TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

Information Exposure Checker

Paul Muntean und Alexander Malkis

TUM-I1646

SIBASE
Technische Universität München
Chair for IT Security
Workgroup TP 5.1

SIBASE Report

TP 5.1 – AP 5.1.4: Information Exposure Checker

Paul Muntean (TUM) and
Alexander Malkis (TUM)

SIBASE
Technische Universität München
Chair for IT Security
Workgroup TP 5.1

SIBASE Report

TP 5.1 – AP 5.1.4: Information Exposure Checker

Editor: Paul Muntean
and Alexander Malkis
(paul@sec.in.tum.de and
malkis@sec.in.tum.de)

Authors:	Paul Muntean, TUM
	Alexander Malkis, TUM
TP Responsible:	Alexander Malkis
Version:	30. Juni 2016
Submission date:	30. Juni 2016

Funded by BMBF under grant number 01IS13020

Abstract

This work package presents an information exposure checker which is designed to detect information exposures in C/C++ code.
Information flow vulnerabilities in C code are detrimental as they can cause data leakages or unexpected program behavior. Detecting such vulnerabilities with static code analysis techniques is challenging because of complex control and data flow. Static analysis tools used for detecting information exposure bugs can help software engineers detecting bugs without introducing run-time overhead. Such tools can make the detection of information-flow bugs faster and cheaper without having to provide user input in order to trigger the bug detection. We present a bug-detection tool for detecting information exposure bugs in C/C++ programs. Our tool is context-sensitive and uses static code analysis for bug detection, which was developed in the SIBASE working package 5.2.1. We developed our bug finding tool as an Eclipse plugin in order to easily integrate it in software development work flows. Textual annotations introduce information flow constraints into code as described in the SIBASE working package 5.1.2. The constraints are checked later by our tool. The bug reports provide user friendly visualizations that can be easily traced back to the location where the bug was detected. We discuss one static analysis approach for detecting information exposure bugs and relate briefly the usability of our bug testing tool to empirical research. We conducted an empirical evaluation based on 90 test programs which were selected from the National Institute of Standards and Technology (NIST) Juliet test suite for C/C++ code. We reached a true-positive coverage of 94.6% in $\approx$121 seconds for the test programs . Our results show that our approach is effective and can be further applied to detection of different types of vulnerabilities. This report is based on publications listed in Chapter 6.

Version	Date	Author	Comment
0.1	21.01.2016	P. Muntean	Initial draft
1.0	30.06.2016	A. Malkis	Intermediate version
1.1	18.07.2016	A. Malkis & P. Muntean	Final version.

Thanks to Hermann Drexler for reviewing version 1.0.

Contents

List of Figures

List of Tables

List of Acronyms

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
AST	Abstract Syntax Tree
ECORE	EMF File Extension
EMF	Eclipse Modeling Framework
ESC	Extended Static Checking
IDE	Integrated Development Environment
IE	Information Exposure
IF	Information Flow
IFC	Information Flow Control
IFW	Information Flow Weakness
LOC	Lines of Code
OCL	Object Constraint Language
OS	Operating System
SAE	Static Analysis Engine
SW	Software
TUM	Technische Universität München
UML	Unified Modeling Language
xTend	A programming language on top of Java Virtual Machine, see [64]
xText	A framework for developing programming languages and domain-specific languages, see [65]

Chapter 1: Introduction

Information exposure weaknesses are a type of Information Flow (IF) weaknesses. IF weaknesses represent one type of software weakness, which can exist in the software without directly breaking the code but rather offering useful information to an attacker who could exploit IF leakages [66]. These types of software bugs can lie dormant in an application for a long time period without being detected and can cause huge harm [66]. According to Common Weakness Enumeration (CWE) CWE-200 (the parent weakness class of the test programs used in this paper) [37] Information Exposure (IE) is the "intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information". IE vulnerabilities are a subtype of IF vulnerabilities. As of 2007 IE leakages rank 6th in the OWASP top ten list [50] and as of 2010 rank 7th according to VERACODE mobile app top ten list [62]. In the past 12 months, 7.76% of all the vulnerabilies and exposures registered in the US National Vulnerability Database (NVD) [49] are information leaks/disclosures caused by inappropriate handling of information flow in software applications. We argue that software should be thoroughly tested before it is released in order to detect potential exploitable IF vulnerabilities.
The process of software testing accounts for more than 50% of the whole effort during software engineering projects according to [34] and [35]. Detecting software bugs, which cause information exposure vulnerabilities is crucial because potential exploitation possibilities should be removed from source code before release. Software weaknesses are hard to detect and can cause information leaks which attackers can exploit. By building Control Flow Graphs (CFG) which describe possible execution paths and tracking taint data as it "moves" along the path nodes guarantees high path coverage.
Many static analysis approaches are very promising but still have to be applied to security scenarios. At the same time a relative high number of tool vendors (e.g., Microsoft, IBM, Coverity, klocWork, Infosys, Cognizant, Hexaware) start to address the need for static analysis into mainstream tools. Some example tools are ESP [19], which is a large scale property verification approach, model checkers as SLAM [5] and BLAST [29], which use predicate abstraction to examine program safety properties. FindBugs [52] is a lightweight byte code checker based on predefined bug patterns. Triggering IF bugs is not a trivial job and can be addressed using dynamic analysis, static analysis or hybrid approaches. Dynamic analysis introduces computing overheads and it cannot guarantee that all possible execution paths are exercised. Where as static execution provides all potentially execution paths but needs some heuristic for selecting only the relevant paths. Also it is relevant to select only reachable paths, which can be determined with the help of an SMT solver. The mathematical expressions provided to the SMT solver often are blown up in size and can get very complex [10].
IF vulnerabilities can be addressed by dynamic analysis, as for example by Fenton et al. [23] or by Sabelfeld et al. [55], static analysis techniques [2, 26, 47, 57, 60, 63], and hybrid approaches

which combine static and dynamic techniques‘[41]. Tracking taint variables through the program execution is key to detect IE weaknesses, which are a type of IF vulnerabilities. IF controls focus on preventing leaks from confidential (or high) to output (or low) data. The desired baseline policy is noninterference [18] that demands that there is no dependence of public outputs on confidential inputs. There are two types of IF variants, which can be taken into account when dealing with variable interference. Information is passed from right-hand side to left-hand in an assignment through an explicit flow. Assume variables *confidential* and *output* have high and low security levels, respectively. For example, the assignment

$$\mathit{output} := \mathit{confidential}$$

exhibits an explicit flow from confidential to output. Information is passed via control-flow structure in an implicit flow. For example, the conditional statement

