the syntactical features, we utilize TFIDF on the LLVM IR representation of program basic blocks. Before generating the corpus, documents (IR of binaries) undergo a cleaning process in which constant values, variable and register identifiers, and alignments are removed. The TFIDF features are then mapped on the corresponding basic blocks. Each basic block is assigned with two labels based on the integrity checking measure it contains along with a concatenation of utilized obfuscations.

SC guards reside in one logical basic block. However, OH, SROH, and CSIV guards will be split into multiple basic blocks. Preceding blocks undertake *hashing* into the hash variables of SROH/OH guards or the shadow stack variable of CSIV guards. Nevertheless, one block will be in charge of matching the calculated values against the expected values.

We consider the block with the branching condition as the target of our localization.

As per structural features, we collect an ensemble of control flow graph (CFG), interprocedural (explicit and implicit) dependency [39], and strongly connected components (SCC) [142] relations. Each block obtains a unique identifier and gets translated to a graph node. Syntactical features (TFIDF values) are mapped on the nodes, too. Structural features are then introduced as relations amongst the nodes (blocks). We run LLVM's native static analyses on the protected programs to capture CFG and SCC relations. Each relation amongst the blocks is translated to an arc between the respective nodes on the graph. For dependencies, we use the SVF tool to capture instructions' data and control flow relations [168]. Instructions' relations are then lifted to their blocks. That is, if two instructions from two disjoint blocks have a dependency relation, we lift the relation to their blocks and subsequently to their corresponding nodes on the graph.

Configuration. We use distinct classifiers for each data set of programs. This setting entails that attackers need to use the corresponding classifier to the obfuscation permutations that are utilized by the protected programs. From a practical perspective, we assume that attackers can use several trained models to localize protections. Alternatively, attackers can first predict the obfuscations applied in the binary (e.g., using techniques similar to [154]) and subsequently use the corresponding model to localize checkers.

We use the *StellarGraph* library and set the number of epochs to 40 [70]. Our classifiers are set to optimize for maximum categorical accuracy and a minimum loss (categorical cross-entropy). We train and test our classifiers in 10-fold cross-validation in which we use 80% of blocks for training, and 20% for validation measurements.

Results. Table 6.1 illustrates the effectiveness of our classifiers on the MiBench data set for each integrity checking technique per obfuscation configurations. The results of No-Protection classifiers, which are omitted due to space limitations, indicate an average f-measure of 99.19% (σ=0.16%). Table 6.2 presents the localization results corresponding to the simple data set. The results confirm that particular configurations of obfuscations cause more difficulties than others for the learners. When the number of samples is small (in the case of the simple data set), ML performs worse than comparable configurations with more samples. For example, BC+IS+CFF brings the f-measure of SC down to 10.71% for the simple data set (Table 6.2), while the same value is 82.54% for MiBench (Table 6.1). More importantly, supported by our experiments, we can favor configurations with lower f-measures over others. Our results indicate that BC($_{100}$)+IS+CFF, BC($_{100}$)+IS($_{2}$)+CFF($_{2}$), and BC+IS+CFF best obstruct SC, SROH/OH, and CSIV classifications.

Taint-Based Attacks

In the case of SC protection, attackers can also use the taint-based attack to localize guards [150]. Here our objective is to measure the effectiveness of the taint attack in terms of detection rate (f-measure) as well as scalability. Since this attack operates on the program traces, we experiment with the impact of trace pollutions (i.e., trace size) on the attack time (scalability). As mentioned previously, we use CFF and Virt obfuscations as

Table 6.1.: ML-based attack effectiveness in localizing integrity checking measures; the subscript values (2, 40, and 100) indicate the parameter passed to the corresponding transformations; *DS*, *FM*, and σ denote the number of (basic block) samples in the data set, the average f-measure over the 10-fold validations, and the standard deviations of different folds, respectively

		SC		***OH/SROH***		***CSIV***	
Obfuscation	***DS***	***FM***	σ	***FM***	σ	***FM***	σ
None	95112	99.99%	0.02%	98.57%	0.80%	97.94%	2.31%
BC	266365	99.88%	0.09%	88.00%	5.35%	95.95%	3.74%
BC+CFF	454834	80.03%	9.21%	32.27%	18.25%	19.11%	19.99%
BC+CFF+IS	454944	87.32%	4.05%	38.08%	13.50%	12.29%	6.00%
BC+CFF$_2$	454126	85.30%	8.07%	43.15%	15.73%	21.71%	17.20%
BC+CFF$_2$+IS$_2$	455252	84.29%	8.27%	33.69%	14.96%	15.49%	13.52%
BC+IS	267236	99.84%	0.11%	83.58%	8.67%	96.12%	3.36%
BC+IS+CFF	453426	82.54%	8.68%	32.33%	18.85%	8.48%	8.99%
BC+IS$_2$+CFF$_2$	455475	82.05%	6.19%	**6.26%**	6.33%	**0.84%**	0.62%
BC$_{100}$	646318	99.87%	0.19%	82.53%	13.32%	97.37%	3.41%
BC$_{100}$+CFF	895135	76.38%	9.91%	7.26%	8.48%	16.19%	11.67%
BC$_{100}$+CFF+IS	895135	82.39%	4.78%	9.95%	11.17%	13.91%	15.74%
BC$_{100}$+IS	646318	99.91%	0.10%	86.82%	6.88%	97.44%	3.03%
BC$_{100}$+IS+CFF	895135	**64.28%**	13.30%	7.20%	7.10%	12.62%	10.46%
BC$_{40}$	322074	99.83%	0.13%	88.10%	5.85%	96.29%	3.39%
BC$_{40}$+CFF	518117	85.08%	6.77%	33.61%	17.76%	15.89%	11.03%
BC$_{40}$+CFF+IS	518063	87.01%	6.80%	28.03%	10.08%	15.72%	13.93%
BC$_{40}$+IS	321260	99.67%	0.27%	84.80%	8.22%	95.84%	4.39%
BC$_{40}$+IS+CFF	517582	79.01%	8.26%	23.35%	17.45%	13.71%	13.74%
CFF	255851	95.62%	2.09%	64.22%	17.13%	62.36%	10.00%
CFF+BC	730627	95.02%	3.39%	74.25%	8.12%	64.04%	8.10%
CFF+BC+IS	726230	91.15%	6.79%	68.89%	11.83%	60.66%	5.83%
CFF+BC$_{100}$	1783514	95.69%	2.06%	87.34%	6.55%	83.13%	5.43%
CFF+BC$_{100}$+IS	1783514	96.90%	1.59%	86.06%	5.86%	84.70%	4.78%
CFF+BC$_{40}$	882526	94.80%	2.24%	71.59%	8.89%	67.37%	7.79%
CFF+BC$_{40}$+IS	877958	95.17%	1.23%	72.30%	7.45%	67.49%	7.68%
CFF+IS	255851	95.72%	4.21%	64.45%	13.27%	44.19%	23.51%
CFF+IS+BC	728632	92.42%	5.23%	73.34%	9.13%	67.13%	5.21%
CFF+IS+BC$_{100}$	1783514	96.50%	2.35%	82.16%	8.39%	81.49%	7.54%
CFF+IS+BC$_{40}$	880624	95.07%	2.12%	72.49%	12.18%	68.82%	5.46%
IS	95112	99.97%	0.07%	98.72%	0.73%	98.72%	1.15%
IS+BC	266290	99.89%	0.09%	85.41%	5.23%	94.48%	4.96%
IS+BC+CFF	455058	90.07%	3.80%	34.55%	13.74%	18.06%	11.68%
IS+BC$_{100}$	646318	99.82%	0.26%	86.62%	5.62%	98.20%	1.83%
IS+BC$_{100}$+CFF	895135	71.90%	12.82%	7.48%	5.05%	15.84%	16.47%
IS+BC$_{40}$	320527	99.81%	0.14%	86.34%	6.73%	95.90%	4.06%
IS+BC$_{40}$+CFF	517787	83.79%	12.69%	35.37%	13.49%	18.54%	14.09%
IS+CFF	255851	95.83%	2.37%	69.20%	12.39%	49.56%	21.44%
IS+CFF+BC	730069	94.01%	3.03%	65.73%	13.72%	64.20%	7.99%
IS+CFF+BC$_{100}$	1783514	94.74%	3.37%	82.35%	9.11%	81.21%	5.65%
IS+CFF+BC$_{40}$	884683	93.52%	4.21%	62.63%	16.16%	66.84%	7.73%

Table 6.2.: Machine-learning-based attack effectiveness in localizing SC, OH/SROH, and CSIV on different combinations of IS, BC, and CFF obfuscations for the simple data set; *DS* denotes the number of samples in the data set; *FM* captures the average f-measure over the 10-fold validations; σ denotes the standard deviations of different folds

			Integrity Checking Measures							
			No-Protection		***SC***		***OH/SROH***		***CSIV***	
		DS	***FM***	σ	***FM***	σ	***FM***	σ	***FM***	σ
Obfuscation Compositions	None	10808	99.72%	0.15%	99.94%	0.18%	96.91%	2.43%	97.19%	3.14%
	BC	30166	99.41%	0.20%	99.26%	0.73%	81.75%	5.35%	93.34%	4.72%
	BC+CFF	52443	97.92%	0.09%	15.31%	15.47%	19.86%	10.72%	7.84%	0.00%
	BC+CFF+IS	51866	97.96%	0.09%	26.36%	16.40%	18.66%	12.33%	**nan**	nan
	BC+IS	30419	99.32%	0.21%	98.62%	0.86%	80.29%	6.45%	90.56%	5.91%
	BC+IS+CFF	51718	97.87%	0.05%	**10.71**%	10.49%	**nan**	nan	18.18%	0.00%
	CFF	29468	98.13%	0.47%	69.57%	28.21%	36.47%	9.23%	**nan**	nan
	CFF+BC	83319	98.98%	0.18%	49.42%	25.03%	50.59%	10.83%	21.48%	9.41%
	CFF+BC+IS	84362	98.96%	0.20%	49.14%	23.02%	47.81%	19.00%	10.84%	7.65%
	CFF+IS	29468	98.22%	0.46%	72.70%	25.74%	45.02%	17.76%	**nan**	nan
	CFF+IS+BC	83462	98.99%	0.11%	49.27%	21.11%	51.21%	10.12%	16.42%	9.21%
	IS	10808	99.68%	0.27%	99.76%	0.54%	97.21%	1.53%	95.36%	8.89%
	IS+BC	30254	99.39%	0.23%	99.41%	0.70%	82.28%	6.92%	90.72%	7.01%
	IS+BC+CFF	51849	97.86%	0.15%	13.39%	13.15%	**10.09**%	0.72%	**nan**	nan
	IS+CFF	29468	98.02%	0.27%	69.54%	20.99%	43.95%	13.22%	**2.70%**	2.70%
	IS+CFF+BC	84526	99.03%	0.15%	52.32%	22.98%	49.44%	16.09%	11.53%	6.87%

trace polluting measures due to their impact on the trace size.

Taint Data Set. For the sake of this experiment, we generate two sets of protected programs: SC-protected, SC-protected-CFF-trace-polluted, and SC-protected-Virt-trace-polluted binaries from the test program set. We limit the application of both of the CFF and Virt to SC checks and 20% of other program functions. Note that the larger application of these techniques leads to extremely large traces for which we are unable to perform the taint analysis. We use DynamoRIO to capture one trace for each program in the set with constant inputs. Subsequently, we use Triton [71] to mount the taint-based attack introduced in Section 6.4 on the captured traces of the MiBench data set programs. In our experiment, we limit the trace size $\approx$ 370 GB and set the analysis time to 80 hours.

Results. We measure the trace size and attack time of the experiment for each program separately. Table 6.3 presents the results of the taint-based attack. According to our results, the attack successfully localizes all SC checks (FM=100%) for both Vanilla SC and SC+CFF20 configurations. Clearly, CFF does not mitigate the risk (except for the basicmath_l program). Nonetheless, attack times are significantly increased, which could potentially render the attack impractical for large programs. The attack fails to properly localize protection in many cases when virtualization is applied (Section 6.8.1 elaborates further).

Table 6.3.: Taint-based localization effectiveness experiment; Insts., TS, AT, CFF20, Virt20 denote the number of LLVM instructions in the original programs (unprotected), the trace size (MB), the attack time in seconds, CFF on SC checks and 20% of other functions, and Virt on SC checks and 20% of other functions, respectively; N/A indicate failure of the attack or analysis

		Vanilla SC		*SC+CFF20*		*SC+Virt20*	
Program	**Insts.**	**TS**	**AT**	**TS**	**AT**	**TS**	**AT**
qsort_s	92	5	37.4	13	76.8	8063	36648.8
crc	147	1	15.9	4	30.3	385	1744.6
qsort_l	147	3	25.0	5	35.6	2922	13671.9
dijkstra_l	323	3062	16791.6	35219	177031.4	374961	N/A
dijkstra_s	323	650	3594.7	9822	49734.6	374606	N/A
caudio	418	0.1	7.4	0.1	7.8	10	53.7
daudio	418	0.1	10.5	0.1	14.9	10	53.6
bmath_s	532	494	2893.0	846	4621.4	203943	N/A
tetris	629	43	276.1	250	1319.2	7669	34660.6
bmath_l	643	13850	N/A	20699	N/A	374952	N/A
sha	657	201	1324.8	575	3317.6	258373	N/A
bitcnts	664	4	30.8	14	76.7	11212	51833.8
fft	742	272	1573.9	533	2894.9	61581	283411.3
2048	749	33	211.7	67	375.1	21156	99059.3
search_l	827	60	378.8	360	1844.8	35972	165931.4
search_s	827	3	23.0	24	130.2	1565	7240.7
snake	1065	25	160.3	57	312.3	1111	5131.0
patricia	1087	958	5951.6	2567	14365.4	374978	N/A
bf	3607	4	32.2	9	59.2	37730	N/A
rijndael	5866	280	1816.9	511	2984.4	374952	N/A
say	6859	225	1593.0	1200	9486.3	25078	115695.1
susan	12656	450	2795.4	5259	26623.3	225351	5401.1
toast	13930	291	1736.7	485	2778.2	170259	N/A
djpeg	52708	84	539.5	86	535.0	174	N/A
cjpeg	54837	334	2094.5	3773	19544.6	69202	N/A
Mean	6430.1	**853.3**	1756.6	**3295.1**	13258.3	**120648.6**	35675.5
Median	742.0	84.0	378.8	360.0	1582.0	35972.0	1744.6
Std	14730.4	2778.2	3460.5	8025.5	36739.4	149939.9	69840.5

6.7.3. Integrity Checking Defeatability

In this section, our objective is to qualify the difficulty of disabling protections. In a realistic scenario, often enough attackers have to first identify the protections. Thereafter, they mount their attacks to alter or disable a certain behavior in the program of interest.

In this experiment, our objective is to measure the impact of different composition of protections on the segmental or overall defeatability of a program. To estimate the defeat difficulty, we resort to the EPS metric we presented in Section 6.6.2. Composing protections is a non-trivial task [8]. To generate reliable and different compositions of protections, we resort to the `sip-shaker` composition framework. The `sip-shaker` framework utilizes an ILP solver to compose protections according to desired requirements. The framework comes with three baked-in modes for protection compositions: namely *max-manifest*, *max-explicit*, and *max-implicit* [8]. The max-manifest mode applies all possible protections. The max-explicit and max-implicit devise compositions that yield maximum explicit and maximum implicit coverages, respectively. **Data Set.** For the sake of this experiment, we generate three instances, one for each mode of the composition, of each program in the Mibench data set. We utilize SC, OH/SROH, and CSIV protections to increase the chances of forming protection chains. Having such chains in protected programs enables us to study the impact on the EPS metric. We define the order of protection applications as SC$\rightarrow$CSIV$\rightarrow$OH/SROH to ensure the formation of protection chains.

Results. We compute the EPS score at the level of functions for each program in the data set. Table 6.4 presents the median, mean, and standard deviation of the EPS score for the programs in the MiBench data set. One immediate benefit of this experiment is that the defender can quantify and subsequently compare the estimated steps that attackers need to take to defeat different compositions of protections. The results indicate that the EPS metric deems max-manifest configurations harder to defeat than other coverage related metrics. We elaborate further on the results in Section 3.7.

6.7.4. Resilience

The previous experiments provide the information that we need to compile a relation matrix. The fuse of results corresponding to the pattern-based attacks (Section 6.7.2), machine-learning-based attacks (Table 6.1), and taint-based attacks (Table 6.3) captures the resilience of different protections (i.e., the relations amongst protection and defenses).

6.7.5. Attack Defense Relation Table

The fuse of results corresponding to the pattern-based attacks (Section 6.7.2), machine-learning-based attacks (Table 6.1), and taint-based attacks (Table 6.3) captures the resilience of different protections (i.e., the relations amongst protection and defenses). Table 6.5 captures the compiled relation matrix.

6.7.6. A Complete Example of the Proposed Resilience Quantification

In this section, we use one of the programs in the MiBench data set (`qsort_l`) to demonstrate how practitioners can apply our methodology to firstly pick best protections and subsequently quantify the actual resilience of their protected programs.

As per obfuscations, we utilize the best combinations according to our localizability experiments (see Table 6.1). That is, we choose combinations of BC+IS_2+CFF_2 to obfuscate OH/SROH and CSIV protections, and BC_{100}+IS+CFF to obfuscate SC protection. After

Table 6.4.: EPS metric mean (M), median (Mdn), and standard deviation (σ) for the three different composition of protections: max-manifest, max-explicit, and max-implicit; #P, E, I denote the number of protections (manifests) in the solution, explicit, and implicit protection coverages, respectively

	Max-manifest Defeatability						*Max-explicit Defeatability*					*Max-implicit Defeatability*				
Program	**E**	**I**	**#P**	**M**	**Mdn**	σ	**I**	**#P**	**M**	**Mdn**	σ	**E**	**#P**	**M**	**Mdn**	σ
qsort_s	91	86	23	14.2	6	19.46	86	19	11	6	13.08	87	13	6.2	6	3.71
crc	368	365	56	13.11	9	9.11	365	55	12.67	9	8.60	366	53	11.78	9	8.09
qsort_l	76	71	22	21.33	6	22.4	71	18	16	6	14.85	72	12	8	6	3.56
dijkstra_l	673	668	123	22.4	20	13.87	668	121	21.87	20	12.96	669	116	20.53	20	11.9
dijkstra_s	673	668	123	22.4	20	13.87	668	121	21.87	20	12.96	669	116	20.53	20	11.9
caudio	1247	1245	115	12.38	11.5	8.45	1245	114	12.23	11.5	8.4	1246	113	12.08	10.5	8.42
daudio	1197	1195	99	11.15	10.5	5.97	1195	98	11	10.5	5.87	1196	97	10.85	9.5	5.87
bmath_s	852	787	255	44	11.5	129.65	787	228	38	11.5	105	788	164	23.78	11.5	46.95
tetris	3590	3567	1162	91.08	58	92.27	3567	1153	90.15	58	91.54	3586	1143	86.44	55	91.21
bmath_l	974	901	374	58.32	15	197.25	901	297	44.32	15	133.23	902	225	31.23	15	73.56
sha	2290	2287	599	90.82	47	141.89	2287	597	90.09	47	142.20	2288	594	89	47	142.74
bitcnts	1389	1383	134	14.29	11	13.27	1383	124	13	11	6.59	1384	117	12.1	11	3.19
fft	2110	2104	262	17.21	10	27.51	2104	229	14.07	10	10.34	2105	219	13.12	10	8.37
2048	2446	2436	662	42.56	26	68.69	2436	581	33.56	26	29.76	2437	569	32.22	26	28.28
search_l	1815	1795	181	11.48	8	21.3	1795	171	10.65	8	15.83	1796	152	9.06	8	6.65
search_s	1815	1795	181	11.48	8	21.3	1795	171	10.65	8	15.83	1796	152	9.06	8	6.65
snake	3408	3406	800	48.29	14	88.8	3406	793	47.29	14	88.72	3407	792	47.14	14	88.74
patricia	1641	1632	391	61.41	19	138.85	1632	255	29.41	19	29.54	1633	243	26.59	17	28.67
bf	5070	5057	282	15.7	8	46.28	5057	216	10.81	8	12.22	5058	200	9.63	8	6.93
rijndael	7079	7061	488	45.54	27	72.46	7061	419	35.68	27	29.44	7062	396	32.39	27	23.90
say	32912	32900	8389	116.21	100	115.14	32900	8384	113.18	97	115.14	32909	8375	113.04	97	115.16
susan	32313	32285	6112	140.73	83	157.8	32285	6102	137.54	80	157.52	32311	6091	137.25	80	157.15
toast	61481	61473	16931	225.94	135	301.86	61473	16921	222.83	132	301.92	61479	16918	222.79	132	301.94
djpeg	90789	90770	6282	37.75	28	32.95	90770	6258	34.65	25	32.38	90786	6256	34.64	25	32.34
cjpeg	93843	93824	6466	26.77	18	38.16	93824	6453	23.72	15	37.95	93839	6450	23.71	15	37.9
Mean	14006	13990	2020	**48.66**	**28.38**	71.94	13990	1996	**44.25**	**27.78**	57.27	13995	1983	**41.73**	**27.5**	50.15
Median	1815	1795	282	26.77	15	38.16	1795	229	23.72	15	29.44	1796	219	23.71	15	23.9
Std σ	27527	27526	3965	50.94	32.65	73.42	27526	3970	50.75	31.81	70.5	27530	3974	51.38	31.77	69.46

Table 6.5.: Attack-defense relation table capturing the degree of difficulty inflicted by defenses against attacks on integrity checking protections; numeric cell values of pattern matching, machine learning, and tainting correspond to the attacks effectiveness (f-measure) in localizations of hardened integrity checks; ✓and **X** denote susceptibility and tolerance to attacks; grayed out cells represent irrelevant configurations

Defenses		Attacks				
Integrity Checking	Obfuscation / Hardening	Pattern Matching	Machine Learning	Tainting	Symbolic Execution	Memory Split
SC	None	100%	100%	100%	▨	✓
	BC_{100}+CFF	▨	76%	▨	▨	✓
	IS+BC_{100}+CFF	▨	72%	▨	▨	✓
	BC_{100}+IS+CFF	▨	64%	▨	▨	✓
	BC+IS_2+CFF_2	▨	82%	▨	▨	✓
	CFF	▨	96%	100%	▨	✓
	Virt	▨	▨	**X**	✓	✓
	Self-modifying code	▨	▨			**X**
OH/ SROH	None	94%	99%	▨	▨	▨
	BC_{100}+CFF	▨	73%	▨	▨	▨
	IS+BC_{100}+CFF	▨	7%	▨	▨	▨
	BC_{100}+IS+CFF	▨	7%	▨	▨	▨
	BC+IS_2+CFF_2	▨	6%	▨	▨	▨
CSIV	None	100%	98%	▨	▨	▨
	BC_{100}+CFF	▨	16%	▨	▨	▨
	IS+BC_{100}+CFF	▨	16%	▨	▨	▨
	BC_{100}+IS+CFF	▨	13%	▨	▨	▨
	BC+IS_2+CFF_2	▨	1%	▨	▨	▨

deciding upon obfuscations, we feed them to `sip-shaker` (see Section 6.7.3) to synthesize a maximum-protection compositions of SC, OH/SROH, and CSIV protections combined with the specified obfuscations. The `qsort_1` program has only one function (`main`). Since the SC protection needs at least two functions to be applied, `sip-shaker` utilizes a basic function splitting transformation yielding three functions, namely `main`, `main0`, and `main1`. The list of the applied protections (manifests) are presented in Table 6.6. We assign a unique id to each protection manifest (see the Id column in Table 6.6). Figure 6.2 presents the defense graph of the composition capturing the relations amongst manifests. Node Ids in the graph correspond to the manifest Ids in Table 6.6. Note that graph nodes with no relations are omitted to improve readability. We then compute the EPS score for each of the nodes in the graph. Defeatability (EPS) and localizability (f-measure) scores of each node are annotated on the graph nodes as tuples of (f-measure, EPS). For the protected program we then compute the aggregate defeatability and localizability scores (as presented in Section 6.6.3) capturing the actual resilience (⟨ aggregate EPS , aggregate f-measure ⟩) of the applied protections. That is, $res(qsrot\_l) = \langle 21.33, 8.72E - 27 \rangle$.

Table 6.6.: List of all protection manifests applied to `qsort_l`; note that SC operates at the function level and thus we apply function splitting transformation, which breaks the `main` function into 3 functions (`main`, `main0`, `main1`), to improve the protection coverage

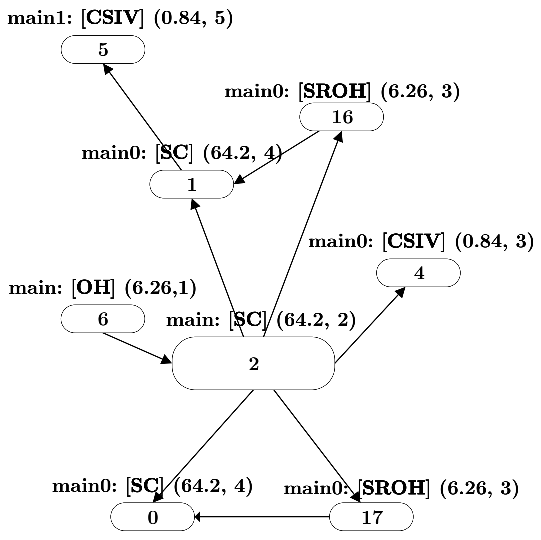
Id	Protection	EPS	FM	Protecting Function
0	SC	4	64.2%	main0
1	SC	4	64.2%	main1
2	SC	2	64.2%	main0
3	CSIV	1	0.84%	main
4	CSIV	3	0.84%	main0
5	CSIV	5	0.84%	main1
6	OH/SROH	1	6.26%	main
7	OH/SROH	1	6.26%	main
8	OH/SROH	1	6.26%	main
9	OH/SROH	1	6.26%	main
10	OH/SROH	1	6.26%	main
11	OH/SROH	1	6.26%	main
12	OH/SROH	1	6.26%	main
13	OH/SROH	1	6.26%	main
14	OH/SROH	1	6.26%	main
15	OH/SROH	1	6.26%	main
16	OH/SROH	3	6.26%	main0
17	OH/SROH	3	6.26%	main0
18	OH/SROH	1	6.26%	main0
19	OH/SROH	1	6.26%	main
20	OH/SROH	1	6.26%	main
21	OH/SROH	1	6.26%	main

6.8. Discussion

6.8.1. ML-Based Localization Attack

We would like to reiterate that we fuse structural features such as data/control flow dependencies, strongly connected components, and control flow graphs. We believe the effectiveness of ML-based attacks is due to the vulnerability of the obfuscation passes available in the O-LLVM framework to structural pattern extractions. In fact, according

Figure 6.2.: Defense graph of the `qsort_l` program with the maximum protection composition of SC, OH/SROH, and CSIV


main1: [**CSIV**] (0.84, 5)
5
main0: [**SROH**] (6.26, 3)
16
main0: [**SC**] (64.2, 4)
1
main0: [**CSIV**] (0.84, 3)
4
main: [**OH**] (6.26,1)
6
main: [**SC**] (64.2, 2)
2
main0: [**SC**] (64.2, 4)
0
main0: [**SROH**] (6.26, 3)
17

to further experiments we performed, plain TFIDF-based features demonstrated average effectiveness (f-measure) of 30%.

Interestingly, concealing SC checks in protected programs is far harder than OH/SROH and CSIV protections for both simple and MiBench data sets. We believe this is because SC guards have more code than OH/SROH and CSIV guards, leaving more significant fingerprints in the program blocks. Our results confirm that picking up such large fingerprints is easier for the learners while withstanding applied obfuscations. Nevertheless, it appears that the two obfuscation compositions of BC_{100}+IS+CFF (64.28%; Table 6.1) and BC+IS+CFF (10.71%; Table 6.2) induce the highest difficulties on the ML-based attacks for the MiBench and simple data sets, respectively. Based on the results from both data sets, we conclude that the combination of BC($_{100}$)+IS+CFF impairs localization attacks on SC protection best.

An insight of the ML-based attack results is the impact of the data set size on the effectiveness of SC localizations. There is a clear difference between the MiBench data set, which contains large programs, and the Simple data set results in terms of the SC detection efficacies. The reason, we believe, is a combination of obfuscations deficiency in their offered diversity together with the larger number of block samples that MiBench data set provides in the training process. We argue the high effectiveness of the simpler obfuscation compositions for the simple data set supports the very idea. For instance, IS+CFF yields an f-score of 2.7% for CSIV in the case of the simple data set while the corresponding value for the MiBench data set is 49.56%. Our experiments with the MiBench and Simple data sets indicate that the combinations of BC+IS_2+CFF_2 and IS+BC+CFF best suit OH/SROH and CSIV, respectively.

Taint-Based Attack on SC

In the case of SC protection, an alternative to the ML-based attack is to mount the taint-based attack. Our experiment (Table 6.3) confirms that without trace pollutions (obfuscations), the attack successfully localizes all the SC checks in all programs, except for the basicmath_l program. It appears that taint analysis fails to converge for the mentioned program within the given time frame (i.e., 80 hours). Applying CFF exhibit average trace size increase and average taint attack time by 3.9x (3295.1/853.3) and 7.5x (13258.3/1756.6), respectively. The same values show growth of 141.4x (120648.6/853.3) and 20.3x (35675.5/1756.6) in the case of Virt application.

Despite the bloated analysis and attack time, CFF (when applied to the 20% of the program besides the checks) is unable to mitigate the taint-based attack against the programs in our data set. One observation here is that for some programs (rows with AT value of N/A), the taint analysis fails to converge in the given time bound when Virt is utilized. This is due to the introduction of extensive taint poisoning inflicted by the virtual machine instruction fetch and execution model. Nevertheless, the failures do not mean virtualization is a silver bullet. Attackers need to first de-obfuscate virtualization (for instance, using [156]) and then run their trace-based attacks. We would have liked to use Salwan et al.'s script [156] to simulate these attacks. However, it is mainly designed to recover hash functions and thus cannot be applied to our binaries.

6.8.2. Defeatability

The EPS score provides a way first-order proxy to quantify the difficulty of defeating protections based on the number of imposed patches on attackers. A higher EPS value indicates a better quality of protection compositions. We argue compositions shall favor solutions with the maximum EPS. As the EPS value increases, localization attacks recall needing to increase substantially. For a given segment with EPS score of N, attackers need to localize all the N protections; otherwise, the attack will fail. Simply put, when the EPS score is high, the localization attack has to have considerably higher f-scores. Note that we capture the relation between localizability and defeatability in the resilience metric.

Nevertheless, the EPS value by —no means— translates to the attack effort. Our investigations confirm that IS, BC, and CFF obfuscations have no impact on the defeatability of the protections. This is mainly because our protection guards boil down to a comparison instruction in which the expected value is verified. This can be further hardened by hiding such comparison instructions using more advanced obfuscations. Alternatively, comparisons can be replaced with more robust reactions, for instance, by intertwining the checks with the program logic. Instead of comparing values, the calculated hashes can be used in the calculation of jump addresses in protected programs [47]. That is, the degree of the entanglement of the response mechanism in the program logic has a significant impact on the actual effort. The weighted EPS can enable defenders to approximate the effort by assigning difficulty coefficients to protections. However, we do acknowledge that coming up with characteristic coefficients is quite tricky. This calls for further research.

6.9. Threats to Validity

In this work, we split the attacker's effort into two hypothetically disjoint tasks: localizability and defeatability. We emphasize that such a split is merely for the sake of classification and subsequent mapping of attacks and defenses. In reality, the two are tightly coupled. Harder localizability often implies higher defeatability and vice versa.

Localizability is computed on the basis of the f-measure on our data set of programs. This value may not generalize for programs of different nature (and for practicality reasons, we relied on a medium-size size set of benchmark programs with results that may naturally be different for larger sets). That is, the f-measure can potentially be worse or better for programs with logics similar to the protection guards. For instance, a program with extensive operations on constant data can be hit by a large number of false positives in the OH/SROH classification. In Table 6.1 we reported on f-score, standard deviation and median of classifications. We argue high f-scores with low standard deviations show both good and robust classifications. High f-scores with high standard deviations can be interpreted as frail classifiers or partially effective obfuscations. That is, some programs have better obfuscation amenability and hence lower f-scores. Conversely, some programs or (segments of) appear to be classified more effectively. Another possible explanation is a weak classification.

The proposed EPS metric captures defeatability for acyclic protections. In our work, all the reviewed protections, as well as the compositions, preserve the acyclicity property. Therefore, we believe the acyclic EPS metric suffices our needs, but this may not be the case for other protections.

6.10. Conclusions

Measuring the resilience of combinations of software protection transformations is crucial for risk analysis. Without a way to estimate the difficulty of attacking protections, the degree of trust in the system is unknown. To answer our 1st research question (RQ1), we propose a methodology that quantifies the resilience of software integrity protections combined with obfuscation and other hardening transformations, as a conjecture of localizability and defeatability of protections. We conducted empirical evaluations, ranging pattern matching, taint-based, and machine-learning-based attacks, to measure the effectiveness of defenses approximating the localizability of protections. To answer our 2nd research question (RQ2), our results showed which obfuscations hide which integrity protections best for the representative data sets of programs. We also proposed a metric called *Estimated Patch Steps* (EPS) to estimate the number of patches that compositions of protections impose on attackers offering a first-order proxy of the defeatability of protections.

One direction for future work is to develop tailored obfuscations against machine-learning-based localizations. Regarding defeatability, we need further research to quantify the effort of patching attacks. A quantification metric must be developed to assign difficulty coefficients to each protection according to the entanglement of response mechanisms in the program logic. Another future work direction is verifying the external validity of the defeatability and localizability metrics in hacking contests (e.g., CTFs).

Part V.

Conclusions & Future Work

7. Conclusions

This chapter presents the conclusions, lessons learned, future work, and the limitations of this thesis.

7.1. Research Findings

We started this thesis to answer the main research question of *"Which security guarantees at which cost could software integrity protection schemes offer against automated MATE attackers?"* To answer the main research question, we further refined it into a set of subquestions. In this section, we elaborate on the answers to each subquestion.

RQ1 Which aspects of software integrity protection techniques are security-relevant?
In Appendix A, we surveyed the literature and further performed an in-depth analysis of the reviewed papers. The outcome of the analysis (captured as a taxonomy in Chapter 2) indicates two main categories of integrity assets, namely *data* and *behavior*. Attacks on integrity (breaches) tend to target either of or both of these assets in systems. We proposed a unified set of criteria comprised of four main categories, namely *measure*, *protection level*, *trust anchor*, and *overhead*, to classify integrity protection techniques in the defense dimension of our proposed taxonomy. The measure category is comprised of eight subcategories: *monitor*, *response*, *transformation*, *check*, *hardening*, *remote*, and *local*. We refined each of these subcategories with two to six further specialized sub-subcategories. We mapped the protection literature to our defense dimension and identified the criteria that contribute to the mitigation of each attack via a collection of correlation analyses. Among the captured criteria, *hardening*, *monitor*, and *response* appear to have the highest impact on the security of the protections. Another indirectly related criterion is *overhead*. That is, we may not be able to use a secure scheme with a high overhead in a device with limited computational resources. Since the imposed overhead by protections is a dominant factor in the applicability of schemes in various use cases, we also looked into the execution overheads by classifying reviewed papers according to three levels of overhead classes, namely fair, medium, and high, corresponding to $\leq$ 100%, between 100% and 200%, and $\geq$ 200%, respectively. Papers with no performance benchmarks as well as questionable evaluations were marked with N/A labels. We believe these criteria provide a unified basis for qualification of the security of integrity protection techniques.

RQ2 How do integrity protection techniques mitigate attacks?
To answer this question, we reviewed and further mapped the protection schemes in the literature to the kind of attack (namely, binary patching, control flow subverting, process memory patching, and runtime data manipulation) that they aim to mitigate. Section 2.5.2 provides a precise mapping of the desired attacks to mitigate and the relevant protection schemes. Schemes use a combination of hardening measures and static code representations to safeguard the program binaries. A majority of defenses use code invariants verifications to protect against process memory tampering attacks. Data invariants and traces are two conventional means to defend against runtime data manipulation attacks. The literature indicates that trace-based verifications can effectively raise the bar against attacks on the program control flow [2, 19, 31, 49, 51, 99, 102, 118, 144, 145].

We also gathered published attacks on the protection schemes (see Section 2.5.8) and thereby analyzed the security of the protections against known attacks.

RQ3 What are the shortcomings of the existing protection schemes?
Amongst the reviewed protections Chapter 2, we identified self-checksumming and oblivious hashing as two prominent protections. We then performed an in-depth analysis on each of these schemes in Chapters 3 and 4. We proposed three main metrics for analyzing shortcomings of schemes: *applicability*, *coverage*, and *resilience*. We found both schemes (SC and OH) to have deficiencies. As we pointed out in Chapter 3, OH requires users to segregate input independent instructions from other program instructions manually. We believe manual segregation of instructions is labor-intensive and thus comes with a substantial applicability problem. Moreover, the number of input-independent instructions are critically limited in programs (less than 1% in our data set). Therefore, the effortful task of segregating instructions, at its best, can protect an insignificant set of program instructions. That is, low coverage is a fundamental problem in OH. On the bright side, OH exhibits acceptable resilience to known attacks. Unlike OH, SC comes with almost no coverage constraints (see Chapter 4). The protector instead regulates the coverage. That is, the user can decide which portions of the programs shall be protected. Nevertheless, SC comes with severe composability constraints (applicability) as well as susceptibility to the trace-based [150] and memory split [177, 183] attacks.

RQ4 How can we improve protection schemes aiming for enhanced security?
Concerning OH (Chapter 3), we first implemented a tool for the input dependency analysis based on graph reachability analysis. This tool enabled us to automate the OH protection fully. Second, we proposed Short Range Oblivious Hashing to extend the protection coverage to all branching conditions and instructions with control flow dependency on the input. Besides the 20 and 35 fold increases in the instruction and block coverages, respectively, the SROH extension provisions fully-fledged control-flow integrity protection. Integral SC guards then protect the remainder of the

instructions (i.e., input-data-dependent instructions). We also introduce another layer of protection by having OH and SROH guards to verify SC guards. The extended layer adds resilience against attacks on SC, imposing more effort on the adversaries.