$$\textbf{if } \mathit{confidential} \textbf{ then } \mathit{output} := \mathit{true} \textbf{ else } \mathit{output} := \mathit{false} \textbf{ fi}$$

has an implicit flow. The value of the output variable depends on the *confidential* variable. We will call a conditional or a loop high if its guard involves a high variable. Information-flow control is concerned with preventing explicit and implicit flows in order to guarantee non-interference. One possibility to prevent explicit and implicit flows is by using purely static Denning-style enforcement [22]. Each assignment is checked if it fulfills the following conditions: the level of the assigned variable must be high when there is a variable on the right-hand side of the assignment (tracking explicit flows) or in case the assignment appears inside of a high conditional or loop (tracking implicit flows). This mechanism guarantees that no low computation occurs in the branches of high conditionals and loops as used by Russo et al. [54].
Another possibility is through dynamic enforcement, which is based on dynamic security checks similar to the ones done by static analysis as presented by Russo et al. [54]. Whenever there is a high variable on the right hand side in an assignment (tracking explicit flows) or the assignment appears inside a high conditional or while loop (tracking implicit flows) then the assignment is only allowed when the assigned variable is high. This mechanism dynamically keeps a simple invariant of no assignment to low variables in high context
We have chosen the C programming language and Eclipse IDE [61] as stated in the proposal. By designing an Information Exposure Checker (IEC), which can run in different running modes, we think that we can increase the productivity of the code debugging process. The IEC can detect bugs during run-time of the Eclipse IDE and it offers two main advantages. First, it offers the possibility of detecting IE bugs during development. Second, we get a high level of integration between the IDE and the bug detection mechanisms.
The goal of our research is to develop a tool for detecting IF exposure bugs using static analysis. The tool should use context-sensitive analysis and should rely on a Satisfiable Modulo Theories (SMT) [1, 28] solver. In summary we make the following contributions:

- We developed an context-sensitive IE detection tool capable to detect information exposure bugs fully automated using SMT-LIB 2.0 [6].
- We devised a novel light-weight security annotation language used to define information flow constraints in source code.
- We propose a new method to define sinks, sources and taint confidential symbolic variables by first, function models, and, second, automatically loading previously added code annotations.

- We defined an easy-to-use method for adding new checkers into the Static Analysis Engine (SAE) [32] by adding the required function models for the sinks, sources and tainting confidential variables.
- The SAE statement processor was extended to support explicit information flow propagation of symbolic variables.
- We designed our checker as an eclipse plug-in which can be run in different modes: as presented in Fig. 1.1.

IF checker can be run:

- as the user types in,
- on incremental build,
- on full build, or
- on demand.

Figure 1.1: IF checker running modes

The remainder of the paper is organized as follows: Chapter presents related work, Chapter 3 gives a brief overview about the architecture and the implementation, Chapter 4 presents experimental results, and Chapter 5 contains the conclusion and future steps.

Chapter 2: Related Work

This Chapter presents related work to ours: first, static analysis techniques and tools are presented 2.1, second, dynamic analysis techniques are presented 2.2, and finally, hybrid analysis techniques are presented 2.3.
According to Ceara et al. [11] every kind of static taint analysis is based on a type of formalization. For example program dependence graphs as used by Hammer et al. [27, 59], program slicing techniques as used by Pistoia et al. [51], or type systems used by Foster et al. [25] in their tool cqual [24] tool.
For scalability reasons, internal representations as Static Single Assignment (SSA), Gated Single Assignment (GSA) or Augmented Single Static Assignment (aSSA) are used [56]. Similarly, we employ symbolic execution in which per each path we use a separate set of symbolic variables. We check for potential IE bugs only for reachable paths by querying the path validation. The path validation decides, based on queries submitted to the Z3 SMT solver [20], whether the current path is satisfiable or not.
Deciding during static execution whether a path is reachable reduces computational overhead and the total number of potential paths on which IE bugs could be located. Tainting confidential variables is done statically in the function models which are used to model each trust-boundary. Taint variable propagation is based on explicit IF.
Thus, a large amount of research work has been already published concerning symbolic variables tainting and propagating their values using static, dynamic, or hybrid taint-analysis.
We briefly review in this section the most commonly used approaches focusing more on the works that are close to the one we proposed in this paper.

2.1 Static Taint-Analysis

One of the approaches to compute variable taintness is to use Static Taint Analysis (STA) techniques, allowing taking into account all possible execution paths. STA does not provide runtime information, and environment interaction has to be simulated. Thus, the environment model introduces imperfections because it can not capture each real-world interaction.
The majority of static taint analysis tools are based on user input dependencies [12]. Our tool handles each potential input source independently by modeling it with function models that simulate their execution during static execution. We are capable to model inputs from users, files, sockets and input streams in this way. Our tool is similar to the compile time analyzer PREfix [7] in the sense that both tools sequentially are tracing distinct execution paths and simulate the action of each operator and function call on the path.
Static taint-analysis can be used to enforce privacy control on mobile devices. Xiao et al. [63] propose a transparent privacy control approach that uses static symbolic execution [34] based

on implicit IFs. Data is tainted using scripts developed with TouchDevelop [42], which allows users to create applications using an imperative and statically typed language. Variable tainting is based on the fact that the whole TouchDevelop API on which the user scripts are based is in advance tainted with information concerning sources and sinks. This approach is different from ours because it supposes a previously known API where everything is tainted and no other untainted sources or sinks exist. We define our sinks and sources directly in the function models and simulate real execution using them. Our approach can handle the definition of sinks and sources for the same procedure. This introduces more flexibility during analysis.
Guarnieri et al. [26] taint variables using a central knowledge base. The authors propose an Eclipse-based tool capable to detect IF vulnerabilities due to missing input validation with the help of a decentralized knowledge base and on AST's generated by the JDT compiler. The static analyzer detects security issues related to input validation problems in web applications. The IF analysis does not consider context sensitivity and it is not using a SMT solver. The framework offers the possibility to taint classes, packages or methods as trusted. The problem of IF vulnerability detection translates to identifications of errors between entry (sources) and exit points (sinks) that do not use a trusted object.
FindBugs [3, 4, 30] is based on Eclipse; it does not detect IE bugs but uses the concept of easy integration of new checkers into the static analysis. It checks for bugs in Java byte code based on currently 300 patterns of coding mistakes for Java byte code. It employs intra-procedural analysis that takes into account information from instance of tests. FindBugs has a plugin architecture in which detectors (code checkers) can be defined reporting different bug patterns. Detectors can access information about types, constant values, special flags and values stored on the stack or local variables. Some of the defined detectors perform intra-procedural summary information. FindBugs doesn't use SMT [28] or a SAT [1] solvers in order to perform the static analysis but is rather based on the previously mentioned patterns, which can be extended by the so-called detector concept. This is similar to our checker concept of easily attaching checkers to the language interpreter.