We then gave heed to enhancing SC protection. Improved security is one immediate benefit of building intertwined protection with OH. That is, when we utilize the layered protection (SC and OH), attackers have to defeat OH protections in addition to SC guards. However, detecting SC remains somewhat feasible as SC guards leave recognizable tainting patterns in the program trace. Moreover, the post patching step of SC protection severely limits applications of additional protections. Without adequate applications of other protections, particularly obfuscations, detecting SC guards might become even trivial, e.g., using pattern matching. In Chapter 4, we proposed an SC protection atop virtualization obfuscation. The proposed VirtSC has two major advantages. First, it requires no post-patching step and hence remarkably facilitates the application of other protections (enhanced composability). Second, virtualization obfuscation blows the program trace hardening against taint and trace-based attacks.

RQ5 How can we compose protections to mitigate a wider range of attacks?
While the idea of composing protection at first appeared to be trivial, the task quickly became rather complicated. Simultaneously applying certain protections on the same program segment may lead to conflicts among protections triggering false tampering alerts. More importantly, the decision of utilizing which protection on each segment can have a notable impact on the resilience as well as the runtime overhead of protection compositions. The bottom line is that we have to optimally decide on which protection shall protect which segments of a program. In Chapter 5, we proposed an ILP-based optimization system to find optimal compositions of protection for a given program. Besides the coverage metric, we proposed a novel *implicit coverage* metric that captures the degree of overlaps and cross checks that composed protections form together. We formulated the composition problem as an ILP problem with objective functions of minimizing overhead, maximizing (explicit) coverage, and maximizing implicit coverage. Our optimization technique yielded overhead decreases of 39% and 500% for maximum protection (coverage) and (compared to) state of the art heuristics, respectively.

RQ6 How to quantify the resilience of individual and composed protection techniques?
In Chapter 6, we proposed a methodology for estimating the resilience of protections against automated attacks. We took the perspective of defenders who aim to quantify the trouble that attackers shall endure to be able to break protections. The methodology suggests splitting the ongoing battle between attackers and defenders into two hypothetically disjoint fronts: localizability and defeatability. Various automated attacks are then mapped to each of these battlefronts. On the defender side, relevant defenses are considered against each of the attacks. Our methodology suggests cap-

turing the effectiveness of attacks on defenses (or vice versa) as the actual *resilience* of protections. For localizability attacks, we considered f-measure as our effectiveness metric, while for defeatability attacks, we proposed a novel metric (EDS) that captures the steps attackers must take to defeat protections.

Overall Conclusion. One goal of this thesis was to explore efficient and secure constructs to protect software integrity. We achieved this goal by proposing various enhancements [5, 7] as well as a composition framework [8] for integrity protection techniques. The other goal was to build a unified method for assessing the strength of protections. We accomplished this goal with our quantification methodology (see Chapter 6). All in all, we met all the goals we set for this thesis.

7.2. Recap of the Research Gaps and Contributions

7.2.1. Gaps

In the previous chapters, we discussed the related work and identified gaps. Here we briefly reiterate over the research gaps:

G1 Lack of a set of unified criteria to analyze software-based integrity protection schemes;

G2 Unknown security guarantees and performance bounds of protection schemes;

G3 Severe shortcomings in two well-known protection schemes, namely, oblivious hashing and self-checksumming;

G4 Nonexistence of effective integrity protection compositions;

G5 Existence of no means to quantify the resilience of protections; and

G6 Limited publicly-available implementations of protection schemes

7.2.2. Contributions

C1 Regarding G1, we proposed a taxonomy for software integrity protection capturing advantages and disadvantages, constraints, shortcomings, and security guarantees of different protections (refer to [9]);

C2 Throughout the thesis, we benchmarked integrity protection schemes w.r.t. security, performance using datasets of real-world programs to close G2 (refer to [5, 7, 8]);

C3 Proposed a novel extension so-called Short Range Oblivious Hashing to address the coverage problem of the vanilla oblivious hashing; and proposed a self-checksumming protection atop virtualization to enhance the composability of protections closing G3 (refer to [5, 7]);

C4 To close G4, proposed and evaluated an ILP-based optimization framework that guarantees a conflict-free and optimized composition of protections yielding a 5-fold decrease in overhead compared to the heuristic-based compositions (refer to [8]);

C5 To address G5, we propose a quantification methodology to estimate the resilience of protections via metrics surrogating the localizability and defeatability of combinations of protections (refer to Chapter 6); and

C6 Open sourced all the developed protections, composition framework, and evaluation benchmarks to support the reproducibility of results and thereby address G6 (available at `https://github.com/tum-i22/sip-toolchain`).

7.3. Results and Lessons Learned

In this section, we highlight interesting results and lessons learned throughout the thesis.

- **Taxonomy of software integrity protection built around a representative protection process**
  We proposed a taxonomy encompassing three dimensions: system, defense, and attack. The system dimension captures assets and their exposure in various lifecycles of a system. The attack dimension captures threats and hints on the desired protection requirements. The defense dimension provides a technical catalog of protections capturing characteristics of protections, attacks each scheme can mitigate, and potential overheads on the system.

- **Self-checksumming is the most used technique for integrity protection**
  In the analysis of the literature (Section 2.5.5), we found out that self-checksumming (self-hashing) is the prominent technique for checking the integrity of programs in literature.

- **The vanilla OH has significant practical drawbacks with a barely 1% program coverage**
  OH cannot protect input-dependent instructions. Our analysis confirmed that a substantial portion of program instructions directly or indirectly depend on input values.

- **The existing static analysis tools (as is) cannot single out input-dependent instructions** After attempting to segregate input dependent instruction using the existing tools, we realized for the existing open source projects cannot directly meet our requirements. Our needs included input flow detection from system and library calls with no sink and source specifications whatsoever, distinctions of data and control flow dependencies, and propagations of dependencies at the module level (inter-procedural) instead of functions only. These requirements led us to implement an analysis tool from scratch.

- **Our SROH extension increases instruction and block coverages by factors of 20 and 35**
  Our SROH technique enables protecting data-independent instructions that reside in data-dependent branches of programs.

- **SROH enhances the amenability of OH to SC (and other protections)**
  SROH can protect all branching conditions in programs regardless of their input dependency. Consequently, all the branching conditions added by other protections can potentially be protected by SROH. Complete control-flow integrity protection is another benefit of SROH.

- **The composition of OH/SROH and SC constitutes a protection mechanism with tolerable performance overheads**
  The combination of OH/SROH and SC with networks of checkers constituted an overhead of 52% for non-CPU intensive programs in our data set. Although the overhead can be very high for some programs, e.g., programs with nested loops, we argued the construct could have acceptable overhead when preferably parts of programs (deemed as sensitive) need to be protected instead of entire programs.

- **The conventional SC protection comes with atypical memory accesses and poor composability**
  Reading and subsequently hashing the code segment is a sign that could potentially give SC guards away. Moreover, hashing the code segment requires a post patching step to adjust the expected hash values after the binary is compiled.

- **VirtSC removes the obvious code segment accesses as well as the post patching step**
  The proposed VirtSC scheme moves the checksumming to the level of virtualized code instead of the program code segment. Consequently, apparent self-memory reads no longer hamper the stealth of the protection. Moreover, the virtualized code is available upon transformations, and thus hashes can be calculated directly (i.e., no post-patching whatsoever).

- **Indirect threading optimization can yield a five-fold decrease in the overhead VirtSC**
  The emulator of our VirtSC protection imposes a high execution overhead, most of which is solely about the logic of the emulator itself than the protection. We observed that applying emulator optimization, such as indirect threading, can boost the performance up to 500%. However, this optimization yields fewer branch reuses, which can ease attacks on the emulator. Further research needs to be done on security-preserving optimization techniques.

- **The problem of optimal protection composition can be formulated as an ILP optimization problem**

We proposed that an optimal protection composition shall address two problems: **(i)** deciding on the order in which transformations shall be applied; and **(ii)** deciding upon which program segments shall be protected with which protections to avoid conflicts while maximizing security and minimizing the cost. We then formulated both problems as subjecting constraints in an ILP optimization problem. In our experiments, we noticed that the GLPK solver takes more than an hour to find solutions for six large programs in our data set. However, the Gurobi solver finds optimal composition solutions for those large programs in less than a second. Therefore, it is crucial to utilize adequate solvers for larger compositions.

7.4. Limitations

We discussed the limitations of each part of the thesis in their corresponding chapters. Here we briefly reiterate over general limitations of the thesis.

7.4.1. Correctness

We did not provide formal proofs for our proposed protections (SROH and VirtSC) and ILP-based composition. We implemented the input dependency tool based on the proposed approach by Scholz et al. [159]. We carried out tests and manual inspections. Despite our best efforts in tests, potential pitfalls or bugs in the implementations are possible.

7.4.2. Prototype Implementation

We implemented our tools in LLVM 6.0. In theory, our protections should work for any program that is lifted to LLVM-IR. However, we have not tested our tools with programs beyond simple and MiBench data sets. Both OH/SROH and SC protection implementations only contain the post-patching procedures for x86 Linux Executable formats (i.e., standard ELF). Extending the post patcher for other architectures is merely a matter of engineering efforts.

The CSIV protection verifies the program traces to sensitive functions. We used LLVM native program call graph analysis to explore all legitimate paths to sensitive functions. The analysis can potentially overlook indirect calls or callback functions. A user of CSIV has to cater for such cases manually. Moreover, the legitimate traces are stored in a text file located next to the protected binary. In practice, such information needs to be securely stored in the binary using white-box cryptography. Pure implementation efforts can address both of these limitations.

Our implementation of the input dependency analyzer makes conservative assumptions about the dependency of program pointers on the input. This ensures that OH/SROH never hashes any input dependent values, and thus hash consistencies are guaranteed. On the flip side, our analysis pass can, in the worst case, misclassify some input-independent pointers. We consider a precise static program dependency analysis to be a gap in the literature.

7.4.3. Data sets

All the reported results are based upon our sample C programs. We used the MiBench data set, the sample data set, and three open-source games. A majority of these programs are CPU-intensive and thus tend to capture a worst-case scenario. That is, we expect better results with less CPU-intensive programs. Nevertheless, we believe conducting experiments on programs with real integrity constraints (such as DRMs) is a better way of studying protections' impact. Unfortunately, we did not have access to such programs. Application to MiBench may lead to results that are not representative of real DRM programs, because we don't know what would be a relevant asset to protect. The other obvious concern is, of course, that the MiBench program results may not be representative of all other programs. However, if we don't focus on DRM programs, even then, we don't know if the findings for MiBench generalize or to what they generalize.

7.4.4. Automated Attacks

In our security analyses, we considered automated attacks. This may not generalize for a human MATE attacker. We believe human ingenuity, together with the extended attack surface in the MATE case, makes it extremely hard to study the actual security of protections. It is worth mentioning that our study's limitations are in line with the constraints of other work in the literature and other prototype software.

7.4.5. Side Channels

In this work, we did not study the feasibility of side-channel attacks on the proposed schemes.

7.4.6. Resilience Quantification

Regarding the resilience quantification methodology, we do acknowledge that our metrics rather capture an approximation of the localizability and defeatability. In the case of defeatability's metric (Estimated Patch Steps), we don't know if the abstract number of steps corresponds to any real-world efforts. In Chapter 6, we discussed the limitations of our resilience quantification methodology in detail.

7.5. Future Work

We provided a detailed discussion of possible directions for future work in each chapter of this thesis. In this section, we provide a summary of possible ventures for future work.

7.5.1. Additional Integrity Protections

In our experiments, we only included four integrity protection schemes. All these schemes operate locally. It would be useful to include further protections, particularly remote-based protections.

7.5.2. Controlled Human-Based MATE Attacks

A valuable addition to our experiments is human-based attacks. To do so, we first need to develop a basis to measure the degree of expertise of attackers objectively. Then, we need to conduct experiments to study the resilience of protections in a controlled environment. Such experiments need to be repeated for various combinations of protections with professional attackers. It would be interesting to empirically study aspects such as the time to break, the number of defeated protections, and resistant protections with human attackers. In this thesis, we did not manage to conduct such experiments due to a multitude of reasons: **i)** it isn't easy o find people with such skill-sets; **ii)** hiring such people was well beyond our research budget; and **iii)** even if we had managed to find and hire such specialists, it would have been hard to argue why such results generalize.

7.5.3. Formal Proofs

Verifying the correctness of the proposed protection schemes as well as the ILP-based composition appears as a good future work.

7.5.4. Fusion with Hardware-Based Protection

With the current trend in hardware-based security, it would be interesting to combine software and hardware protections, and understand the provided guarantees. Another exciting venture is to integrate hardware-based protection into the composition framework.

Bibliography

[1] A Survey on Function and System Call Hooking Approaches. *Journal of Hardware and Systems Security*, 1(2):114–136, 2017. ISSN 2509-3436. doi: 10.1007/s41635-017-0013-2. URL `https://doi.org/10.1007/s41635-017-0013-2`.

[2] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[3] Bert Abrath, Bart Coppens, Stijn Volckaert, Joris Wijnant, and Bjorn De Sutter. Tightly-coupled self-debugging software protection. In *Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering*, SSPREW '16, pages 7:1–7:10, New York, NY, USA, 2016. ACM, ACM. ISBN 978-1-4503-4841-6. doi: 10.1145/3015135.3015142.

[4] Mohsen Ahmadvand, Anahit Hayrapetyan, Sebastian Banescu, and Alexander Pretschner. Practical integrity protection with oblivious hashing. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 40–52. ACM, 2018.

[5] Mohsen Ahmadvand, Anahit Hayrapetyan, Sebastian Banescu, and Alexander Pretschner. Practical integrity protection with oblivious hashing. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 40–52, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6569-7. doi: 10.1145/3274694.3274732.

[6] Mohsen Ahmadvand, Alexander Pretschner, Keith Ball, and Daniel Eyring. Integrity protection against insiders in microservice-based infrastructures: From threats to a security framework. In *Workshop on Microservices: Science and Engineering (MSE), STAF'2018, Toulouse, France*, 2018. doi: 10.1007/978-3-030-04771-9_43.

[7] Mohsen Ahmadvand, Daniel Below, Sebastian Banescu, and Alexander Pretschner. Virtsc: Combining virtualization obfuscation with self-checksumming. In *Proceedings of the 3rd ACM Workshop on Software Protection*, SPRO'19, pages 53–63, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6835-3. doi: 10.1145/3338503.3357723.

[8] Mohsen Ahmadvand, Dennis Fischer, and Sebastian Banescu. Sip shaker: Software integrity protection composition. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, pages 203–214, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450376280. doi: 10.1145/3359789.3359848.

[9] Mohsen Ahmadvand, Alexander Pretschner, and Florian Kelbert. Chapter eight - a taxonomy of software integrity protection techniques. In Atif M. Memon, editor, *Advances in Computers*, volume 112 of *Advances in Computers*, pages 413–486. Elsevier, 2019. doi: 10.1016/bs.adcom.2017.12.007. URL `http://www.sciencedirect.com/science/article/pii/S0065245817300591`.

[10] Adnan Akhunzada, Mehdi Sookhak, Nor Badrul Anuar, Abdullah Gani, Ejaz Ahmed, Muhammad Shiraz, Steven Furnell, Amir Hayat, and Muhammad Khurram Khan. Man-at-the-end attacks: Analysis, taxonomy, human aspects, motivation and future directions. *Journal of Network and Computer Applications*, 48:44–57, 2015.

[11] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE, 2008.

[12] Bertrand Anckaert, Mariusz Jakubowski, and Ramarathnam Venkatesan. Proteus: virtualization for diversified tamper-resistance. In *Proceedings of the ACM workshop on Digital rights management*, pages 47–58. ACM, 2006.

[13] Dennis Andriesse, Herbert Bos, and Asia Slowinska. Parallax: Implicit code integrity verification using return-oriented programming. In *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, pages 125–135. IEEE, 2015.

[14] David Aucsmith. Tamper resistant software: An implementation. *Proceedings of the First International Workshop on Information Hiding*, pages 317–333, 1996. doi: 10.1007/3-540-61996-8_49. URL `http://link.springer.com/chapter/10.1007/3-540-61996-8{_}49`.

[15] David Aucsmith. Tamper resistant software: An implementation. In *International Workshop on Information Hiding*, pages 317–333. Springer, 1996. ISBN 3-540-61996-8. doi: 10.1007/3-540-61996-8_49. URL `http://link.springer.com/chapter/10.1007/3-540-61996-8{_}49`.

[16] Avast Lab. Wannacry ransomware, 2017. URL `https://blog.avast.com/ransomware-that-infected-telefonica-and-nhs-hospitals-is-spreading-aggre`

[17] Sebastian Banescu and Alexander Pretschner. A tutorial on software obfuscation. In Atif M. Memon, editor, *Advances in Computers*, volume 108 of *Advances in Computers*, pages 283–353. Elsevier, 2018. doi: 10.1016/bs.adcom.2017.09.004. URL `http://www.sciencedirect.com/science/article/pii/S0065245817300499`.

[18] Sebastian Banescu, Martín Ochoa, Nils Kunze, and Alexander Pretschner. Idea: Benchmarking indistinguishability obfuscation–a candidate implementation. In *International Symposium on Engineering Secure Software and Systems*, pages 149–156. Springer, 2015. doi: 10.1007/978-3-319-15618-7_12.

[19] Sebastian Banescu, Alexander Pretschner, Dominic Battré, Stéfano Cazzulani, Robert Shield, and Greg Thompson. Software-based protection against changeware. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 231–242. ACM, 2015.

[20] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 189–200, New York, NY, USA, 2016. ACM, ACM. ISBN 978-1-4503-4771-6. doi: 10.1145/2991079.2991114.

[21] Sebastian Banescu, Ciprian Lucaci, Benjamin Krämer, and Alexander Pretschner. Vot4cs: A virtualization obfuscation tool for c#. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, pages 39–49, New York, NY, USA, 2016. ACM, ACM. ISBN 978-1-4503-4576-7. doi: 10.1145/2995306.2995312.

[22] Sebastian Banescu, Mohsen Ahmadvand, Alexander Pretschner, Robert Shield, and Chris Hamilton. Detecting patching of executables without system calls. In Gail-Joon Ahn, Alexander Pretschner, and Gabriel Ghinita, editors, *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 185–196. ACM, The Association for Computing Machinery, 2017. ISBN 978-1-4503-4523-1/17/03. doi: 10.1145/3029806.3029835.

[23] Sebastian-Emilian Banescu. *Characterizing the Strength of Software Obfuscation Against Automated Attacks*. PhD thesis, Technische Universität München, 2017.

[24] UC Santa Barbara. Interpretation and interpreter optimization, 2018. URL `https://www.cs.ucsb.edu/~cs263/lectures/interp.pdf`.

[25] Benoit Baudry and Martin Monperrus. The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Computing Surveys (CSUR)*, 48 (1):16, 2015. doi: 10.1145/2807593.

[26] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[27] Georg Becker. Merkle signature schemes, merkle trees and their cryptanalysis. *Online im Internet:http://imperia. rz.rub.de*, 9085, 2008.

[28] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3588–3600. Curran Associates, Inc., 2018.

[29] Robert L Bernstein. Producing good code for the case statement. *Software: Practice and Experience*, 15(10):1021–1024, 1985.

[30] Dirk Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 887–904. Springer, 2016.

[31] Brian Blietz and Akhilesh Tyagi. Software tamper resistance through dynamic program monitoring. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3919 LNCS:146–163, 2006. ISSN 1611-3349. doi: 10.1007/11787952_12.

[32] Guido Bologna and Yoichi Hayashi. A Comparison Study on Rule Extraction from Neural Network Ensembles, Boosted Shallow Trees, and SVMs. *Applied Computational Intelligence and Soft Computing*, 2018:4084850, 2018. ISSN 1687-9724. doi: 10.1155/2018/4084850. URL `https://doi.org/10.1155/2018/4084850`.

[33] Marc Boulle. Compact mathematical formulation for graph partitioning. *Optimization and Engineering*, 5(3):315–333, 2004.

[34] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. Tytan: tiny trust anchor for tiny devices. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[35] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. *arXiv preprint arXiv:1702.07521*, page 33, 2017.

[36] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous attestation. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, 2004. doi: 10.1145/1030083.1030103.

[37] Saartje Brockmans, Raphael Volz, Andreas Eberhart, and Peter Löffler. Visual modeling of owl dl ontologies using uml. In *International Semantic Web Conference*, volume 3298, pages 198–213. Springer, 2004.

[38] ED Bryant, Mikhail J Atallah, and MR Stytz. A survey of anti-tamper technologies. *CrossTalk: The J. Defense Softw. Engin*, 17(11):12–16, November 2004.

[39] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 162–175, New York, NY, USA, 1986. ACM. ISBN 0-89791-197-0. doi: 10.1145/12276.13328.

[40] Jan Cappaert, Bart Preneel, Bertrand Anckaert, Matias Madou, and Koen De Bosschere. Towards tamper resistant code encryption: Practice and experience. In *International Conference on Information Security Practice and Experience*, pages 86–100. Springer, 2008.

[41] Martim Carbone, Weidong Cui, Marcus Peinado, Long Lu, and Wenke Lee. Mapping Kernel Objects to Enable Systematic Integrity Checking. *Analysis*, pages 555–565, 2009.

[42] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006. URL `http://dl.acm.org/citation.cfm?id=1298455.1298470$\delimiter"026E90F$nhttp://www.usenix.org/event/osdi06/tech/full{_}papers/castro/castro{_}html/`.

[43] Luigi Catuogno and Ivan Visconti. A format-independent architecture for run-time integrity checking of executable code. In *International Conference on Security in Communication Networks*, pages 219–233. Springer, 2002.

[44] Joshua Cazalas, J Todd McDonald, Todd R Andel, and Natalia Stakhanova. Probing the limits of virtualized software protection. In *Proceedings of the 4th Program Protection and Reverse Engineering Workshop*, page 5. ACM, 2014.

[45] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Bart Coppens, Bjorn De Sutter, Paolo Falcarin, and Marco Torchiano. How professional hackers understand protected code while performing attack tasks. In *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on*, pages 154–164. IEEE, 2017. doi: 10.1109/icpc.2017.2.

[46] Mariano Ceccato, Paolo Tonella, Cataldo Basile, Paolo Falcarin, Marco Torchiano, Bart Coppens, and Bjorn De Sutter. Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. *Empirical Software Engineering*, 24(1):240–286, February 2019. ISSN 1573-7616. doi: 10.1007/s10664-018-9625-6.

[47] Hoi Chang and Mikhail J Atallah. Protecting software code by guards. In *ACM Workshop on Digital Rights Management*, pages 160–175. Springer, 2001.

[48] Hsiang-Yang Chen, Ting-Wei Hou, and Chun-Liang Lin. Tamper-proofing basis path by using oblivious hashing on java. *SIGPLAN Not.*, 42(2):9–16, February 2007. ISSN 0362-1340. doi: 10.1145/1241761.1241762.

[49] Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz H Jakubowski. Oblivious hashing: A stealthy software integrity verification primitive. In *International Workshop on Information Hiding*, pages 400–414. Springer, 2002.

[50] Zhe Chen, Chunfu Jia, Tongtong Lv, and Tong Li. Harden tamper-proofing to combat mate attack. In Jaideep Vaidya and Jin Li, editors, *Algorithms and Architectures for Parallel Processing*, pages 98–108, Cham, 2018. Springer International Publishing. ISBN 978-3-030-05063-4.

[51] Zhe Chen, Zhi Wang, and Chunfu Jia. Semantic-integrated software watermarking with tamper-proofing. *Multimedia Tools and Applications*, 77(9):11159–11178, 2018. ISSN 1573-7721. doi: 10.1007/s11042-017-5373-7. URL `https://doi.org/10.1007/s11042-017-5373-7`.

[52] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*, 2019.

[53] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-box cryptography and an aes implementation. In *International Workshop on Selected Areas in Cryptography*, pages 250–270. Springer, 2002.

[54] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C. Van Oorschot. White-box cryptography and an aes implementation. In Kaisa Nyberg and Howard Heys, editors, *Selected Areas in Cryptography*, pages 250–270, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-36492-4.

[55] Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales, Daniele Sgandurra, and Diego Zamboni. Cloud security is not (just) virtualization security: a short paper. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 97–102. ACM, 2009.

[56] Frederick B Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, 1993.

[57] C Collberg, S Martin, J Myers, and B Zimmerman. The tigress diversifying c virtualizer, 2015.

[58] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, 2002.

[59] Christian Collberg. Defending against remote man-at-the-end attacks, 2017. URL `https://www2.cs.arizona.edu/projects/focal/security/project1.html`.

[60] Christian Collberg. Code obfuscation: Why is this still a thing? In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, CODASPY '18, pages 173–174, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5632-9. doi: 10.1145/3176258.3176342.

[61] Christian Collberg and Jasvir Nagra. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Addison-Wesley Professional, 1st edition, 2009. ISBN 0321549252, 9780321549259.

[62] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[63] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 184–196, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: 10.1145/268946.268962.

[64] Christian Collberg, Jasvir Nagra, and Fei-Yue Wang. Surreptitious software: Models from biology and history. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 1–21. Springer, 2007.

[65] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 319–328. ACM, 2012.

[66] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28 (8):735–746, 2002.

[67] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 275–284. ACM, 2011.

[68] James R Cordy. Txl-a language for programming language tools and applications. *Electronic notes in theoretical computer science*, 110:3–31, 2004.

[69] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[70] CSIRO's Data61. Stellargraph machine learning library. `https://github.com/stellargraph/stellargraph`, 2018.

[71] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656, March 2016. doi: 10.1109/SANER.2016.43.

[72] Nenad Dedić, Mariusz Jakubowski, and Ramarathnam Venkatesan. A graph game model for software tamper protection. In *International Workshop on Information Hiding*, pages 80–95. Springer, 2007.

[73] Yves Deswarte, Jean-Jacques Quisquater, and Ayda Saïdane. Remote integrity checking. In *Integrity and internal control in information systems VI*, pages 1–11. Springer, 2004.

[74] Prashant Dewan, David Durham, Hormuzd Khosravi, Men Long, and Gayathri Nagabhushan. A hypervisor-based system for protecting software runtime memory and persistent storage. In *Proceedings of the 2008 Spring Simulation Multiconference*, SpringSim '08, page 828–835, San Diego, CA, USA, 2008. Society for Computer Simulation International, Society for Computer Simulation International. ISBN 1565553195.

[75] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. *2019 IEEE Symposium on Security and Privacy (SP)*, pages 472–489, 2019.

[76] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2007.01.015.

[77] Paolo Falcarin, Stefano Di Carlo, Alessandro Cabutto, Nicola Garazzino, and Davide Barberis. Exploiting code mobility for dynamic binary obfuscation. In *Internet Security (WorldCIS), 2011 World Congress on*, pages 114–120. IEEE, 2011.

[78] Rafael Fedler, Sebastian Banescu, and Alexander Pretschner. Isa^{2}r: Improving software attack and analysis resilience via compiler-level software diversity. In *International Conference on Computer Safety, Reliability, and Security*, pages 362–371. Springer, 2014.

[79] Wu-chang Feng, Ed Kaiser, and Travis Schluessler. Stealth measurements for cheat detection in on-line games. In *Proceedings of the 7th ACM SIGCOMM Workshop on Network and System Support for Games*, pages 15–20. ACM, 2008. doi: 10.1145/1517494.1517497.

[80] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[81] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus, 2018. URL `https://www.agner.org/optimize/instruction_tables.pdf`.

[82] J. Gan, R. Kok, P. Kohli, Y. Ding, and B. Mah. Using virtual machine protections to enhance whitebox cryptography. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 17–23, May 2015. doi: 10.1109/SPRO.2015.12.

[83] Z. Gao, N. Desalvo, P. D. Khoa, S. H. Kim, L. Xu, W. W. Ro, R. M. Verma, and W. Shi. Integrity protection for big data processing with dynamic redundancy computation. In *Autonomic Computing (ICAC), 2015 IEEE International Conference on*, pages 159–160, July 2015. doi: 10.1109/ICAC.2015.34.

[84] Peter Garba and Matteo Favaro. Saturn - software deobfuscation framework based on llvm. In *Proceedings of the 3rd ACM Workshop on Software Protection*, SPRO '19, pages 27–38, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368353. doi: 10.1145/3338503.3357721.

[85] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 193–206. ACM, 2003.

[86] Sanjam Garg, Craig Gentry, Shai Halevi, Mariana Raykova, Amit Sahai, and Brent Waters. Candidate indistinguishability obfuscation and functional encryption for all circuits. *SIAM Journal on Computing*, 45(3):882–929, 2016. doi: 10.1137/14095772x.

[87] Sudeep Ghosh, Jason D Hiser, and Jack W Davidson. A secure and robust approach to software tamper resistance. In *International Workshop on Information Hiding*, volume 6387 LNCS, pages 33–47. Springer, 2010. ISBN 364216434X. doi: 10.1007/978-3-642-16435-4_3.

[88] Sudeep Ghosh, Jason Hiser, and Jack W Davidson. Software protection for dynamically-generated code. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, PPREW '13, page 1, New York, NY, USA, 2013. ACM, ACM. ISBN 978-1-4503-1857-0. doi: 10.1145/2430553.2430554.

[89] Jonathon T Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening software self-checksumming via self-modifying code. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 10–pp. IEEE, 2005.

[90] Kenneth Goldman, Reiner Sailer, Dimitrios Pendarakis, and Deepa Srinivasan. Scalable integrity monitoring in virtualized environments. In *Proceedings of the fifth ACM workshop on Scalable trusted computing*, pages 73–78. ACM, 2010.

[91] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, pages 2:1–2:6, New York, NY, USA, 2017. ACM, ACM. ISBN 978-1-4503-4935-2. doi: 10.1145/3065913.3065915.

[92] Serge Guelton, Adrien Guinet, Pierrick Brunet, Juan Manuel Martinez Caamaño, Fabien Dagnat, and Nicolas Szlifierski. [research paper] combining obfuscation and optimizations in the real world. *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 24–33, 2018.

[93] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, pages 3–14. IEEE, December 2001. doi: 10.1109/WWC.2001.990739.

[94] Vivek Haldar, Deepak Chandra, and Michael Franz. Semantic remote attestation: a virtual machine directed approach to trusted computing. In *USENIX Virtual Machine Research and Technology Symposium*, volume 2004, 2004.

[95] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pages 1025–1035, USA, 2017. Curran Associates Inc. ISBN 978-1-5108-6096-4. URL `http://dl.acm.org/citation.cfm?id=3294771.3294869`.

[96] Kelly Heffner and Christian Collberg. The obfuscation executive. In *International Conference on Information Security*, pages 428–440. Springer, 2004.

[97] Bill Horne, Lesley Matheson, Casey Sheehan, and Robert Tarjan. Dynamic Self-Checking Techniques for Improved Tamper Resistance. *Security and Privacy in Digital Rights Management*, pages 141–159, 2002. doi: 10.1007/3-540-47870-1_9. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.13.3308`.

[98] Amjad Ibrahim and Sebastian Banescu. Stins4cs: A state inspection tool for c. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 61–71. ACM, 2016.

[99] Matthias Jacob, Mariusz H Jakubowski, and Ramarathnam Venkatesan. Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In *Proceedings of the 9th workshop on Multimedia & security*, pages 129–140. ACM, 2007.

[100] Markus Jakobsson and Karl-Anders Johansson. Retroactive detection of malware with applications to mobile platforms. In *Proceedings of the 5th USENIX Conference on Hot Topics in Security*, HotSec'10, pages 1–13, Berkeley, CA, USA, 2010. USENIX Association.

[101] Markus Jakobsson and Karl-Anders Johansson. Practical and secure software-based attestation. In *Lightweight Security & Privacy: Devices, Protocols and Applications (LightSec), 2011 Workshop on*, pages 1–9. IEEE, 2011.

[102] Hongxia Jin and Jeffery Lotspiech. Forensic analysis for tamper resistant software. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 133–142. IEEE, 2003.

[103] Pontus Johnson, Alexandre Vernotte, Mathias Ekstedt, and Robert Lagerström. pwnpr3d: an attack-graph-driven probabilistic threat-modeling approach. In *Availability, Reliability and Security (ARES), 2016 11th International Conference on*, pages 278–283. IEEE, 2016.

[104] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm: software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the 1st International Workshop on Software Protection*, pages 3–9. IEEE Press, IEEE, 2015. doi: 10.1109/SPRO.2015.10.

[105] Teemu Kanstrén, Sami Lehtonen, Reijo Savola, Hilkka Kukkohovi, and Kimmo Hätönen. Architecture for high confidence cloud security monitoring. In *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pages 195–200. IEEE, 2015.

[106] Arun K Kanuparthi, Mohamed Zahran, and Ramesh Karri. Architecture support for dynamic integrity checking. *IEEE Transactions on Information Forensics and Security*, 7 (1):321–332, 2012.

[107] Yuichiro Kanzaki, Clark Thomborson, Akito Monden, and Christian Collberg. Pinpointing and hiding surprising fragments in an obfuscated program. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, PPREW-5, pages 8:1–8:9, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3642-0. doi: 10.1145/2843859.2843862.

[108] Nikolaos Karapanos, Alexandros Filios, Raluca Ada Popa, and Srdjan Capkun. Verena: End-to-end integrity protection for web applications. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2016.

[109] Stamatis Karnouskos. Stuxnet worm impact on industrial cyber-physical system security. In *IECON 2011-37th Annual Conference on IEEE Industrial Electronics Society*, pages 4490–4494. IEEE, 2011. doi: 10.1109/iecon.2011.6120048.

[110] Kaspersky Lab. Projectsauron: top level cyber-espionage platform covertly extracts encrypted government comms, 2017. URL `https://securelist.com/75533/faq-the-projectsauron-apt/`.

[111] D. Kelly, C. Wellons, J. Coffman, and A. Gearhart. Automatically validating the effectiveness of software diversity schemes. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Supplemental Volume (DSN-S)*, pages 1–2, June 2019. doi: 10.1109/DSN-S.2019.00006.

[112] Chongkyung Kil. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. *IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 115–124, 2009.

[113] Chongkyung Kil, Emre Can Sezer, Peng Ning, and Xiaolan Zhang. Automated security debugging using program structural constraints. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 453–462. IEEE, 2007.

[114] Gene H Kim and Eugene H Spafford. Experiences with tripwire: Using integrity checkers for intrusion detection. 1994.

[115] William B Kimball and Rusty O Baldwin. Emulation-based software protection, October 2012. US Patent 8,285,987.

[116] Johannes Kinder. Towards static analysis of virtualization-obfuscated binaries. In *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pages 61–70. IEEE, 2012.

[117] Barbara Kordy, Piotr Kordy, Sjouke Mauw, and Patrick Schweitzer. Adtool: security analysis with attack–defense trees. In Kaustubh Joshi, Markus Siegle, Mariëlle Stoelinga, and Pedro R. D'Argenio, editors, *International Conference on Quantitative Evaluation of Systems*, pages 173–176, Berlin, Heidelberg, 2013. Springer, Springer Berlin Heidelberg. ISBN 978-3-642-40196-1.

[118] Aniket Kulkarni and Ravindra Metta. A new code obfuscation scheme for software protection. In *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*, pages 409–414. IEEE, 2014.

[119] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[120] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside sgx enclaves with branch shadowing. In *26th USENIX Security Symposium, USENIX Security*, pages 16–18, 2017.

[121] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979. ISSN 0164-0925. doi: 10.1145/357062.357071.

[122] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1 (1):121–141, 1979.

[123] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. Authenticated index structures for aggregation queries. *ACM Transactions on Information and System Security (TISSEC)*, 13(4):32, 2010.

[124] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th Conference on Symposium on Operating*

Systems Design & Implementation - Volume 6, OSDI'04, page 9, USA, 2004. USENIX Association.

[125] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 285–298, 2017.

[126] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[127] Han Liu, Chengnian Sun, Zhendong Su, Yu Jiang, Ming Gu, and Jiaguang Sun. Stochastic optimization of program obfuscation. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 221–231, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-3868-2. doi: 10.1109/ICSE.2017.28.

[128] Lannan Luo, Yu Fu, Dinghao Wu, Sencun Zhu, and Peng Liu. Repackage-proofing android apps. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 550–561. IEEE, 2016.

[129] Symphony Luo and Peter Yan. Fake apps: Feigning legitimacy. Technical report, Trend Micro, 2014. URL `http://www.trendmicro.de/cloud-content/us/pdfs/security-intelligence/white-papers/wp-fake-apps.pdf`.

[130] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In Eran Yahav, editor, *Static Analysis*, pages 95–111, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-23702-7.

[131] Matias Madou, Bertrand Anckaert, Patrick Moseley, Saumya Debray, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *International Workshop on Information Security Applications*, pages 194–206. Springer, 2005.

[132] Corey Malone, Mohamed Zahran, and Ramesh Karri. Are Hardware Performance Counters a Cost Effective Way for Integrity Checking of Programs. *Proceedings of the sixth ACM workshop on Scalable trusted computing - STC '11*, page 71, 2011. ISSN 1543-7221. doi: 10.1145/2046582.2046596. URL `http://www.scopus.com/inward/record.url?eid=2-s2.0-80755143408{&}partnerID=40{&}md5=ad5db1f8e5c0131a2a17f457ba1b0497$\delimiter"026E90F$nhttp://dl.acm.org/citation.cfm?doid=2046582.2046596`.

[133] Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Conqueror: Tamper-proof code execution on legacy systems. *Lecture Notes in Computer Science (including subseries*

Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6201 LNCS: 21–40, 2010. doi: 10.1007/978-3-642-14215-4_2.

[134] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[135] Jan M. Memon, Asma Khan, Amber Baig, and Asadullah Shah. A study of software protection techniques. In Tarek Sobh, editor, *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, pages 249–253, Dordrecht, 2007. Springer Netherlands. ISBN 978-1-4020-6268-1.

[136] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. ISBN 978-3-540-48184-3.

[137] Benoît Morgan, Eric Alata, Vincent Nicomette, Mohamed Kaâniche, and Guillaume Averlant. Design and implementation of a hardware assisted security architecture for software integrity monitoring. In *Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium on*, pages 189–198. IEEE, 2015.