2.2 Dynamic Taint-Analysis

Another approach to computer variable taintness is based on Dynamic Taint Analysis (DTA), meaning that concrete program execution is performed. The main advantage of DTA is the possibility to use data flow information available during runtime but only from one path of execution at a time. Thus sanity checks can be handled accurately avoiding many false positives. However, since each analysis is reduced to a single (current) execution path, its coverage level may remain very weak and control dependencies cannot be fully taken into account. At the same time it cannot guarantee that all possible execution paths are exercised. Thus, it is in general geared towards explicit IF's.
The notion of taint variable was introduced with the Perl scripting language and its taint running mode where taint variables are propagated using the language interpreter across variable assignment, and security errors are raised when an insecure system call appears. There is a wide range of proposed tools until now which are based on language information-flow security: Java-based JFlow [47] with its software tool Jif [15] developed an annotation language for Java code. Data values are labeled using security policies. The attached labels restrict the movement of data values thus enforcing a policy on the data flow. The programming languages such as an

Caml-based FlowCaml [58], an Ada-based SPARK Examiner [13] and scripting languages Perl, PHP, Ruby, and Python have a taint mode similar to the taint mode available in Perl.
Dynamic taint analysis is not suitable for us because we want to have high path coverage and exercise all possible execution paths. Thus, we rely on a SMT solver, which helps to detect satisfiable paths and afterward to provide candidate paths to our IE checker.

2.3 Hybrid Taint-Analysis

Hybrid taint-analysis is a combination of the previously two mentioned approaches. Hybrid taint-analysis approaches benefit from making information from dynamic execution available to static analysis or vice-versa (to dynamic execution from static analysis). This circumvents some shortcomings of the two approaches.
The first approach explores executable paths in the same way as static symbolic execution does, interleaving concrete execution with symbolic execution. Concrete values from execution are used by these techniques when difficult constraints are reached, allowing the algorithm to proceed. Concolic testing [53] is one of the most prominent hybrid analyses. Concolic techniques can reason precisely about complex data structures and simplify constrains when they exceed the capabilities of the solver. KLEE [8], as a part of another tool S2E [14], is a concolic testing tool for C programs which extends EXE [9] and addresses path explosion by allowing interaction with the outside environment without using entirely concrete procedure call arguments.
The second approach is mostly based on IF monitors which monitor the execution of the program and use information from a static analysis to decide, for example, when it is safe to stop tracking confidential variables.
Moore et al. [41] propose a hybrid IF monitor, which combines static analysis and dynamic mechanisms in order to provide strong information security guarantees. Their implementation extends the information-flow monitor presented by Russo et al. [54]. Their approach adds runtime overhead in comparison to pure static analysis. The authors argue that their static analysis can determine when it is sound for a monitor to stop tracking the security level of certain variables. This extra information is provided through the usage of static analysis, which can reason in general more precise about certain IF's as discused by Russo and Sabelfeld [54].
The authors present conditions for incorporating memory abstractions and analysis into a hybrid information-flow monitor. The static analysis relies on a flow-sensitive security type system [31] which helps to determine when a variable cannot cause a security violation.
The environment taints each program variable with a security level and tracks the currently stored security level. Thus, allowing for a kind of automatic tainting and propagation of taint variables whereas we initially taint variables statically in our function models.
To the best of our knowledge our checker is the only IE bug checker that uses symbolic execution to find potential candidate paths on which IE bugs could reside and that propagates confidential variables based on explicit IF's.

Chapter 3: Architecture and Implementation

This Chapter presents: first, present the architecture of our tool 3.1, second, the inference rules used inside our tool 3.2, third, briefly implementation details of our tool 3.3, and finally, how our tool can be used in order to detect information flow bugs 3.4.

3.1 Architecture

Our IE checker is based on the static analysis engine [32] which is used for C/C++ source code analysis and has in the back-end the MathSat [17] SMT solver. This engine was developed within the SIBASE working package 5.2.1 [33]. Here, we explain the steps needed to extend the SAE. A new function model is created for each *sink* and *source* and added to the environment models package, which contains models for all potential *sources* and *sinks* present in the selected test cases. Function models are used to model *sinks*, *sources* and other types of trust-boundaries. At the same time function models are used to notify the interpreter when a previously tagged variable is about to pass through a trust boundary. The interpreter will be notified by sending it a potential tagged symbolic variable. Afterward all the currently attached checkers will be notified by sending the tagged symbolic variable to them. Inside the checker class it is checked if the variable is *confidential*, sensitive, etc. If the check if positive then a bug report will be issued. We expand on our checker and its subsequent improvements in more detail in [44, 46], for now we are taking a short overview.

3.1.1 Static Analysis Engine Architecture

The function models contain a tainted symbolic variable with a *confidential* label assigned to it. The symbolic variable will be propagated along a path. The interpreter will be notified when passing over a *sink*. The *sink* notifies the interpreter by sending a symbolic variable, which could be *confidential* or not. The interpreter calls each previously attached checker. The symbolic variable is checked whether or not it is *confidential* by the checker. If the variable is *confidential* then a bug report will be issued. If additional logic for checking other relevant conditions is needed then this can be added in the IE checker class. The architecture of the used static analysis engine is presented in Figure 3.1. A more-detailed explanation of the main classes contained in the SAE can be seen in the paper [32]. The reused Codan API interfaces and classes are presented in the Codan API package shown in Figure 3.1. The interface IChecker adds to the implementing class the possibility to work with project resources such as: projects, files, etc. The class `AbstractCheckerWithProblemPreferences` is an extension of the class `AbstractChecker`. It contains methods for defining the run-time settings for the checker class. Checkers can generate several types of outputs. Each checker preference settings can be