[138] Gideon Myles and Hongxia Jin. A metric-based scheme for evaluating tamper resistant software systems. In Kai Rannenberg, Vijay Varadharajan, and Christian Weber, editors, *IFIP International Information Security Conference*, pages 187–202, Berlin, Heidelberg, 2010. Springer, Springer Berlin Heidelberg. ISBN 978-3-642-15257-3.

[139] Jasvir Nagra and Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[140] Ricardo Neisse, Dominik Holling, and Alexander Pretschner. Implementing trust in cloud infrastructures. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 524–533. IEEE Computer Society, 2011.

[141] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, pages 445–451, November 2016. doi: 10.1109/CIC.2016.065.

[142] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Inf. Process. Lett.*, 49(1):9–14, 1994.

[143] Mathilde Ollivier, Sébastien Bardin, Richard Bonichon, and Jean-Yves Marion. How to kill symbolic deobfuscation for free (or: Unleashing the potential of path-oriented protections). In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 177–189, New York, NY, USA, 2019. Association for Computing

Machinery. ISBN 9781450376280. doi: 10.1145/3359789.3359812. URL https://doi.org/10.1145/3359789.3359812.

[144] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. IEEE, 2012.

[145] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Transparent rop exploit mitigation using indirect branch tracing. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 447–462, 2013.

[146] Sungjin Park, Jae Nam Yoon, Cheoloh Kang, Kyong Hoon Kim, and Taisook Han. Tgvisor: A tiny hypervisor-based trusted geolocation framework for mobile cloud clients. In *Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on*, pages 99–108. IEEE, 2015.

[147] U. Piazzalunga, P. Salvaneschi, F. Balducci, P. Jacomuzzi, and C. Moroncelli. Security strength measurement for dongle-protected software. *IEEE Security Privacy*, 5(6): 32–40, November 2007. ISSN 1540-7993. doi: 10.1109/MSP.2007.176.

[148] Stephen Pohlig and Martin Hellman. An improved algorithm for computing logarithms over GF(2) and its cryptographic significance. *Information Theory, IEEE Transactions on*, 24(1):106–110, 1978.

[149] M. Protsenko, S. Kreuter, and T. MÃŒller. Dynamic self-protection and tamperproofing for android apps using native code. In *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, pages 129–138, August 2015. doi: 10.1109/ARES.2015.98.

[150] Jing Qiu, Babak Yadegari, Brian Johannesmeyer, Saumya Debray, and Xiaohong Su. Identifying and understanding self-checksumming defenses in software. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 207–218. ACM, 2015.

[151] CPP Reference. RAII resource acquisition is initialization. https://en.cppreference.com/w/cpp/language/raii/ CPP Reference, 2018. Accessed: 2018-09-06.

[152] Rolf Rolles. Unpacking virtualization obfuscators. In *3rd USENIX Workshop on Offensive Technologies.(WOOT)*, 2009.

[153] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[154] Aleieldin Salem and Sebastian Banescu. Metadata recovery from obfuscated programs using machine learning. In *Proceedings of the 6th Workshop on Software Security,*

Protection, and Reverse Engineering, SSPREW '16, page 1, New York, NY, USA, 2016. ACM, ACM. ISBN 978-1-4503-4841-6. doi: 10.1145/3015135.3015136.

[155] Gerald Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988. ISSN 0306-4573. doi: 10.1016/0306-4573(88)90021-0. URL `http://www.sciencedirect.com/science/article/pii/0306457388900210`.

[156] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: From virtualized code back to the original. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 372–392. Springer, 2018.

[157] Michael Schmidt and Hod Lipson. Distilling free-form natural laws from experimental data. *science*, 324(5923):81–85, 2009.

[158] Bruce Schneier. Story of the greek wiretapping scandal, 2007. URL `https://www.schneier.com/blog/archives/2007/07/story_of_the_gr_1.html`. `https://www.schneier.com/blog/archives/2007/07/story_of_the_gr_1.html/`.

[159] B. Scholz, C. Zhang, and C. Cifuentes. User-input dependence analysis via graph reachability. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 25–34, 2008.

[160] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1), April 2016. ISSN 0360-0300. doi: 10.1145/2886012.

[161] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems. *ACM SIGOPS Operating Systems Review*, 2005. ISSN 0163-5980. doi: 10.1145/1095809.1095812. URL `http://dl.acm.org/citation.cfm?id=1095809.1095812`.

[162] Hovav Shacham, E Buchanan, R Roemer, and S Savage. Return-oriented programming: Exploits without code injection. *Black Hat USA Briefings (August 2008)*, 2008.

[163] Jiri Slaby, Jan Strejček, and Marek Trtík. Symbiotic: Synergy of instrumentation, slicing, and symbolic execution. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 630–632, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-36742-7.

[164] Diomidis Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security*, 3(1):51–62, 2000. ISSN 1094-9224. doi: 10.1145/353323.353383.

[165] Diomidis Spinellis. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):51–62, 2000.

[166] Amitabh Srivastava, Andrew Edwards, and Hoi Vo. Vulcan: Binary transformation in a distributed environment. Technical report, technical report msr-tr-2001-50, microsoft research, 2001.

[167] Q. Su, F. He, N. Wu, and Z. Lin. A method for construction of software protection technology application sequence based on petri net with inhibitor arcs. *IEEE Access*, 6:11988–12000, 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2018.2812764.

[168] Yulei Sui and Jingling Xue. Svf: Interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 265–266, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4241-4. doi: 10.1145/2892208.2892235.

[169] Yuqiong Sun, Susanta Nanda, and Trent Jaeger. Security-as-a-service for microservices-based cloud applications. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 50–57. IEEE, 2015.

[170] B. De Sutter, P. Falcarin, B. Wyseur, C. Basile, M. Ceccato, J. DAnnoville, and M. Zunke. A reference architecture for software protection. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 291–294, April 2016. doi: 10.1109/WICSA.2016.43.

[171] Fernando A Teixeira, Gustavo V Machado, Fernando MQ Pereira, Hao Chi Wong, José Nogueira, and Leonardo B Oliveira. Siot: securing the internet of things through distributed system analysis. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 310–321. ACM, 2015.

[172] Nikolai Tillmann and Peli de Halleux. Pex - white box test generation for .net. In *Proc. of Tests and Proofs (TAP'08)*, volume 4966, pages 134–153, Prato, Italy, April 2008. Springer Verlag. URL `https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/`.

[173] Frank Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[174] Flavio Toffalini, Martín Ochoa, Jun Sun, and Jianying Zhou. Careful-packing: A practical and scalable anti-tampering software protection enforced by trusted computing. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, page 231–242, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450360999. doi: 10.1145/3292006.3300029. URL `https://doi.org/10.1145/3292006.3300029`.

[175] Ramtine Tofighi-Shirazi, Irina-Mariuca Asavoae, Philippe Elbaz-Vincent, and Thanh-Ha Le. Defeating opaque predicates statically through machine learning and binary analysis. In *Proceedings of the 3rd ACM Workshop on Software Protection*, SPRO'19, page 3–14, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368353. doi: 10.1145/3338503.3357719.

[176] Sid-Ahmed-Ali Touati and Denis Barthou. On the decidability of phase ordering problem in optimizing compilation. In *Proceedings of the 3rd conference on Computing frontiers*, pages 147–156. ACM, 2006.

[177] Paul C Van Oorschot, Anil Somayaji, and Glenn Wurster. Hardware-assisted circumvention of self-hashing software tamper resistance. *IEEE Transactions on Dependable and Secure Computing*, 2(2):82–92, 2005.

[178] Michael Velten and Frederic Stumpf. Secure and privacy-aware multiplexing of hardware-protected tpm integrity measurements among virtual machines. In *International Conference on Information Security and Cryptology*, pages 324–336. Springer, 2012.

[179] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 193–202. IEEE, 2001.

[180] Ping Wang, Seok-kyu Kang, and Kwangjo Kim. Tamper Resistant Software Through Dynamic Integrity Checking. In *Proc. Symp. on Cyptography and Information Security (SCIS 05)*, 2005. ISBN 8242866627.

[181] Michael G Wrighton and André M DeHon. Sat-based optimal hypergraph partitioning with replication. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 789–795. IEEE Press, 2006.

[182] Glenn Wurster, P. C. Van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. *Proceedings - IEEE Symposium on Security and Privacy*, pages 127–135, 2005. ISSN 1081-6011. doi: 10.1109/SP.2005.2.

[183] Glenn Wurster, P. C. Van Oorschot, and Anil Somayaji. A generic attack on checksumming-based software tamper resistance. *Proceedings - IEEE Symposium on Security and Privacy*, pages 127–135, 2005. ISSN 1081-6011. doi: 10.1109/SP.2005.2.

[184] Brecht Wyseur. *White-Box Cryptography*, pages 1386–1387. Springer US, Boston, MA, 2011. ISBN 978-1-4419-5906-5. doi: 10.1007/978-1-4419-5906-5_627.

[185] Brecht Wyseur and Bjorn De Sutter. D1.04 Reference Architecture. 2014. URL `https://aspire-fp7.eu/sites/default/files/D1.04-ASPIRE-Reference-Architecture-v2.1.pdf`.

[186] Mi Xianya, Zhang Yi, Wang Baosheng, and Tang Yong. A survey of software protection methods based on self-modifying code. In *Computational Intelligence and Communication Networks (CICN), 2015 International Conference on*, pages 589–593. IEEE, 2015.

[187] W. Xiong, A. Schaller, S. Katzenbeisser, and J. Szefer. Software protection using dynamic pufs. *IEEE Transactions on Information Forensics and Security*, 15:2053–2068, 2020.

[188] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 674–691. IEEE, 2015.

[189] Fangzhou Yao, Read Sprabery, and Roy H Campbell. Cryptvmi: a flexible and encrypted virtual machine introspection system in the cloud. In *Proceedings of the 2nd international workshop on Security in cloud computing*, pages 11–18. ACM, 2014.

[190] Raz Ben Yehuda and N. Zaidenberg. Protection against reverse engineering in arm. *International Journal of Information Security*, 19:39–51, 2019.

[191] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *2013 IEEE Symposium on Security and Privacy*, pages 559–573, May 2013. doi: 10.1109/SP.2013.44.

A. Brief Summaries of Selected Related Work

This chapter presents brief summaries of selected related work in the field of software integrity protection. This survey is the basis upon which we build a taxonomy for software integrity protection techniques.

A.1. Overview

We conduct a literature survey to review state of the art in the field of software integrity protection. When applicable, we bear a critical analysis of the papers in a separate paragraph prefixed with "Remarks". For some articles, we propose potential improvements in an "Idea" paragraph. Please note that our review was closed on October 1st, 2020.

A.2. Survey of papers

- Integrity Verification Kernel & Interlocking Trust Mechanism [15]
  This approach, being one of the oldest and the most cited integrity protection techniques, utilizes code block encryption and signature matching to mitigate tampering attempts. The author refers to the integrity checking codes as *Integrity Verification Kernel (IVK)*. These IVKs, for better stealth, are interleaved with the program functionality. Using this technique, the entire application code except for the starting block is encrypted in a way that only valid (untampered) execution paths can decrypt their next block. The decryption key of each block is obtained from the trace leading to it. A pseudo-random generator generates a key from the trace hash. Furthermore, each block has to re-encrypt its caller (decryptor), so that at a time only one block is in plain text. Reencryptions make dynamic program analysis more difficult for an adversary.

  Another security measure in this approach is a system-wide security service so-called *Interlocking Trust Mechanism (ITK)*. ITK verifies the integrity of IVKs in the system. A set of IVKs protects ITM's integrity.

  Remarks: The paper comes with no performance evaluations, so the upper-bound of the overhead is not clear. It is not wrong to assume large overheads due to the constant encryption and decryption at runtime. Also, it is not clear how multiple paths to a block (traces) could lead to a consistent decryption key.

- Use of Hardware Performance Counters (HPC) [132]
  In this approach, a program is protected in two phases. In phase one, the program is executed and based on the hardware performance counters, static information about event function calls, number of branches, floating points, IO, and number of instructions are recorded.

  Later on, using Eurequa (a mathematical tool) [157] hidden relations between previously captured data are discovered and stored. In phase two (at runtime), the captured values (in the magnitude of kilobytes) are compared to new event values and evaluated using the captured mathematic relations. Since the captured values may differ from one run to another, a threshold is introduced. Only if the relations with a certain threshold at runtime do not hold, the program's integrity is assumed to be violated. In their experiments, they set the threshold value to 5%. Their evaluation results indicated an overhead of 10%.

  This approach can be seen as a complement for TPM based techniques in which only the integrity files are secured but not the runtime. In such cases, the expected mathematical relations can be secured using the TPM.

  Remarks: Setting the threshold value is critical because a considerable threshold value may fail to detect a substantial number of tampering attacks. On the other hand, a small number may issue false alarms. Another significant limitation is this technique cannot be used in a multitasking environment, which severely limits the applications of the method.

- Dynamic Program Monitoring [31]
  In this work, the authors proposed a tamper detection technique based on Control Flow Integrity (CFI) using two processes. A primary process (P) and a monitoring process (M). The two collaborate. The communication is applied to the target program at compile time (implemented in GCC) in which P is modified such that it reports its control flow to M at runtime. M has to be aware of the correct control flow and respond to the reports accordingly. That information is also at compile time shipped into M. The authors recommend protecting M intensive tamper-proofing techniques. Since M is a compact application, it introduces less performance overhead as opposed to intensively protecting P.

  Idea: a possible direction I see here is to apply a similar technique but with the help of SGX. M will reside inside the enclave, so it is tamper protected.

  Remarks: The attackers can forge the control flow reports to M. Such forges can be quite easy as there is no integrity protection on the P side.

- Dynamic Kernel Integrity Checking [41]
  In this work, a method is proposed to inspect up to 99% of kernel's dynamic data and detect malicious tampering attempts. They evaluated their technique against nine existing kernel malware, all of which were detected successfully. Main features are **a)**

apply inter-procedural points **b)** resolve type ambiguity **c)** recognize dynamic array boundaries. The technique leverages static code analysis for initiation and memory analysis for detection.

- Data flow integrity [42]
  This work puts its focus on the input data to prevent well-known attacks such as buffer-over-flow and the like. The assumption here is to assure the input data will not corrupt memory, and subsequently, the attack is prevented. This work targets category one attacks. This method instruments the running process and verifies the data flow using a (previously captured) data model at runtime. Static analysis is used to capture the data flow graph.

- Reflection for Software Integrity Verification [164]
  This work relies on an external (trusted) entity to verify software integrity. The idea is to retrieve the program's state using reflection and report it to a trusted party who can verify it. To improve tamper resistance, they proposed to exchange CPU performance counters of the state preparation period along with the actual state values.

  Remarks: One obvious attack is to keep an untouched version of the application in the memory next to the tampered version and compute all hash computations to the good version. The authors suggested using memory expansion and timing as possible countermeasures.

- Dynamic self-checking [97]
  In this work, the authors proposed a dynamic self-checksumming technique that utilizes a collection of testers to verify random intervals in the executable code pages. The testers are added to the post compilation process. Their approach only handles linear checks so, and no circular checks are supported. In this event, an interval is checked only by one block. A 32bit space is added outside basic blocks as the corrector that tries to fix the hash values in the patch process. The patch process is part of the software watermarking after-installation process.

- Oblivious Hashing [49]
  OH refers to a dynamic tamper resistance mechanism that (instead of code bytes) incorporates instructions, memory access, initial code counter, and memory values and computes a hash over them. The term oblivious refers to the fact that adversaries are oblivious to the fact that programs are computing hashes at runtime. In this work, they implemented OH at a higher level of representation instead of binary-level object code. The reason is that, in their case, low-level code injection requires maintaining a hash value in a particular register. This quickly becomes problematic because we need to keep pushing and popping the corresponding register value. This, in turn, eases the detection of hash variables for adversaries. Therefore, to maintain the correctness of the application, speed up the instrumentation, and leverage from compiler interleaving hash code with the program code through optimization and benefiting from the

machine-independent protection method, they injected the hashing into the syntax tree during the compilation process. They implemented hashing in C Intermediate Language (CIL). Assignments, expressions, and control flows are used as program features for hash computations. In the intermediate level, expressions are transformed to a comma operator (e.g. $a = exp$ is transformed to $a = (t = exp, hash(t), t)$). Control flows are verified by adding one or two hashing instruction in the basic blocks.

As another application of oblivious hashing, the authors proposed to use their technique as a remote authentication means between a client and a server. They argue such verification provides more reliable guarantees compared to remote static code checksumming. This approach, although promising, comes with major limitations. Using this technique, we can only hash a fixed portion of the code that is input agnostic. Otherwise, with different runs of the program, the hash values will be different, and thus we cannot verify the application behavior. To cope with this limitation, they suggest running multiple test cases to capture deterministic parts of the code. **Remarks:** In general, It is tedious to use oblivious hashing as a means for tamper resisting software. Because of three reasons: **i)** pre-computation of the correct hashes is hard, **ii)** the protection highly depends on the path coverage, and **iii)** large portions of programs depends on nondeterminism (i.e., they are input dependent) and thus cannot be protected. Another possible attack against this scheme might be to identify the verification instructions and to patch them in the binary. If a matching pattern could be recognized by the attacker, this can be done without any effect on the program execution.

- Oblivious Hashing Using Overlapped Instructions [99]
  In this method, an initialized hash value is continuously updated with the program assignments and branch transfers. At various points, the hash value is verified at runtime. They continuously keep rehashing a given value with a previously initialized hash variable. In the case of branches, the same thing is done with a unique basic block id. The main contribution lies in the two overlapping technique for stealth improvement that are introduced and discussed in technical details, namely *interleaved basic blocks* and *Instruction outlining*.

  Interleaved basic blocks technique combines a set of basic blocks into one single so-called *super block* with multiple entries and exit points. Within the superblock, a set of cleverly chosen absurd instructions (exploiting x86 architecture's variable instruction size) are injected into the binary. These instructions are kept isolated from the correct (untampered with) execution routine using opaque predicates, which are known to be hard to be detected by static analyzers. If the bogus instructions are cleverly selected, two main benefits are gained: **i)** they can manage to fool disassembler and make reverse engineering harder [126], and **ii)** any tampering attempts potentially trigger faulty instructions in an interleaved control flow and eventually a cause crash in the execution.

The instruction outlining approach attempts to makes the tampering attacks harder by creating multiple dependencies between different control flow branches. In effect, common instructions within a code block or the entire application are outlined (opposite to in-lining code in C) so that minor tampering with attempts would affect multiple disjoint execution branches. Their experiments indicate an overhead of 300%.

- Host Runtime Integrity Checking [43]
  This technique attempts to guarantee the integrity of the standard operating system applications. A compromise-free system is tough to have due to the vulnerabilities white-hat hackers report daily. In this work, the authors emphasize that the window of time from detection of a vulnerability until a patch is installed may allow the attackers to persist their breach by modifications at the system level. Rootkits and code modification are the first groups of attacks to permanently persist a breach. Another approach is to inject malicious libraries into executable binaries dynamically. In the Unix system, some features of the OS are provided in the form of Python or Perl scripts, e.g., remote package installer. A malicious modification of these scripts may enable an adversary to keep the system infected for a more extended period. In this type of attack, after a successful compromise, the host might not be able to detect modifications in the system. For example a rootkited OS may selectively hide system states information retrieved by *ls* or *ifconfig* commands.

  This scheme assures ELF and script files' integrity via runtime checking. As a prerequisite, a trusted authority has to sign all packages. OS will be modified to verify signatures of each downloaded library before attempting to install them. For executable scripts, three kernel-level verifications are implemented, namely `elfsign`, `scriptsign` and `verify`. The verification is initiated whenever a file is loaded into memory at the kernel level. For libraries, this is handled by hooking `uselib` and `dlopen` system calls. ELF signature verification is capable of verifying program dynamic dependencies (using a recursive approach). The signing keys are handled using a key management component reachable by an in-kernel API call. Key management supports multiple keys, at verification time the right key is retrieved by a key-id meta-data appended/prepended to files.

  Remarks: The performance of this approach firmly depends on the executable characteristics, e.g., the number of dynamic libraries dependencies. Since their experiment results are not comprehensive, it is difficult to judge the performance overhead.

- Virtual Machine Introspection [55]
  This method focuses on dynamic integrity checking for virtual machines, which are commonly used in cloud computing. The authors introduce a mechanism for the integrity checking of a guest VM from a virtual machine monitor (VMM) without requiring VMs to go through a secure boot process. Although the work claims to be OS agnostic protection approach, on the contradictory, this technique requires that all supported operating systems to be examined in a so-called lab phase. In the

lab phase, a white list of benign applications along with their signature is generated. Furthermore, the tool maintains a dataset of up-to-date malware. This dataset serves as a blacklist and compliments the previously captured white list data. At runtime, first, the OS is detected (using the lab data), and then its state is verified using both whitelist and blacklist data. The verification is done with the help of an uncompromisable (assumption) hypervisor and a set of unbreakable (assumption) security virtual machines. Unfortunately, these two assumptions are hard to hold true in practice. Speaking of technicalities, they did not address how the *randomized access memory layout* is regarded at runtime. Given that on each run application's memory address may change, how are the memory addresses of the known applications detected at runtime? They reported an overhead of 2% in their performance evaluations.

Remarks: Whitelisting, at its best, can only verify signatures of the stored binaries. It immediately becomes ineffective when only the program is tampered with dynamically. Likewise, the blacklisting approach cannot detect all malware for the same reason that anti-virus applications failed so far.

- Remote Integrity Checking [73]
  Internet-facing servers are always dealing with the risk of being infected by malware, thus the integrity protecting such servers is crucial. In this work, the goal is to enable a verifier to test the integrity of files residing on a remote server. For this purpose, a simple solution might be to reset the servers using a secure boot and verify system integrity according to expected hash values. Such a technique is impractical as it causes downtime. An alternative is to verify the integrity of files remotely through an integrity checking agent. However, it is risky to rely on such calculations as the agent itself is susceptible to modifications. Adversaries can potentially alter the agent such that it always reports good hash values. A countermeasure is to use a challenge-based protocol in which a randomly-generated challenge needs to be incorporated in hash computations. This forces the verifier to keep a copy of all files or to compute the hash for a set of prepared challenges. Under the assumption that the attacker will not know the possible challenges, she always has to compute the value using the untampered file. However, a challenge-based approach can also be defeated if the attacker maintains both tampered and untampered version of all files and respond to challenges using the unchanged data. The authors suggest this sort of activity can be detected by a specially tuned intrusion detection system. Clearly, for an efficient precomputed challenge list, the following equation needs to hold:

$$|precomputed_{challenges}| \times verification_{interval} \geq server_{lifetime}$$

  The authors, in the second part of their work, introduce a Diffie-Hellman-based protocol for remote verifications in which they user file fingerprints as secrets.

  Remarks: Given that the attacker can know the file fingerprint (the secret), we do not see any benefit of their Diffie-Hellman based verification.

- Tamper-proofing using control flow flattening [104]
  This approach develops control flow flattening obfuscation technique equipped with a tamper-resistance mechanism based on LLVM toolchain[1]. Control flow flattening is a technique that makes control graph analysis harder by merging all code branches into a broad switch case statement. Their approach consists of two main functions, namely *check()* and *respond()*. The former one is responsible for computing a checksum over code regions, whereas the latter one decide whether the calculated value matches the expected result or not. In their article, they state that a random slice of the basic block byte code is hashed at runtime. It is not clear if such slices include absolute addresses or not, this is important because if they do include them, then each instance of the application may change the hash values thanks to the address space layout randomization.

  To accelerate the hashing process, on Intel x86 architecture, the hardware supported *CRC32* instruction is used.

  In this method, the *respond()* function is implicitly implemented in which all checksums are incorporated in the switch case of the control flow flattening mechanism such that any tampering attempts would break the control flow of the application.

- Remote Attestation For Dynamic Properties [112]
  This work stresses that static attestation has limitations given that the object's behavior can be modified at runtime (e.g., buffer overflow). Therefore, static verification is insufficient for integrity protection. However, attesting dynamic properties of a program is challenging because **i)** there are a wide range of dynamic properties that need to be monitored (heap, stack and the like), **ii)** good (untampered) dynamic states may revisit a variety of values over time. Therefore, an approach is needed to verify volatile states periodically. This work extends the TPM-based static integrity checking model with a dynamic monitoring tool. A trusted entity is still needed to be plugged to the system and request the integrity measures periodically. TPM protects the monitoring tools. The tool monitors two dynamic features: structural integrity and global data. Automatic security vulnerability detection in C programs uses structural integrity features [113]. Safety-critical systems flag inconsistencies in the global data as undesired states, e.g., NASA used it to detect problematic states [76]. To add runtime integrity, a program has to go through a dynamic property collection using static and dynamic analysis. The structural constraints are identified by static analysis and focus on identifying dynamic memory mutation patterns. For example, memory allocated in the heap memory using `malloc` should be linked together. Global data constraints are captured using dynamic analysis and data invariant analysis using the Daikon tool [76]. To capture such invariants, they execute programs multiple times (aiming for 100% coverage). Afterward, both data invariants and structural constraints are fed into the monitoring tool.

[1]a software compiler suite, `www.llvm.org`

At runtime, the monitoring tool (which is implemented as a *Linux Loadable Kernel Module*), upon system calls asserts, the data invariants and dynamic memory structure with some optimization tricks. For instance, memory-related constraints are verified when relevant system calls are triggered. Dynamic objects within constructor and destructor are only verified before the main function gets executed.

Upon any detection of integrity violation, the monitoring tool updates TPM's PCR8 with violation evidence. Later on, when the trusted entity inquires, these violation lists, along with the up-to-date system measurements are sent back.

The authors evaluated their technique using a set of applications that are known to have dynamic vulnerabilities. After plugging the integrity monitoring tool, they compromised the known vulnerability. According to their results, the tool managed to detect 100% of attacks with zero false positives. They reported an overhead of 8%.

- File System Integrity Checking [114]
  This paper implements the Tripwire tool that starts with collecting integrity measures upon a filesystem. Subsequently, it monitors the files, their properties, and reports any deviations to dedicated administrators. The tool comes with a possibility to configure the monitoring process fine-grained, e.g., to ignore certain inoffensive modifications in the file system.

  Remarks: The success of this is highly dependent on the time when the integrity measures are collected, if the system at that time is already infected, then Tripwire may start with a faulty state. The system assumes that the *sys admin* is trustworthy, and falls short of protecting the tool itself from malicious modifications.

- Dynamic Self-Protection for Android [149]
  This work reports on an implementation of a self-encrypting code for Android application that performs at the intermediate level, i.e., between loading the native code and the application code. The authors used Soot (a framework for native Android code analysis and transformation) F-Droid (an open-source android repository) and Mandelbrot map and 0xbench to develop and evaluate their technique. They reported a performance overhead of $\approx$500%.

  Remarks: Besides the significant overheads, the approach suffers from three limitations: i) reflection is not supported, i.e., functions that use reflections cannot be protected; ii) after the protection, service provider contracts are suppressed and thus no longer available due to the encryption; and iii) the protection takes place at the java dex files, and therefore, the native byte code is not protected.

- StIns4CS: A State Inspection Tool for C# [98]
  The goal of this work is to verify software integrity by exercising its functionality at runtime continuously. The protection process consists of two phases: preparation and monitoring. In the preparation phase, using symbolic execution (the PEX tool) a set of possible inputs and expected outputs for different functions are generated. However,

users (developers) can specify which functions they wish to protect via annotations. The preference is respected in the preparation phase. Once the input and expected outputs are generated the program code base (functions that are to be protected) is augmented with runtime assertions. Assertions verify the program returns the expected output for the given input. These assertions are not to be confused with conditions, but rather, they call the protected function with the given input and assure the expected value is returned. Assertions are placed in a way such that they form a network together. Like that, they can cover each other, and so patches can be detected circularly. This approach can produce tamper-resistance software against static and dynamic patches. However, performance downgrades depend on the size of the protection networks.

Remarks: Checkers execute parts of the program. Triggering functions with side effects may change the program logic. Consequently, the proposed protection can strictly protect program functions with no side effects (i.e., pure functions) can be protected.

- Software Protection Through Dynamic Code Mutation [131]
  This work aims to protect software against reverse engineering attacks by a dynamic code obfuscation technique. Dynamic code mutation attempts to keep the code unreadable until right before execution. After execution, the scheme re-mutates the code for better protection. The underlying assumption is that reverse engineering is more difficult when **i)** program code and data are indistinguishable, and **ii)** there is no fixed mapping between instructions and their residing memory such that multiple code blocks use the same memory regions. The idea in this paper is to modify a program at runtime continually such that memory analysis becomes ineffective. Code blocks are transformed into obfuscated template-like regions filled with random instructions. A system component so-called *edit script* can only decode such templates. The edit script component contains the necessary information needed to transform an obfuscated block into a plain code block. However, another component, *build engine*, undertakes the transformations. Once the build engine transforms the code into the executable code, two types of dynamic mutations one-pass and cluster-based are applied. The one-pass mutation maps an edit script on a single code region, whereas the cluster-based mutation alters the control flow graph on the fly.

  One likely threat to the scheme is that the attacker reverse engineers the edit script and the build engine. If those components are static, an adversary can learn all the necessary information to break the obfuscation and recover the code. The scheme utilizes opaque predicates to prolong such reversing efforts.

  Remarks: Due to the constant relocation of the addresses, this technique cannot be used for dynamic libraries and multi-threaded applications. We also anticipate significant overheads stemming from continuous mutations.

- Tamper Resistance Using Multi-block Hashing [180]

The authors in this protection method propose to encrypt every basic block by deriving a key from the hash of the preceding blocks. In contrast to typical checksumming methods, in this scheme, there is no need for verification of the expected hashes by the verifiers. Instead, each block, if untampered, can be used to derive a key that can decrypt the next block. Since without protections, the program cannot be executed, attackers cannot only remove the protection logic. Any tampering attempt will change the hash of the block and will affect the derived key. With a faulty key, the block will be decrypted to a set of junk instruction that will change the program behavior and eventually terminates the execution. Their experiment results on the gzip application indicate an overhead of 107% overhead with 70 encrypted blocks.

Remarks: There exists blocks reachable from various paths. Each path could lead to the derivation of a different key. The presented approach does not discuss how blocks with multiple entry points can be protected such that decryption from all possible execution paths remains intact. Also, when ASLR is enabled, it is not clear how the program can find out the address where external libraries are loaded.

- Software Protection using Guards [47]
  In this work, the authors proposed a software protection technique for WIN32 (DLL and EXE) applications. The idea is to build up a network of compact protection blocks (aka Guards) that, along with performing integrity checking of the application blocks (basic blocks), also verify their peer guards. The guards constitute a protection network, each protecting parts of the program as well as other guards. The guards can also attempt to heal the tampered with code regions. Since the guards are connected, an attacker has to disable all interconnected nodes to circumvent the protection successfully, which provides a higher level of security.

  In addition to being protected by other guards, each guard should utilize obfuscation techniques for syntactical resemblance ramification. Otherwise, a simple attack can disable all blocks, for example, using pattern matching. Their evaluation results show tolerable overheads. Overheads entirely depend on the number of guards and the circularity degree. Triggering a vast network of protections can impose high costs.

- Hardware-Assisted Software Integrity Checking [137]
  They proposed hardware-aided integrity protection comprised of two main components: trusted hardware and security hypervisor. In this work, program-specific integrity checks are supposed to be input by the user, and the security hypervisor is in charge of carrying them out. The integrity of the hypervisor itself is verified using the external trusted hardware. In addition to that, it is assumed that the trusted hardware has access to the host's memory and can perform environmental attestation on demand. For this purpose, the hypervisor has to run at the highest privilege level. Authors claim if malware were to downgrade the hypervisor's privilege, two problems would arise: **i)** some instructions cannot be executed in downgraded privilege (e.g., retrieving the CPUID) **ii)** the performance of their assertions noticeably suffers

when executed with lower privileges so that a simple timing analysis can detect it.

Remarks: The work comes with no empirical evidence.

- Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems [161]
  This work presents an entirely software-based tamper-proofing technique. In this technique, a remote verifier can attest the integrity of a particular application and verify its execution. The protocol is initiated with a *nonce* that is sent by the server based on which the client is ought to compute a hash of the sensitive code blocks, including the verification protocol itself. This *nonce* will prevent the client from preparing the hash values before the server request. The general assumption is that the remote verifier has a holistic knowledge about the client host, including hardware latency and computation power. This knowledge enables the verifier to perform timing analysis (with a certain threshold) and decide whether an application is genuine or not. Furthermore, clients during the verification shall not communicate with any other hosts so that the outsourcing attack can be prevented. For the same reason, this approach does not apply to clients with multiple cores. Otherwise, additional cores can be utilized to compute the correct hash values in an acceptable time interval.

  The authors have implemented their approach and reported an overhead of 4-5%.

  Remarks: Full knowledge of the client configuration is a privacy issue. This approach, due to its inapplicability on multi-core CPUs, is quite impractical for today's context. Moreover, the requirement to shut the entire system down during the verification has severe usability problems. This work is also known to be vulnerable to virtualization attacks [133].

- Conqueror: Software-Based Remote Attestation for Legacy Systems [133]
  This approach presents a static and dynamic tamper-resistance protection purely software-based. Their attestation protocol receives an entirely new random verification routine (encrypted and obfuscated) on each attestation try. They (based on empirical results) claim their approach is resistant against disassembly and hardware supported hypervisor-based attack. This work claims to beat Pioneer attestation [161], which is known to be vulnerable to hash forging and virtualization attacks.

  Remarks: Conqueror has two limitations: **i)** it only works on single thread execution mode, because attackers can utilize a secondary core to calculate expected hash values; and **ii)** it is unusable when the goal is to verify a guest host with a trusted hypervisor, because Conqueror, by default, considers hypervisor as a sign of an attack.

- Tamper Proofing Using Encryption, Guards and Process-level Virtualization [87]
  This work proposes a combination of checksumming guards, encryption, and visualization techniques to protect software.

At first, using a probabilistic model and based on the user's desired protection level, a set of cyclic protection guards [47] are injected into the binary. These guards later assure the integrity of their protection targets. Afterward, the application's basic blocks, along with their guards, are encrypted, and the encryption key is obscurely placed inside the binary. The key retrieval by an attacker is assumed to be hard guaranteed by the white-box cryptography approach [53]. Then, an emulator is shipped into the binary. The emulator is capable of decrypting the cipher code blocks. That is, each instruction to be executed needs to be fetched, decrypted, and executed by the emulator. To improve the performance, the emulator preserves a cache of the most recent blocks. However, to harden the dynamic analysis attacks, this cache is periodically flushed. The emulator itself utilizes cyclic guard protection. The only difference is that neither the emulator nor the protection guards are encrypted (the starting point has to be in plain-text). The paper explains some heuristics for an optimal number of protection guards for the desired coverage. This work omits some technical details, e.g., it is not clear how the application guards can verify the integrity of basic blocks that are yet to be decrypted. On the positive side, they implemented their technique and carried out experiments using the SPEC2000 suite. Besides the performance overhead, they evaluated two additional characteristics: **i)** the connectivity of the bytes that indicates the number of checks on each program byte and **ii)** the average time delay between a tampering attempt and its detection. According to their results, this technique introduces an overhead of 35% (21% of which being the overhead of the visualization itself). This number, however, is closely tied to the amount of the protection guards and the length of the protected blocks. Concerning the connectivity, 80% of the code bytes are protected by at least four checks. Also, their results showed that the average time between tampering attacks and detections is $\approx$3.5 seconds.

Remarks: They do not discuss the potential impact of ASLR on the protection. A dedicated attack can gradually extract the complete binary in plain-text by constantly dumping the emulator's cache. Although the authors point out that their approach is resistant against the separation of code read and execution pipelines [182], it does not include the VM itself. Precisely, with the assumption of access to the kernel, an attacker can manage to rout emulator's checksums (which are in plain-text) and subsequently disable the cache flushing function.

- White-box AES within Secured Virtual Machines [82]
  This paper proposes a technique for building a virtual machine acting as the root of trust using white-box AES. Several hardening techniques are also utilized to protect the VM. Emulator and debugger detection approaches along with various obfuscation techniques at different levels (from source code to the actual machine code) are employed to thwart static and dynamic analyses. Additionally, to prevent replacement attacks, the secure VM can be bound to the host devices using hardware and software fingerprinting.

As an application of this technique, a system (as a commercial tool) has been implemented that enables general-purpose secure computation (e.g., license checks, authentication verifier and the like) with the help of the embedded VM for mobile clients, namely Android and iOS.

Remarks: The role of white-box AES in protecting the VM is unclear. In their evaluation, they reported an overhead of 128%. This number seems to be acceptable for highly secure and not so frequently used functions(e.g., license check). Nevertheless, they do not indicate precise obfuscation settings nor benchmarked applications.

- Distributed Tamper Detection Via Continuous Software Updates [65]
  The goal of this work is to authenticate client applications by a trusted server remotely. The scheme employs a continuous software diversification technique in which clients are forced to receive frequent updates so that by the time that an active attacker manages to tamper with the client application, it becomes outdated. This technique applies to contexts in which the client and server continually remain connected so that changes can continuously get pushed to the client. They reported 4% to 23% overhead depending on the required frequency of the updates.

  Remarks: Trusted server resembles a single point of failure. Additionally, this approach does not provide any robust integrity verification of the clients. Instead, it makes the code hard for the attacker to understand.

- Trusted Computing for Real-time Embedded Devices [34]
  In this work, the general idea is to design a secure premise that is suitable to be used in embedded devices with real-time requirements. The secure boot is a primary security cornerstone of their approach. Furthermore, the authors propose two types of tasks in this system: **i)** *normal tasks* that are accessible by the OS; and **ii)** *secure tasks* that are isolated from the OS and other tasks. Besides the complete isolation, a secure task comes with three main features: **i)** secure inter-process communication (IPC), **ii)** secure storage and **iii)** attestation.

  The sourcing process loads message m and destination id, d_{id} into CPU registers, and then triggers a secure IPC interrupt. The interrupt ultimately makes m along with the s_{id} accessible for destination process/task. The secure storage feature is a usage of the IPC in which for any given task, a counterpart storage task is initiated. A task-specific key additionally encrypts all communication. The security layer is implemented with the help of EA-MPU. EA-MPU is a hardware-based memory access control that manages task isolation and interrupts handling. The work was implemented on the Intel Siskiyou Peak and FreeRTOS operating system. Interestingly, although some primitives are briefly introduced (e.g., the key derivation), the paper falls short in defining the remote attestation protocol. It is not clear in their evaluation results whether the remote attestation overhead was considered or not. Their results indicate overheads of 300%, 60%, and 30% for secure task creation, context saving, and context restoration, respectively. They conclude their system is acceptably efficient to be used by real-time embedded devices.