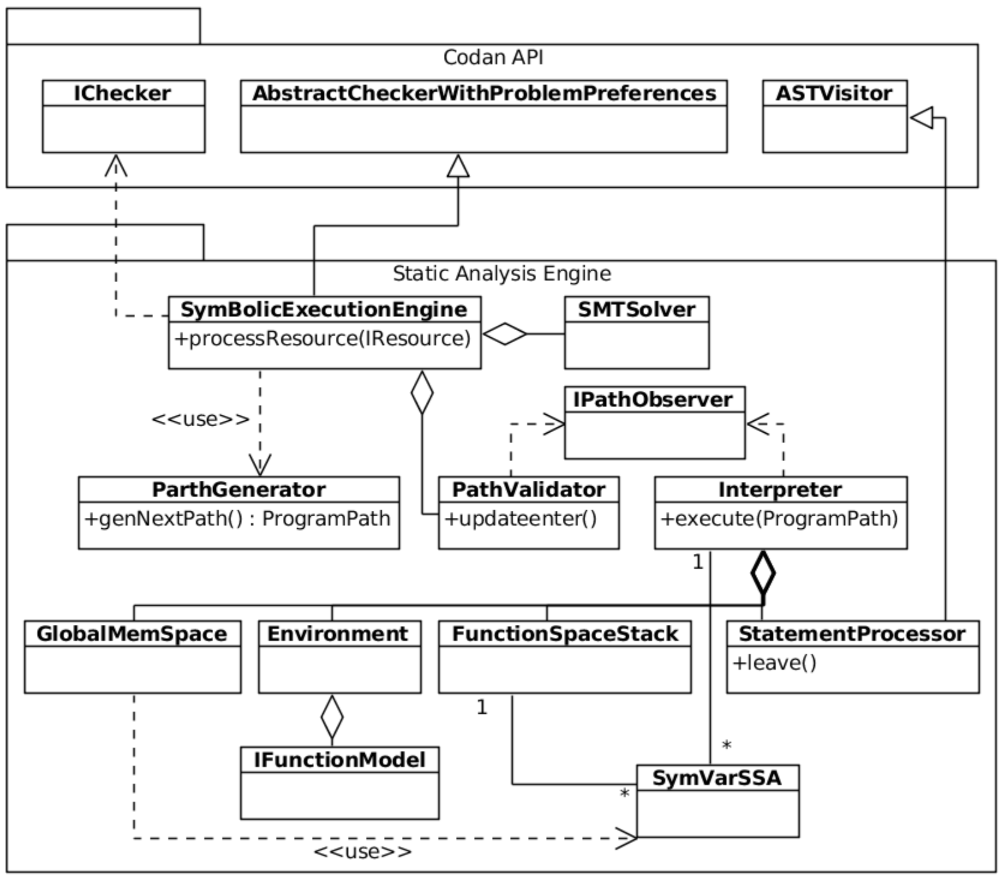


Figure 3.1: SAE architecture

defined individually. The abstract class `ASTVisitor` is a Codan base class, which is extended by all visitor classes that need to traverse the nodes of an Abstract Syntax Tree (AST). The class `ASTVisitor` implements the visitor design pattern. The `visit()` methods implement a top-down traversal and the `leave()` methods implement a bottom-up traversal of a C statement represented as an AST. Details on the engine, including the main classes contained in the SAE, are given in [32, 44].

3.1.2 Information Exposure Checker Architecture

The blue lines in Figure 3.2 indicate all the dependencies between the SAE presented in Figure 3.1 and our IE checker. Implementation details for the `InformationExposureChecker` class (IECC) will be presented since it contains the bug triggering logic. In the class `SymVarSSA` we declared a symbolic boolean variable *confidential* and defined its getter and setter. We used it to set the return value of the function call `getenv("PATH")` to *confidential*. Thus, it is possible to specify other types of variables (*sensitive*, etc.) and expressions, which could be tainted. The class `ModelGetEnvironment` (MGE) contained in Figure 3.2 implements the `IFunctionModel` interface. The MGE *sink* model of the `getenv("PATH")` contains the implementations of the `exec()` and `getSignature()` methods. The `exec()` method will

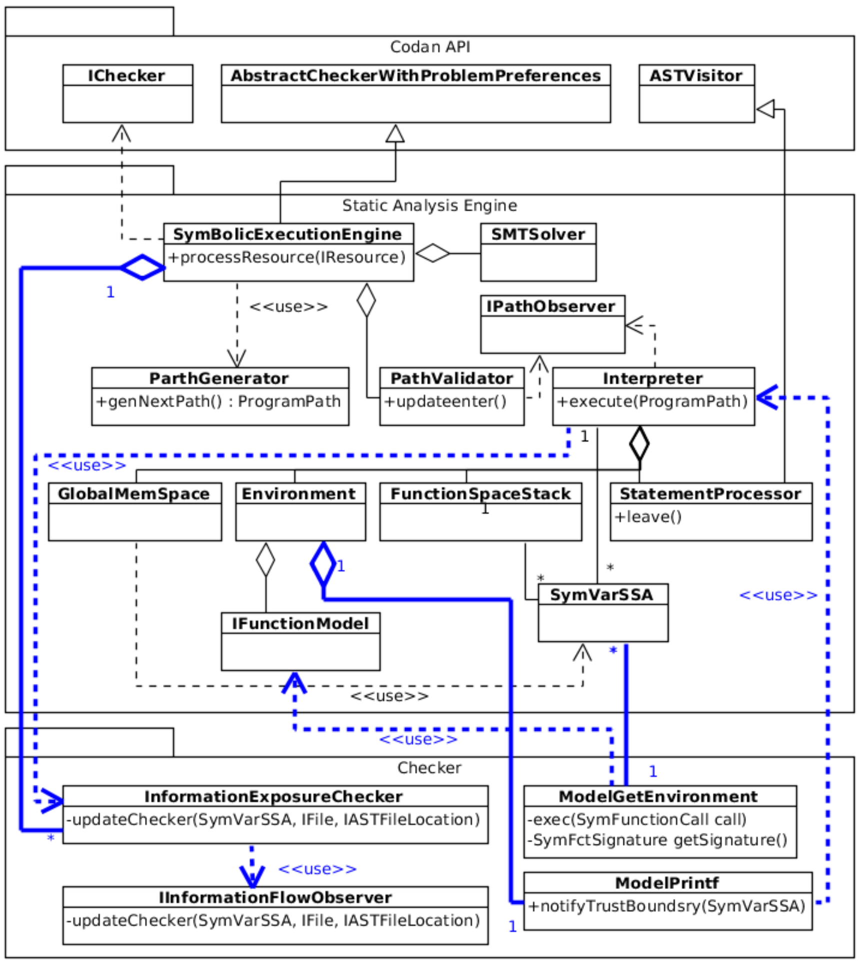


Figure 3.2: The IF checker architecture

be called by the `Interpreter` in order to get the return value of the `getenv()`.
In `getSignature()` the parameters of the `getenv()` are defined and a return type is set. In the `exec()` method we taint the *confidential* value to be the return value of the `exec()` method. The IECC will be attached to the `Interpreter` in the class `SymbolicExecutionEngine` that contains the main program loop, which iterates through all program paths. During loop iteration when the Interpreter reaches the C function `printf()` or other trust-boundaries then the `exec()` method contained in the *sink* function model `ModelPrintf` will be called. Then the `Interpreter` calls the `updateChecker()` method which notifies the IE checker contained in IECC. Details on the IE checker architecture are given in [44].