Remarks: The security assumption is that the adversary is not inside the system. Thus, the system remains secure as long as attackers fail to compromise EA-MPU based access control system.

- Integrity Protection for Big Data [83]
  A typical Big data architecture (e.g., STORM) comprises a set of computation nodes (workers) that attempt to achieve optimal data processing. In this work, the authors propose a simple approach for process integrity checking by duplicating the computation work so that a malicious node could be detected upon the occurrence of inconsistencies. A set of trusted controller nodes manages the distribution of duplicated work and later on detects mismatches. Controllers maintain a reputation history of the nodes based on their computation mismatches. Later on, this reputation record enables the controllers to take actions against compromised nodes.

  They report on an acceptable overhead yet odd for their approach. It seems the results are somehow, in some cases, outperform the baseline (i.e., no protection).

  Remarks: Their assumption about trusted controllers is pretty unrealistic; the same threats against worker nodes apply to controller nodes as well. And thus, this technique needs to be enhanced with trusted computing protection measures. Utilizing trusted hardware introduces additional overheads. Furthermore, in the current setting, worker nodes can attempt to bypass controller alarm by colluding or predicting ″good results.″

- Trusted Geolocation Service [146]
  In some applications (e.g., GovCloud), servers must verify their clients' locations and vice versa genuinely. However, without a sophisticated protocol, this geographical information can potentially be forged. In this paper, the trusted geolocation problem is tackled using a Trusted Geolocation Server (TGS), a cloud agent, and a tiny hypervisor (XMHF) equipped with a TPM. The tiny hypervisor is self-contained and (without the need for an OS and TPM drivers). The hypervisor can directly communicate with the TPM. Their adversary model counts for the network (i.e., man-in-the-middle) and malicious operating system attacks. They claim their system is secure provided that **i)** the hypervisor is secure (no exploitable vulnerabilities) and **ii)** the GPS device is genuine. The authors have implemented their method and reported an average overhead of 8.3%.

  Remarks: the security of this approach is built upon the idea that hypervisor cannot be dynamically tampered with. However, in practice, this requirement is almost impossible to be attained.

- Architecture for High Confidence Cloud Computing [105]
  Migrating to cloud increases security risks due to a lack of control over data and

infrastructure. In this setting, cloud providers can readily violate users' privacy. In this paper, an approach is proposed to bring confidence in the cloud infrastructures. The method comprises a TPM, a set of attestation probes, a collection of measurement probes, and a monitoring framework. Each physical node must acquire a TPM. The attestation and measurement probes ensure the integrity of the system services and their states at various intervals. The monitoring service reports any inconsistencies to the users. It is noteworthy that this approach supports virtualized environments since via virtual TPM [90, 178]. In the elastic cloud, VMs can be relocated to different physical nodes. This operation can potentially put the system at risk. The proposed scheme utilizes a host identifier in the attestation flow to ensure the system integrity in such relocations. To tolerate faults, instead of conservatively rejecting services that fail to attest themselves, a concept of confidence level is introduced. The idea is to have different levels of trust instead of rejecting or accepting measurement results in a binary manner.

Remarks: The monitoring framework runs inside the cloud provider. This enables the provider to forge results or to avoid reporting compromises. The work provides no comprehensive security analysis. The work is rather conceptual and does not come with any implementation or evaluations.

Idea: Keeping the monitoring framework inside the infrastructure makes the reasoning about good states (hash values) easier than pushing them outside. Maybe the trusted entity (outside cloud provider) could only attest the monitoring service instead of attesting all internal services and their measurements.

- Tamper Resistance Using Nontrivial Code Cloning [118]
  The goal here is to prevent man-at-the-end (MATE) attacks. For this purpose, they introduce a new obfuscation technique that while being labor-intensive (additional development efforts) can harden reverse-engineering attacks. Since this technique can only be applied to the sensitive part of the application (e.g., license checks), it remains usable w.r.t costs.

  Their technique is comprised of four steps: **i)** *logical code fragmentation* in which developers split a sensitive functionality (e.g., license check) into logical fragments; **ii)** *nontrivial fragment clone generation* in which developers implement functionally equivalent but structurally dissimilar clones for sensitive fragments (targetting static program analysis); **iii)** *linking random clone fragments* to deliver a diverse and yet semantically equavalent functionality at runtime (targetting dynamic program analysis), **iv)** *add further resilience by further obfuscations* in which depending on the desired security further obfuscation techniques are utilized.

  This approach introduces a high complexity for dynamic attacks in which K^N paths need to be discovered, N is the number of logical fragments, and K is the number of clones per fragment.

  They implemented their approach on a toy program and reported evaluation results.

Remarks: Developing nontrivial code clones is a daunting task. Furthermore, performance may suffer when less efficient fragments constitute a function. Another foreseeable problem is the risk of race conditions due to the nondeterministic execution times of different fragment clones.

- Security IoT using Distributed System Analysis [171]
  Securing IoT devices is crucial because these devices are more used than ever before. Due to their close interaction with humans, any compromises can potentially endanger the user's life. To cope with limited memory and reduce power consumption, IoT applications are normally developed using C/C++ that are not safe languages. Therefore, out of bound errors are very probable attacks. However, in the context of IoT, the regular bound checking has a high toll on the performance. The authors reported a 70% slowdown and an average additional memory usage of 200%. Such bound checks should only be applied to the user reachable data to avoid unnecessary slowdowns. In a distributed system identifying instructions whose data might be affected by the user input is more challenging. To this end, the authors proposed an approach for abstracting communication between nodes and generating a holistic view of the system (as a standalone application). For this purpose, they introduced *Distributed Control Flow Graph DCFG* and *Distributed Data Flow Graph DDFG* to improve static analysis. In this way, intuitively, the out of bound errors can be identified with less false positives as opposed to individually analyzing each node. They implemented their technique in the LLVM ecosystem(`https://code.google.com/p/ecosoc`).

  Idea: similar approach can be applied to microservices to detect security vulnerabilities with less false positives.

  Remarks: this approach has a complexity in the order of $O = N^2 + 2^N$.

- CryptVMI: Virtual Machine Introspection System in Cloud [189]
  Virtualization enables cloud providers to utilize hardware resources more efficiently by sharing them among various clients with a degree of isolation. In this setting, a rogue virtual machine may bypass the sandboxing and affect other running virtual machines. It is harder to detect such breaches by traditional intrusion detection systems. This paper proposed a *Virtual Machine Introspection (VMI)* technique that inspects memory, CPU registers, and running processes of the running VMs. VMI must reside on every physical node with the root permission. The assumption is that insiders have no root access, and thus, they fail to access VMI. Once VMI is installed on the host, using a VMI Library, it can query the state of the running VMs. All requests and responses are encrypted with the users' key. The authors implemented a proof of concept using open stack and XEN hypervisor. Their evaluation results indicate insignificant overheads.

  Remarks: The VMI itself can become a target for attacks because it has access to all VMs states, including their processes. Running VMI as root in a public cloud is very dangerous and unlikely to be adopted by cloud providers. The assumption that no one has root access is also unrealistic.

- End-to-End Integrity Protection for Web Applications [108]
  This paper argues vulnerable web applications enables attackers to tamper with user's data, computation results, or even access control policies. Therefore, a website can only fulfill end-to-end integrity by assuring access control enforcement, correctness, completeness, and freshness of its operations. To this end, Verena enables application developers to define a set of integrity policies that later can be verified at the user's browser. Their protocol comprises a client, a main server that serves requests, a trusted Identity Provider (IDP) that maintains users' public keys, and finally, a semi-trusted hash server. Semi trusted here means it retains from any collude with the main server, and the assumption is that it cannot be compromised at the same time with the main server. In the beginning, developers have to specify a set of integrity policies at the development time. These policies consist of the group of authorized users so-called trusted context and a set of authorized operations. Besides the plain read, the system supports projection, and aggregation operations are supported. Each trusted context later forms a custom *Authenticated Data Structure (ADS)* that is based on Merkle trees [27] and authenticated index structures [123].

  Only users within a trusted context can write into their corresponding ADSs and they have to sign their requests. IDP is responsible for providing users' public keys to all entities. Upon write requests, users must recompute the root of the target ADS and sign it with their private key. Those values are sent to the hash server for further references. This root computation and signature are necessary to provide freshness to other users.

  To prevent forking attacks [124], users first request the root hash value that must be signed by the hash server private key. In addition to the hash of the root node, the hash server maintains the public key of the last user who modified the file and a counter that indicates the total number of updates (version). This additional information will prevent the main server from serving an older state of the ADS, aka the snapshot attack. The authors implemented their approach and reported efficiency of the CRUD + Aggregations methods, all of which being in the magnitude of milliseconds. However, throughput overheads are quite large, 5x on reads, whereas 8x on writes.

  Idea: this can be generalized for a logging mechanism for microservices. To improve IDP a certificate transparency protocol can be employed on users' public keys.

  Remarks: The trust assumption is that at a time only one of the hash server or main server can be compromised. Furthermore, it is not clear how access control mechanisms are enforced. Although the emphasis is on the detection of tampering attempts, the paper falls short in securing the access that can cause data losses. Another problematic trust anchor is the IDP that can potentially be compromised and result in unnoticed tampering attacks.

- Hypervisor-based System for Protecting Software Runtime Memory [74]
  A malicious OS kernel, by having the highest privilege, can potentially access any program's memory through the linear address space. TPM based attestation, although raising the bar, fail to secure the system completely. The counterexample is a runtime compromise in the kernel due to a vulnerability (after the secure boot). From this point on, a malicious kernel or a compromised driver can forge state hashes and manage to unseal the TPM's secure storage. Additionally, once the system is securely booted, any software can ask TPM for unsealing a blob of encrypted data. Simply put, there is no software-level authentication involved. This work is aiming to enhance this model to protect the application's runtime and conceal application's sensitive data by authenticating the integrity of processes. For this purpose, a lightweight hypervisor (securely initiated with the help of a TPM) is designed to isolate applications' memory from any other processes, including the kernel. The hypervisor resides between the hardware and the OS. It prevents any illegitimate memory access requests and enables authenticated sensitive sealing/unsealing at the application level.

  Their architecture comprises three components (all residing in the hypervisor): *Integrity Measurement Module (IMM)*, *Memory Protection Module (MMP)* and *Secure Vault Module (SVM)*. At the development time, developers generate integrity manifests for all protected programs. Integrity manifest contains program measurements along with a unique program identity. These manifests need to be shipped with the hypervisor.

  At runtime, after OS initialized the protected program, it registers itself with IMM. IMM verifies the integrity of the loaded program (according to the program manifest) and records its memory address bounds.

  From hereon, MMP assures that program address bounds are only accessible by the program itself and no other processes. The hypervisor moves the program page tables to unaccessible shadow page tables. MMP also guarantees the protected memory pages are only accessible via valid program entry points.

  SVM provides authenticated sensitive data locking/unlocking features for the protected application. When an application requests for data sealing, its unique identity (in its manifest) is concatenated the sensitive data and encrypted using AES-GCM. The encryption key resides inside TPM. Likewise, when a protected program request for decrypting its sensitive blob, its identifier will be matched against the id in the sealed data. If it matches, then the sensitive data will be moved to the protected memory of the program.

  They have implemented their approach and reported promising performance results. Registration call (when the program is loaded) is the most substantial overhead with an average of 227 microseconds.

  Remarks: The hypervisor is vulnerable to attacks just like other programs. Another pitfall is the sealing key that resides inside the hypervisor. So, a successful attack can reveal all programs' locked data. Also, a useful program usually uses system libraries. This approach fails to provide any protection for system libraries. Vulnerabilities in the protected application, such as buffer overflows, can potentially lead to compromises.

- Timing Based Malware Detecting in Active Memory [100]
  This work proposes a software-based attestation to detect malware in the active memory of mobile devices using the memory-printing technique. The attestation relies on the fact that a clean (malware-free) an attestee can compute the challenges in a time-bound that is known to the verifier. In this approach, once the attestation is launched, all active memory contents are copied to the flash memory. Then, the verifier sends a random seed that the attestation client uses for proof computation. At different checkpoints, the verifier sends further seeds and defines the memory addresses that need to be incorporated in the proof computation. The authors claim since the hardware configuration is known to the verifier, it can estimate a reliable computation timing. Any attempt by the malware to prepare the results by interfering with the attestation or by outsourcing the computation will cause additional latency, and thus it will be detected by the verifier.

  The authors reported that the entire attestation process takes less than a minute. However, they, in their follow-up work [101], have improved the efficiency of their algorithm so it can be used in devices with large memory space.

  Remarks: The approach causes substantial downtime in service delivery.

- Forensic Analysis for Tamper Resistant Software [102]
  This paper proposes a software integrity protection based on event logs. The authors indicate that reverse engineering attacks are inevitable. However, they argue such attacks can be pro-actively detected, given that there exists a regular connection between clients and servers. The idea is to maintain a secure audit trail of the integrity checks in the application that cannot be forged by the attacker. Here the main desired security property is forward security. It indicates that once the attacker compromised the system, he cannot forge the recorded event logs due to key evolving cryptography that was used in the records. In this work three different secure audit trait techniques are proposed: **i)** key progression independent of log values, **ii)** key progression dependent of log values and **iii)** key progression dependent on the number of evolutions and independent of log values. They have implemented their approach and reported acceptable overhead in both logging and verification processes.

  Remarks: The logging mechanism itself can be attacked by the hacker. The big security assumption is that a hacker starts the process of reverse engineering without the knowledge of a logging mechanism in the system. While he can simply learn that logging is the cornerstone of the integrity checking and forge good log states regardless of the current state of the program. Furthermore, the key evolution independent of log values may enable an attacker to forge a known good key value along with the corresponding number of evolutions to trick the integrity verification process.

- Emulation based Software Protection [115]
  This technical report proposed an emulator-based software protection technique. In this method, the attacker is assumed to have access to the guest only. That means the emulator itself is somehow protected. The basic idea is to encrypt the program code such that it can only be decrypted and later on executed at the emulator level (which is assumed to be out of the attacker access). They also proposed a signature-based approach that applies HMAC verification before attempting to decrypt the code blocks.

  Idea: If we isolate the emulator from the attacker and only allow him to access the emulated application only, then it is intuitively harder for the attacker to compromise the system. Because **i)** the internal behavior of the emulator is unknown and **ii)** the instructions of the emulated application are unreadable (due to the encryption or transformations). To do so, one can build a hypervisor that acts as an emulator of its guest OSes.

  Remarks: if the attacker gets hands on the emulator itself, the system security will be at stake.

- Control-flow integrity [2]
  Subverting a program's execution follow may end in severe exploits in the system. This work presented a robust but straightforward technique to defeat *Control Flow Graph (CFG)* tampering attacks. Such attacks stem from the shared channel between data and control in systems. An attacker may manage to alter a CFG maliciously and, for instance, to jump in the middle of a function to form an exploit. The proposed approach protects the integrity of control flow by enforcing block authentication at the beginning of blocks. They assign unique ids to $caller$ and $callee$ basic blocks so that every $call$ is augmented by a $callee_{id}$. Before executing a callee, its id is verified according to the requested id (similar to a token). Consequently, all destinations are authenticated with their unique block id. All required modifications, adding unique ids and authentication enforcement routines, are done at a binary instrumentation and machine code rewriting process. For this purpose, the Vulcan tool [166] is used.

  They have implemented their proposal and reported an overhead of 16%.

  Remarks: This approach is only effective when the attacker resides outside the system. Against MATE attacks, this approach provides no guarantees. In addition to that, not all low-level attacks require tampering with control flows, for instance, this method cannot resist string formatting attacks. Another limitation is that the application, including all its code blocks, must be instrumented to assign block ids and to prepend authentication routines. Some programs may leverage dynamic code generation or modification (encryption). This approach fails to support such settings.

  Idea: The same concept can be applied to provide distributed control-flow integrity. The other work [171] has proposed a method for distributed control-flow integrity, so the only missing part is to assign unique ids and to implement enforcement routines.

However, it is not clear what the attacks that this approach is aiming to prevent them are?

- Protection Against Changeware [19]
  This work touches upon an interesting and reoccurring problem that goes hand in hand with program non-repudiation. The context problem points to the lack of process-level authentication upon any asset access/modification. This arises from the permission model in operating systems, i.e., user-centric rather than process-centric. This enabled specifically designed malware to surreptitiously change programs' assets, e.g., its data or configurations, in a way that this modification benefits the attackers. Browser hijackers are a perfect example of such modifications in which, for instance, the default search engine is replaced with a malicious one.

  The primary assumption is that OS is trustworthy and remains genuine throughout the entire process. Another assumption is that an attacker (in this case a malware) under no circumstances obtains root privilege.

  The idea in this work is to prevent illegitimate changes to program assets. For this purpose, one has to assure the followings: **i)** Assets are protected from external modifications. This is ensured using Whitebox cryptography [53] so the key is concealed within the application. Whitebox crypto guarantees the difficulty of key retrieval from the binary. Additionally, software diversification techniques are utilized to harden key retrieval; and **ii)** only genuine internal calls can modify assets. This is done via a runtime integrity monitoring that traces stack's return addresses and compares them to known good call traces. This works well for synchronous calls; however, trigger and forget calls remain out of the monitoring sight. To cope with asynchronous calls, which lead to incomplete trace information, caller threads are verified using a Message Authentication Code (MAC).

  The authors designed their protection method as a measure for Google Chromium. Driven from the use case, they reasoned that diversifying the entire browser code raises issues in software updates and bug fixings. Therefore, they developed the protection as a standalone application so-called *WBCrypto* equipped with diversification and white-box AES implementation. At runtime, when a call is made to access assets, WBCrypto acts as a proxy. The integrity of the caller at WBCrypto is evaluated by injecting required instructions into the caller.

  The authors implemented this work and reported 130x, 120x and 137x slowdown for AES-128, AES-192, and AES-256, respectively, Since the calls to configuration changes is rarely taking place, such an overhead is acceptable for this specific use case.

  Remarks: This approach only works for operating systems that allow injection from a process to another one. Linux operating system does not permit such injections. Also, the reported overhead is quite high, which is mainly due to the white-box cryptography.

- A graph game model for software tamper protection [72]
  This paper, while acknowledging there is no ultimate tamper resistance method that resists against polynomial-time adversaries, proposes a set of design ideas that can significantly harden the identification and disabling of a tamper protection technique: *distributed checks* and *threshold detection scheme*. Distributed checks refer to a network of checkers that attackers need to disable to tamper with an application successfully. These checks must blend well in the program logic. The threshold detection scheme introduces straightforward protection against step by step disable of checks. In this response model, the tamper detection will only call the respond function if after k number of checks fail. Their proposed protection scheme comprises four steps: **i)** clone sensitive codes so to avoid single point of failure; **ii)** transform program's flow graph so that it's execution resembles a random traverse; **iii)** blend checkers into the application; and **iv)** transform flow graph such that k tampering check failures triggers the response function.

  Remarks: The presented work comes with no implementation or evaluations.

- Software Protection for Dynamically-Generated Code [88]
  Virtualization is a technique in which a specific instruction set is mapped on a set of random instructions. In software protection, this mapping can be used to complicate software analysis on the attacker's end. There are two type of virtualization: *system-level* and *process-level*. The former provides an entire system stack from hardware all the way to the OS and applications. The latter, however, enables the very same thing at a process within a running system.

  Process-level virtualization, aka dynamic binary translation (DBT), is widely used by the software protection community as protection primitive. In this approach, first, a set of new instructions (typically random) is generated. Then, a binary is translated into the new instruction set. To be able to run the application, it has to be translated to the host instruction set at runtime. This is the responsibility of the emulator, aka PVM, that is shipped with the application.

  For better performance, DBT techniques commonly utilize caching, for instance, to avoid translating hot regions of the application more than once. However, this can potentially enable attackers to tamper with the cached code and subvert the execution at runtime. This paper proposed a tamper-proofing technique for the translated cached program blocks. The idea is to add a network of self-checksumming guards similar to [47] (referred to as knots by the authors) at the translation phase. The protection utilizes some randomness in the generation of networks as well as knots' instructions to make the analysis more difficult. For instruction obfuscation, they used a prepared database that is loaded with a bunch of syntactically different but semantically equivalent protection instructions. This will raise the bar against dynamic tampering attacks. Their implementation introduced, on average, 10% overhead on the SPEC2000 benchmark. This does not reflect the overhead that is introduced by process-level virtualization itself. The virtualization tool that was used by them, Strata, reported 30% slowdowns. So, all in all, 40% overhead is anticipated.

- Bona Fide [140]
  From the end-user perspective, it might be sufficient to be aware of suspicious configuration modification by the cloud providers rather than strongly preserving the confidentiality and integrity of the data. To this end, this paper proposed a scheme for change detection and continuous remote attestation with the help of a third party. The system informs users as soon as a potential malicious behavior from the cloud provider is noticed. BonaFides system is capable of monitoring undesired commands, binary execution, and hardware configuration changes on the system.

  In this approach, *Cloud Certifier*, exterior to the cloud provider, along with the *Infrastructure Provider* equipped with a *TPM*, sets up a configuration enforcement and configuration change monitor. On the cloud certifier, *Storage Service* keeps track, logs all activities, of the service provider configuration changes, and upon change detection, decides whether the incident is legitimate or not. On the infrastructure provider, mainly three components are collaborating: *Attestation Service (AS)*, *Integrity Measurement Engine (IME)* and TPM. AS acts as an intermediate between the certifier and provider. It inquiries the demanded configuration from the certifier then redirects them to IME. The latter keeps observing redirected configuration list, upon change detection informs AS.

  The integrity of AS itself is guaranteed using TPM, which is signed by a trusted certificate authority. Infrastructure provider has to securely boot the host so that TPM captures platform configuration hashes including attestation modules to its platform configuration registers.

  Remarks: This technique relies on the attestation and thus is known to be problematic [94]. Although attestation informs the verifier about the system's configuration, it is challenging for the verifier to accept the configuration. Because acceptable configurations have to be updated with every software/hardware change in the system.

  Revealing the user's machine configuration is violating the privacy as the certifier learns everything about the host literally. For this problem, the Trusted Computing Group has developed a new paradigm to remove the third-party certifier, which results in the *direct anonymous attestation* [36].

  Furthermore, in software configurations case, attestation only considers the binary hash, while in practice, some problems may appear during execution, such as buffer overflows.

- Repackage-proofing Android Apps [128]
  In this paper, integrity protection is utilized to thwart repackaging attacks in Android applications.

Repackaging refers to an attack in which attackers malevolently modify (for instance, to show more ads or to profile users) a well-known App and redistribute it on App Stores. This has two downsides. Firstly, it harms the App developer revenue as well as reputation. Secondly, it violates end-users privacy and security.

Each Android has to be signed by the private key of the developer (who owns the App). Since attackers won't have access to this private key, they have to resign Apps after repackaging them using different key-pairs than the original ones. This paper has built upon this limitation on the attacker side.

Contributions are Stochastic Stealthy Network of checkers, secure communication channel using R object, and stealthy (temporally disjoint) response model. A checker is split into smaller chunks that are blended into a program function while preserving the order of execution. To preserve the order, the basic dominator blocks of a function are identified [122]. Subsequently, the checker chunks are injected into the basic blocks in the order of dominators.

The idea is to inject a network of checkers (referred to as Stochastic Stealthy Network) at development time into Apps, These checkers (which are diversified using various obfuscation techniques) verify the public key of the signing authority (which is retrievable via Android runtime) to ensure it matches the original developer key. To add resilience against pattern matching, the public key is not verified as a whole, but each checker verifies a smaller subset of the key. The authors have not specified how large these intervals are.

Upon mismatch detection, a signal is published to the response mechanism. The communication medium is via the Android.R, which is responsible for the id of UI elements. Simply put, checkers modify the id of an R field (using reflection), which is actively read by responders. The authors argue this medium has high resilience.

They designed a custom response mechanism that messes up the application UI. Suggestions are to add random text to labels, modify integer values, and object attributes.

To reduce the overhead, they used fuzzers (using Monkey and Traceview tools) to identify hot code regions in a given program to protect. These regions are marked not to be used for injecting checkers.

Four apps are used to carry out performance evaluation. First, 1000 random user events are generated (using monkey). And then, these events are fed to both protected and unprotected apps. They reported an overhead of 8-12%

For the security evaluation, they have used human attackers to measure their success rate as well as carried out a thorough security evaluation. Their scheme is vulnerable to *hijacking vtable attacks*, while they state other attacks, including taint analysis, were not successful.

Remarks: Public key is a high entropy data, so it should be easy to detect them in the code. One can simply monitor modifications on R element values at runtime to detect the checkers. Furthermore, taint analysis based attacks similar to [150] could lead the attacker to checkers and responses.

- Transparent ROP Exploit Mitigation Using Indirect Branch Tracing [145]
  In this paper, a protection mechanism is proposed to mitigate ROP-based attacks at the kernel level. That is, programs need not be instrumented and protected, but rather the kernel is responsible for mitigating such attacks. The protection relies on two measures. For direct branches, the ret instruction must not violate the call site, i.e., the instruction before each ret address must conform to the executed call. In indirect branches, a chain of shortcode fragments (potentially gadgets) within a threshold of 10 indicates an ROP attack. While i is simple to implement, i seems to be challenging. However, it turned out that CPU has an LBR register that maintains up to 16 indirect branches. They state that all ROP attacks have to eventually invoke a system call to carry out a meaningful attack. Based on this assumption, the check could take place right before the execution of system calls. Their performance evaluations indicate an overhead between 4-8%.

- Haven: SGX in Cloud [26]
  Haven is a shielding method to secure legacy systems leveraging Intel's Software Guard Extension (SGX). In this approach, they place the entire legacy binary inside the protected enclaves. To protect system calls, they have also developed a secure version of Windows 8 libraries such as threads, scheduling, virtual memory and file system. This effort is mainly to cope with the enclave's security level. They have implemented their technique and reported an overhead between 31% to 54% depending on the application.

- Harden Tamper-Proofing to Combat MATE Attack [50]
  This work sets to secure the self-checksumming (SC) guards via Intel SGX. To this end, they propose a design comprised of three compartments: non-enclave part, enclave part, and sealed addresses table. In the non-enclave part sits the original program augmented with SC calls. Such calls transfer the execution to the code residing in the enclave (containing the SC guard). Since the actual program resides outside the enclave, the SC guard must read from the non-enclave compartment. The checksum function of the guard disguises the target memory addresses by reading extra random memories. The target addresses (checksum intervals) are shipped as an SGX-sealed auxiliary file with the binary. It is worthwhile to mention that attackers cannot read the content of SGX-sealed files. The SC guard within the enclave reads the target addresses from the file upon program start and accesses them (along with other random memories) upon checks. Later the guard calculates the checksum on the target addresses and discards the random read values. The protection is implemented in C++ and is tuned to inject eight guards in each program. They conduct a performance evaluation on five benchmarks (out of 12) from the SPECint-2006 dataset. It is not

clear why seven other benchmarks were excluded. The imposed overhead appears to be constant for the number of calls and accessed memory cells.

Remarks: The assumption is that if attackers cannot guess which addresses are checked, the trick of setting patched memories to legit values before tamper detection calls will not work. There might be cases in which attackers do not necessarily need to know the checked addresses to bypass the protection. One simple solution would be to revert all patched instructions upon calls to tamper detection. It is often enough to patch a branching condition to coerce the intended malicious behavior (e.g., to bypass a license check). Given that the number of patched instructions are generally-speaking limited, the imposed overhead of reverting memory values is rather low. Therefore, it is likely that attackers bypass the protection by resorting to a well-timed patch and revert strategy.

- Semantic-integrated software watermarking with tamper-proofing [51]
  This paper proposes a control-flow-based watermarking technique that comes with built-in tamper detection. The core idea is to apply a control-flow obfuscation transformation that embeds branching conditions into an artificial neural network model. The authors argue recovering a program's control flow in their scheme is equivalent to extracting rules from artificial neural networks, which is assumed to be NP-hard. The model receives the branch variables and information on the previously executed program blocks (trace) as inputs. The model effectively evaluates the branching conditions by spitting out each branch's destination addresses based on the inputs. Since the models are trained with both variables and traces, any modifications leading to changes in the traces cause miscalculation of the addresses (yielding potential program terminations). That is, the scheme realizes an oblivious hashing alike trace protection. In the protected program, all branching conditions are replaced with calls to a function consulting the model to compute destination blocks' address, followed with an unconditional jump to the addresses. The authors implemented their scheme in C++ and evaluated it using five programs from the SPECint-2006 benchmark. Their performance evaluations indicate a low overhead. However, the authors strictly protected 16 branches in each application.

  Remarks: It is unclear why the authors apply the technique to 16 branches and not all the branches in each program. The overhead can significantly differ if they protect all program branches. The rule extraction from models seems to be a problem in the context of a particular class of models, such as Multilayer Perceptrons (MLPs) [32]. Dynamic analysis can still be problematic as calculated addresses can be recorded and subsequently used to replace the calls to the model.

- Careful-Packing: A Practical and Scalable Anti-Tampering Software Protection enforced by Trusted Computing [174]
  This paper proposes a tamper detection technique aided by Intel SGX. The goal is to extend the security guarantees of enclave-protected code regions to the codes

residing outside enclaves. The proposed technique combines software packing and self-checksumming to achieve their goal. In their work, sensitive code regions are shipped encrypted (AES-GCM) to the end-users. A backend server, upon installation, provisions the corresponding cryptographic key. Such sensitive regions are enclosed with *decrypt* (prologue) and *encrypt* (epilogue) calls to the enclave in the protected application. The decrypt function checks the integrity and, if valid, subsequently decrypts the section right before executing the code. On the other hand, the encrypt function re-encrypt the section to raise the bar against dynamic analysis. Upon calls to the trusted enclave, a self-checking mechanism randomly verifies previously declassified sensitive sections to detect post decryption tampering attacks. The packing has the benefit of enforcing calls to the trusted enclave, mitigating disengaging the trusted component. The very same construct is leveraged to broadcast heartbeats containing tampering reports to a trusted third-party server. The authors demonstrated the resilience of the technique to a just-in-time patch and repair attack. They evaluated their approach using a self-made activity monitor tool (which is meant to detect malicious insiders). The induced overhead is estimated to be 5.7% on average.

Remarks: The utilization of packing hardens dynamic analysis, but cannot counter well-timed patch and reverts. The just-in-time patch and repair attack experiment seems to randomly attempt to patch and revert bytes, which is, according to the authors, always detected by the guards. It would be interesting to simulate a real-world attack by targeting a set of sensitive instructions (e.g., the patch and revert of a critical branching condition) in the experiment.

- Software Protection Using Dynamic PUFs [187]
This work combines self-checksumming with Physically Unclonable Functions (PUF) to constitute a tamper protection technique that binds applications to specific hardware. Through a trusted software interface, the protected program queries PUFs. Since the PUF queries are time-bound, modifications in the protected application that yields timing differences (larger than a certain threshold) are detected. A set of cross-checking self-checksumming guards protect the sensitive code as well as the PUF queries. The PUF queries' results are used to determine the next function's address in the control flow (referred to as call graph scrambling). Similarly, PUF values can be used to manipulate program registers such as stack pointer to introduce a fault in the execution upon tamper detections (referred to as register value scrambling). The protection is implemented as an instrumentation tool written in Python for the LLVM bitcode format (IR). The authors used a set of self-implemented sample programs (mainly utilizing AES the algorithm) to evaluate their technique's runtime overhead. The experiments indicate an average overhead of 48%.

  Remarks: The authors presume that the Dynamic PUF interface is trusted.

- Protection against reverse engineering in ARM [190]
This work proposes a software packing technique for the ARM architecture to conceal

the code by resorting to encryption. The authors seek a solution that does not rely on the ARM TrustZone due to what they refer to as vendor lock-in problems. The answer comprises code encryption and a secure hypervisor that acts as an emulator for the encrypted code. The paper details ARM specific considerations regarding the CPU privilege model and the memory management security considerations. The authors used an FFT program to conduct their overhead experiments indicating high overheads on first runs (before caching the decrypted code) followed with insignificant overheads.

A.3. Summary

In this chapter, we summarized 51 papers in the field of integrity protection. We discussed potential improvements and shortcomings of the reviewed articles.

B. Reproducibility Handbook

We made the entire protection toolchain, datasets, and scripts open source. This chapter capture the links, scripts, and manuals on how to run the experiments that were conducted throughout the thesis.

We developed our toolchain in the LLVM ecosystem. That is, we have implemented our protections and the composition framework as LLVM transformation passes. All the passes are written in LLVM 6.0. The simplest way to reproduce our results is to use the docker file provided for each part of the thesis (see Appendix B.3). It is worthwhile to note that the publications corresponding to Chapters 3 and 5 ([4, 8]) have undergone artifact evaluations [1] and subsequently been awarded with ACM Reusable and ACM Functional badges.

B.1. Data sets

We mainly used the MiBench data set [93] to evaluate the runtime overhead of our protection passes. In our machine learning experiments, we used the simple data set [154] in addition to the MiBench data set to conduct our localization attacks. We compiled all these programs to LLVM bitcode format (BC) and used them in our experiments. The bitcode files of the simple and MiBench data sets can be found at `https://github.com/mr-ma/composition-sip-eval/tree/smwyg/simple-dataset`, and `https://github.com/mr-ma/composition-sip-eval/tree/smwyg/mibench-dataset`, respectively.

B.2. Source Code

The LLVM parts are written in C++ version 17. We do use Python 2.7 and Python 3.3 in our experiment scripts. We have tested our implementations on Ubuntu 18.04 and Ubuntu 18.10.

[1]`https://www.acm.org/publications/policies/artifact-review-badging`

B.2.1. C++ Dependencies (Libraries to Link)

DG

We used the `dg` library (`https://github.com/mchalupa/dg`) to compute pointer dependencies in the alias analysis of programs. We forked the dg library to accommodate minor fixes for the programs of our data sets. Our fork is reachable at `https://github.com/tum-i22/dg`.

Lemon Graph

We made use of the lemon library (`http://lemon.cs.elte.hu/pub/sources/lemon-1.3.1.tar.gz`) in our implementation of the defense graph in the composition framework.

GLPK

For our ILP optimization, we used the GNU Linear Programming Kit (GLPK). For larger programs, we used the Gurobi solver (`https://www.gurobi.com/`) under a free academic license.

SVF

Static Value-Flow Analysis Framework for Source Code (`https://github.com/SVF-tools/SVF`) is used to capture a feature vector of the data and control flow dependencies of the program basic blocks. We adopted the build script in our forked repository available at `https://github.com/anahitH/SVF`.

B.2.2. Python Dependencies (Libraries to be installed via pip)

We need the following pip libraries in our toolchain:

- r2pipe (Radare2 must be installed as well)
- numpy
- scipy
- argparse
- pandas
- benchexec==1.16
- pypandoc
- gensim==3.8.1

- sklearn
- tabulate
- tensorflow==2.1.0
- stellargraph==0.10.0
- keras==2.3.1
- tensorflow-cpu==2.1.0
- tables

B.2.3. Utility Passes

The utility passes are needed by the protection passes. These passes are implemented in C++/LLVM.

Function Filter

We use the function filter pass to apply partial protections using function names on sample programs. The pass is available at:
`https://github.com/mr-ma/composition-function-filter.git`

Inter-Procedural Program Input Dependency Analyzer

The input dependency pass segregates program instructions into data-independent, input-dependent, and control-flow-dependent instructions. The source code of the input dependency analyzer is available at:
`https://github.com/mr-ma/composition-input-dependency-analyzer.git`

Program Dependence Graph

This pass generates precise data flow and control flow dependencies of program instructions. We used this pass to capture a feature vector of program dependencies for our machine-learning-based attacks. The program dependence graph source code is reachable at `https://github.com/mr-ma/program-dependence-graph.git`.

B.2.4. Protection Passes

Throughout this thesis we prototyped five integrity protection techniques, namely Self-Checksumming (SC), Oblivious Hashing (OH), Short Range Oblivious Hashing (SROH), Call Stack Integrity Verification (CSIV), and Virtualized Self-Checksumming (VirtSC). The link for each of the protection passes are given below.

1. SC
   `https://github.com/mr-ma/composition-self-checksumming.git`
2. (SR) OH
   `https://github.com/mr-ma/composition-sip-oblivious-hashing.git`
3. CSIV
   `https://github.com/mr-ma/composition-sip-control-flow-integrity.git`
4. VirtSC
   `https://gitlab.lrz.de/ga93jun/sc-virt` (only accessible to TUM students/staff with invitations)

B.2.5. ILP-Based Composition Framework

The framework is written as a set of LLVM passes. The composition framework is available at:
`https://github.com/mr-ma/composition-framework.git`

B.2.6. Machine-Learning-Based Localization

Our machine-learning-based attack is written in python and is publicly available at:
`https://github.com/mr-ma/sip-ml`

B.3. Docker Files

We recommend using the provided docker files to install and configure the required dependencies properly. The link to the docker file for each chapter with reproducible results is given below.

1. Chapter 3
   `https://github.com/tum-i22/sip-oblivious-hashing/tree/acsac/docker/Dockerfile`
2. Chapter 5
   `https://github.com/mr-ma/sip-shaker-artifact/blob/master/Dockerfile`
3. Chapter 6
   `https://github.com/mr-ma/smwyg-artifact/blob/master/Dockerfile`

Please note that we do not provide a dockerized approach for the experiments of Chapter 4. Due to IP problems we are unable to disclose the code publicly.

B.4. Execution Manual

After building the docker images based on the provided docker files, run a container and re-run the desired experiments according to the manual provided for each chapter.

1. Chapter 3
   `https://github.com/tum-i22/sip-oblivious-hashing/blob/acsac/README.md`
2. Chapter 5
   `https://github.com/mr-ma/sip-shaker-artifact/blob/master/README.md`
3. Chapter 6
   `https://github.com/mr-ma/smwyg-artifact/blob/master/README.md`

List of Figures

List of Listings

List of Tables