3.2 Inference Rules for Secure Information Flows

Based on a runtime language interpreter we are handling symbolic variables during static execution. The details are expanded on in the SIBASE working package 5.1.2 [43]; here we are going to give an overview. Our statement processor enforces secure typing of C/C++ expressions,

variables, and statements as the AST is traversed. For each node contained in the new path a statement processor instance will be instantiated. The inference information is constructed for each statement on the fly by enforcing inference rules based on explicit IF's.
The rules themselves are based on the lattice model of Denning [21] in the formulation of Volpano et al. [60]. Mainly, they represent an adaption of the the secure flow typing rules of Volpano et al. (Figure 2 and Figure 3 in [60]) to the setup of two security levels L (low security, public) and H (high security, private), ordered by $L \leq H$. Only those rules were implemented which were required by the examples. In addition, two rules were added that check that the user-given ordering of procedures holds. We are not going to discuss them here and defer the reader to [46].

3.3 Implementation

In this section we are going to present the implementation details of our checker.

```
void CWE526_bad() {
  if(staticFive == 5) {
    /* FLAW: environment variable exposed */
    printLine(getenv("PATH"));
  }
}
```

Figure 3.3: CWE-526 test program's *source*, according to [48].

```
void printLine(const char* line) {
  if(line)
    printf("%s\n", line);
}
```

Figure 3.4: CWE-526 test program's *sink*, according to [48].

In order to propagate the *confidential* return value from the `getenv()` *source* to the `printf()` *sink* (see Figs. 3.3 and 3.4) we had to extend the `StatementProcessor` (SP) class. A `Interpreter` object is instantiated for each new path. A new SP object will be instantiated by the `Interpreter` for every `IASTNode` (`IBasicBlock`) contained in the current path. Thus, propagating only the symbolic variables belonging to one execution path at a time. For each `IASTNode` the corresponding `leave()` methods are called depending on the type of the node. The `leave()` methods are used to traverse each statement AST in a bottom-up fashion. The `leave()` methods are also used for *confidential* variables propagation. The SP extends the `ASTVisitor` class which is an implementation of the visitor design pattern providing top-down (`visit()` methods) and bottom-up (`leave()` methods) traversal of each node contained in the current path. Each `IASTNode` is a C/C++ line (no comment lines are included) originating from the C/C++ test program file. When the SP detects that a symbolic variable or function return variable is *confidential* as each statement is traversed on the current path it tries to propagate the *confidential* variable based on explicit IF. For the CWE-526 test programs MGE is the *source* because from here *confidential* information flows into the program and `ModelPrintf` is the *sink* because here the potential information is leaving the program. When the SP detects the `getenv()` function inside the wrapper function `printLine()` then it adds a new *confidential*

variable in the `Interpreter`. The *confidential* return value comes from MGE, which is the function model of the `getenv()` function call. The *confidential* variable is propagated to `printLine()` as parameter. When the SP reaches the `printLine()` statement a binding call is made. The binding call returns the parameters names of the `printLine()` header function. The new parameter names are needed because these are used inside the `printLine()` implementation. These parameter names are potential *confidential* symbolic variables.
The `printLine()` function header has a single parameter `line`. After we detect `line` in the method header and we know that we are on a potentially reachable path we add a new *confidential* variable called `line` in the `Interpreter`. This means that on this path from the *source* `printLine(getenv("PATH"))` to the *sink* `printf("%s\n",line)` the `getenv("PATH")` *confidential* return value will be assigned to the variable `line` which becomes also *confidential*. This happens when `printLine()` calls the `execute()` method. The implementation contains `printf()`, as presented in Figure 3.4. The SP proceeds until it reaches the `printf("%s\n",line)` node. After reaching this statement the Interpreter will be notified from the function model `ModelPrintf` using `line` as parameter. The interpreter will be notified because the statement `printf("%s\n",line)` is a sink. The `Interpreter` will be called with `resolveOrigSymVar()` and directly afterwards the `getCurrentSSACopy()` method will be called. These methods search in the `Interpreter` for a symbolic variable called `line`. After this call we get a `SymPointerSSA` variable `s` that we send over to the `Interpreter` by calling `ps.notifyTrustBoundary(s)`. The `Interpreter` then calls our previously attached IF checker by calling his `updateChecker(SymVarSSA s, IFile file, IASTFileLocation loc)` method. The `Interpreter` sends to the IF checker the previously found `s` variable, the file, and the location in the file from where it was notified. The IEC checks in the `updateChecker()` method if `s` is *confidential*. If `s` is *confidential* then a new a bug report will be created. For the test programs contained in CWE-534 and CWE-535 the propagation is similar to what we previously presented; only the *sinks* and *sources* are different. A detailed explanation of the checker is contained in [44].

3.3.1 Tainting and Triggering

The implementation of the static analysis engine [32] is based on function models used for behavior description of standard C/C++ library function calls. A function model class contains five methods and implements the interface `IFctModel`:

- the constructor, which sets the used program interpreter,
- the method `getName()` which returns the name of the function
- `getLibrarySignature()` returns the whole function header as it is defined in the C standard library
- `exec(SymFunctionCall call)` which is used for static execution of function calls (variables can be here tainted (Figure 3.5, line 28) and trust boundaries used for notifying a checker)
- `getSignature()` returns a `SymFctSignature` object containing the data types of the function parameters and the return type of the function.

```
private Interpreter ps;

public Mgetenv(Interpreter ps) {
  this.ps = ps;
}

public String getName() {
  return "getenv";
}

public String getLibrarySignature() {
  return "extern char *getenv (const char *name);"
}

public SymFunctionReturn exec(SymFunctionCall call) {
  ArrayList<IName> plist = call.getParams();
  SymPointerOrig isp = ps.getLocalOrigSymPointer(plist.get(0)
   );
  IName nebn = new EnvVarName();
  SymIntOrig sb_size = new SymIntOrig(new ImpVarName());
  SymArrayOrig sb = new SymArrayOrig(nebn,sb_size);
  SymPointerSSA isp_ssa = null;
  try {
    sb.setElemType(eSymType.SymPointer);
    ps.declareLocal(sb);
    ps.declareLocal(sb_size);
    SymArraySSA sb_ssa = (SymArraySSA) ps.
   getLocalOrigSymArray(nebn).getCurrentSSACopy();
    isp_ssa = (SymPointerSSA) ps.ssaCopy(isp);
    isp_ssa.setTargetType(eSymType.SymPointer);
    isp_ssa.setConfidential(true);
    isp_ssa.setTarget(sb);
  } catch (Exception e){
    e.printStackTrace();
  }
  return new SymFunctionReturn(isp_ssa);
}

public SymFctSignature getSignature() {
  SymFctSignature fsign = new SymFctSignature();
  fsign.addParam(new SymPointerOrig(eSymType.SymArray, new
   Integer(1)));
  fsign.setRType(new SymPointerOrig(eSymType.SymPointer, new
   Integer(1)));
  return fsign;
}
```

Figure 3.5: The `getenv()` function model.

Figure 3.5 depicts the function model of `getenv()`. This function model represents a symbolic approximation of the actual behavior of the `getenv()` function in all possible contexts.The difference between the *sink* function model of `printf()` and the *source* function model of `getenv()` is that in the `exec()` method of the `printf()` model class we notify our IF

checker that a trust-boundary is about to be passed and in the `exec()` method of the `getenv()` model we set the return value to *confidential*. Similarly it is implemented for the sinks and *sources* contained in the CWE-534/535 test programs.
The SAE currently contains function models for the following C functions: `atoi()`, `fclose()`, `fgets()`, `fgetws()`, `fopen()`, `free()`, `gets()`, `memcpy()`, `mod()`, `pthread_create()`, `pthread_exit()`, `pthread_join()`, `pthread_mutex_destroy()`, `pthread_mutex_init()`, `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `puts()`, `rand()`, `srand()`, `strcpy()`, `strlen()`, `time()`, `wcscpy()`, `wcslen()`. For the IE test programs the following function models were added: CWE-526 `getenv()` (*source*), `printf()` (*sink*), CWE-534 and CWE-535 `LogonUserA()`, `LogonUserW()` (*sources*), `fprintf()`, `fwprintf()` (*sinks*).
The models are used either to taint a symbolic variable with the tag *confidential* or to notify the IF checker that a *confidential* tagged variable is about to pass a trust-boundary.
Initially [44], the aforementioned function models were hard-coded. In [46], we improved the checker by parsing the information from the user-given annotations in the *source* test files rather than hardcoding the models in the code of the checker.

3.4 Exposure Checker Usage

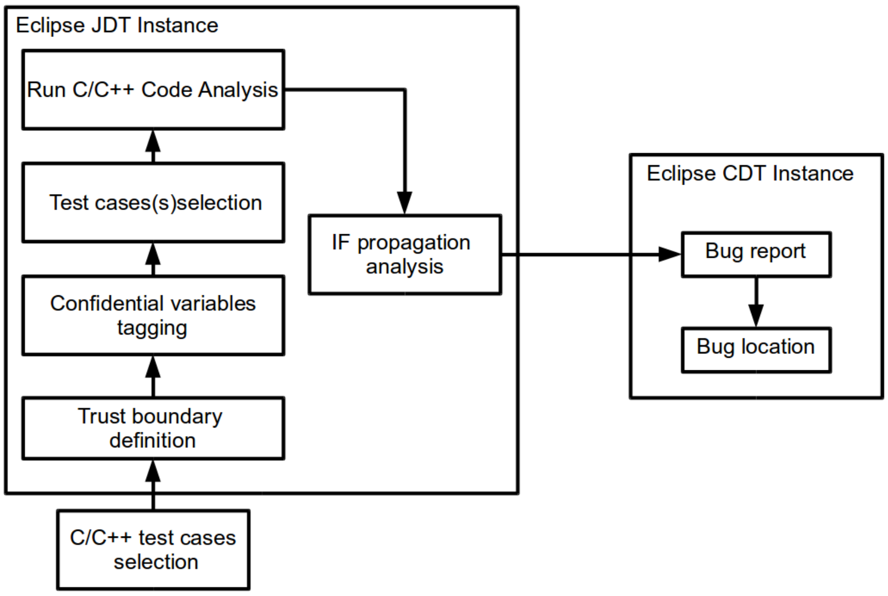
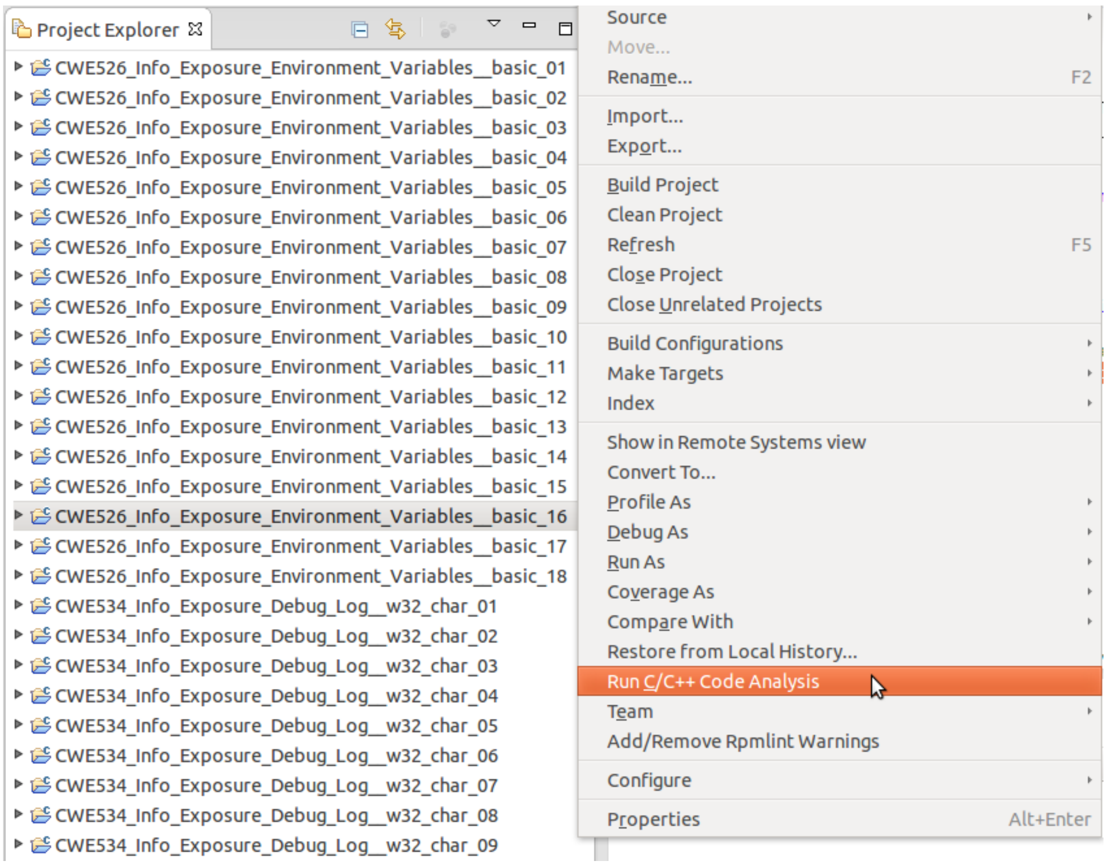


Figure 3.6: Information exposure checker work-flow.

Figure 3.6 depicts the work flow used for running our checker. First, the C/C++ test programs have to be selected and test programs created in the workspace. Second, the trust boundaries have to be defined and *confidential* variables need to be tainted. Third, one or more test programs available in the workspace can be selected and the submenu button *Run C/C++ Code Analysis* needs to be selected.

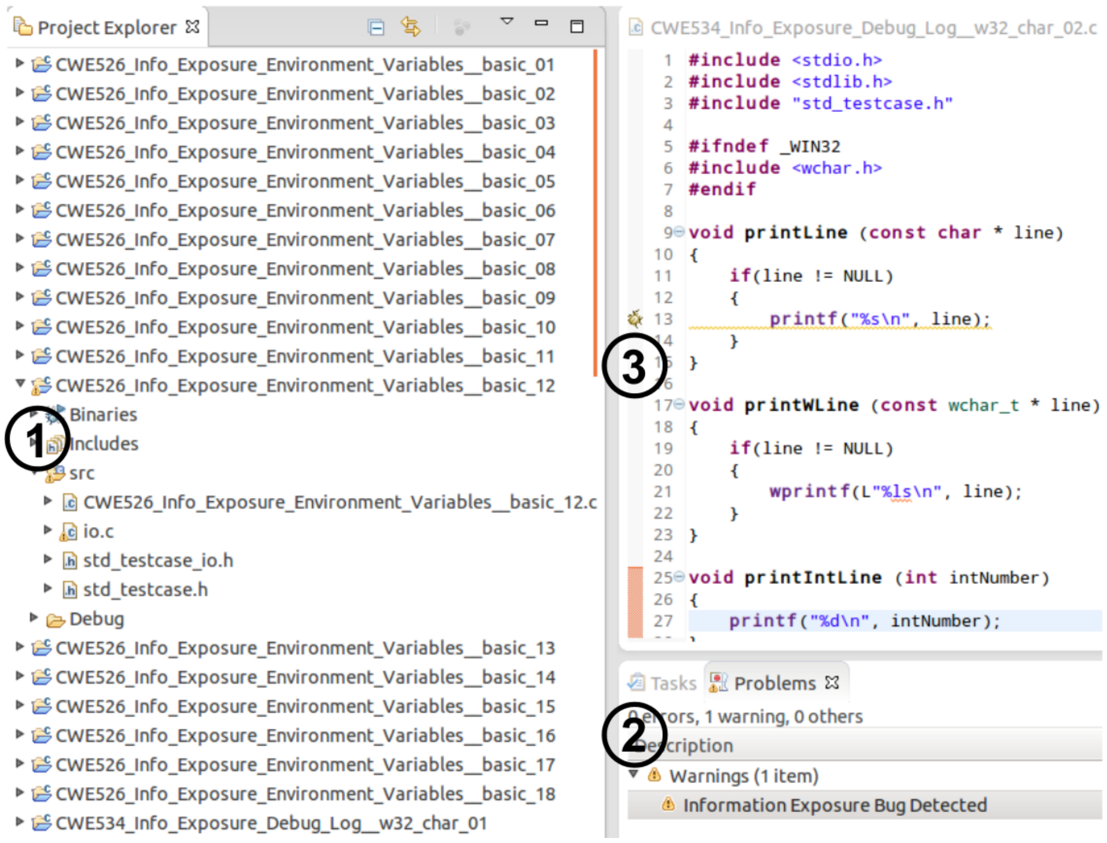
An explicit IF theory for propagating *confidential* variables from trust-boundaries (*sources*) or other program points to trust-boundaries (*sinks*) was used in this paper. Our information-flow checker is based on the SAE that proved to scale for other types of bug checkers and larger test cases as well.
When the static analysis is discovering that a *confidential* variable is about to pass through a previously defined trust-boundary then an IE bug report is issued which is reported in the Problems view inside the second Eclipse CDT instance. By clicking on the bug report available in the Problems view the user navigates to the bug location in the file where the bug was discovered. A bug report is composed of the file and line number where the bug was detected.


Project Explorer

CWE526_Info_Exposure_Environment_Variables__basic_01
CWE526_Info_Exposure_Environment_Variables__basic_02
CWE526_Info_Exposure_Environment_Variables__basic_03
CWE526_Info_Exposure_Environment_Variables__basic_04
CWE526_Info_Exposure_Environment_Variables__basic_05
CWE526_Info_Exposure_Environment_Variables__basic_06
CWE526_Info_Exposure_Environment_Variables__basic_07
CWE526_Info_Exposure_Environment_Variables__basic_08
CWE526_Info_Exposure_Environment_Variables__basic_09
CWE526_Info_Exposure_Environment_Variables__basic_10
CWE526_Info_Exposure_Environment_Variables__basic_11
CWE526_Info_Exposure_Environment_Variables__basic_12
CWE526_Info_Exposure_Environment_Variables__basic_13
CWE526_Info_Exposure_Environment_Variables__basic_14
CWE526_Info_Exposure_Environment_Variables__basic_15
CWE526_Info_Exposure_Environment_Variables__basic_16
CWE526_Info_Exposure_Environment_Variables__basic_17
CWE526_Info_Exposure_Environment_Variables__basic_18
CWE534_Info_Exposure_Debug_Log__w32_char_01
CWE534_Info_Exposure_Debug_Log__w32_char_02
CWE534_Info_Exposure_Debug_Log__w32_char_03
CWE534_Info_Exposure_Debug_Log__w32_char_04
CWE534_Info_Exposure_Debug_Log__w32_char_05
CWE534_Info_Exposure_Debug_Log__w32_char_06
CWE534_Info_Exposure_Debug_Log__w32_char_07
CWE534_Info_Exposure_Debug_Log__w32_char_08
CWE534_Info_Exposure_Debug_Log__w32_char_09

Figure 3.7: Starting the IF checker from the Codan GUI.

Figure 3.7 depicts how our checker is started. The IE checker runs as an Eclipse plug-in project. The checker is launched as a standard Eclipse application. After starting the checker a second Eclipse CDT instance will be launched. For the test cases CWE-526/534/535 we had 90 Test Programs (TPr) contained each in an separate Eclipse CDT project. The TPr's don't have to be executable in order for us to perform static analysis. We run our checker by right-click on the 16th project for example and selecting Run C/C++ Code Analysis as highlighted in Figure 3.7 with the mouse pointer. The sub-menu presented in Figure 3.7 appears by clicking right on one or more selected Eclipse CDT projects.
The Codan API [36] provides a Graphical User Interface (GUI) for running checkers.



```
▸ CWE526_Info_Exposure_Environment_Variables__basic_01
▸ CWE526_Info_Exposure_Environment_Variables__basic_02
▸ CWE526_Info_Exposure_Environment_Variables__basic_03
▸ CWE526_Info_Exposure_Environment_Variables__basic_04
▸ CWE526_Info_Exposure_Environment_Variables__basic_05
▸ CWE526_Info_Exposure_Environment_Variables__basic_06
▸ CWE526_Info_Exposure_Environment_Variables__basic_07
▸ CWE526_Info_Exposure_Environment_Variables__basic_08
▸ CWE526_Info_Exposure_Environment_Variables__basic_09
▸ CWE526_Info_Exposure_Environment_Variables__basic_10
▸ CWE526_Info_Exposure_Environment_Variables__basic_11
▾ CWE526_Info_Exposure_Environment_Variables__basic_12
    Binaries
    Includes
    src
      ▸ CWE526_Info_Exposure_Environment_Variables__basic_12.c
      ▸ io.c
      ▸ std_testcase_io.h
      ▸ std_testcase.h
  ▸ Debug
▸ CWE526_Info_Exposure_Environment_Variables__basic_13
▸ CWE526_Info_Exposure_Environment_Variables__basic_14
▸ CWE526_Info_Exposure_Environment_Variables__basic_15
▸ CWE526_Info_Exposure_Environment_Variables__basic_16
▸ CWE526_Info_Exposure_Environment_Variables__basic_17
▸ CWE526_Info_Exposure_Environment_Variables__basic_18
▸ CWE534_Info_Exposure_Debug_Log__w32_char_01
```

CWE534_Info_Exposure_Debug_Log__w32_char_02.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include "std_testcase.h"
 4
 5 #ifndef _WIN32
 6 #include <wchar.h>
 7 #endif
 8
 9 void printLine (const char * line)
10 {
11     if(line != NULL)
12     {
13         printf("%s\n", line);
14     }
15 }
16
17 void printWLine (const wchar_t * line)
18 {
19     if(line != NULL)
20     {
21         wprintf(L"%ls\n", line);
22     }
23 }
24
25 void printIntLine (int intNumber)
26 {
27     printf("%d\n", intNumber);
```

```
0 errors, 1 warning, 0 others
Description
▾ Warnings (1 item)
    Information Exposure Bug Detected
```

Figure 3.8: IF checker bug report and bug highlighting.

Figure 3.8 highlights the main GUI features available after running the IE checker. These features are marked with circled numbers from ① to ③. Number ① indicates for which project the IE checker was started. Number ② indicates the bug report associated to the found bug. A bug icon near number ③ indicates that at line 13 a buggy statement was detected. Also the whole statement where the bug was detected will be highlighted with an underlining zigzag line.
For the bug report presented in Figure 3.8 with number ② we get the Description (containing a string which the user can configure), Resource (the file where the bug has appeared), Path (path of the file in the project hierarchy were the bug has appeared), Location (the line where the bug was reported) and Type (the type of the reported bug).
The output of the IF checker is a bug report for each detected IE bug. By double clicking on ② the user can navigate in the file at the line number where the bug was detected. One such bug report for the test program 12 contained in the test case CWE-526 is highlighted in Figure 3.8 with number ② and the file location (file name and line number) of the bug with number ③.
Another Codan API feature used for displaying bug reports is represented by the possibility to configure bug reports as *Warnings*, *Errors* or *Infos*, as presented in Figure 3.9.

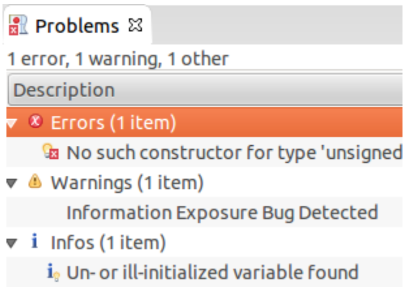

Problems
1 error, 1 warning, 1 other
Description
Errors (1 item)
No such constructor for type 'unsigned
Warnings (1 item)
Information Exposure Bug Detected
Infos (1 item)
Un- or ill-initialized variable found

Figure 3.9: Codan report types.

Figure 3.9 presents bug reports in a tree based view where every bug is classified based on one of the following three categories: Warnings (yellow triangle icon), Errors (red circle icon) or Infos (blue "i" symbol icon). The warning (Information Exposure Bug) presented in Fig. 8a corresponds to the bug report presented in Fig. 7. The "Errors" and "Infos" reports presented in Figure 3.9 are not related to our IE checker. Codan reports use three different bug icons.
By clicking on the generated Information Exposure Bug report presented in Figure 3.9 the appropriate file containing the bug is opened in the main view and the mouse cursor will point to the line number containing the bug as presented in Figure 3.8, number ③. The Codan API offers the possibility to configure each checker to be launched in different modes as presented in Figure 1.1. This bug triggering features can help a developer to control how and when Eclipse will trigger the bug detection analysis, thus, helping to avoid bug insertion during software development. A detailed explanation of user interface can be found in [44].

3.4.1 Source Code Editor

We implemented a source code editor which offers annotation language proposals which are context *sensitive* with respect to the position of the currently edited syntax line. If for example, a C expression is not properly parsed then the proposal mechanism would not work from that line on until the end of the file. Thus, the editor suggestions work only if the whole file is parsed without errors. Particularities of the source code editor are given in [46].

Chapter 4: Experiments

This Chapter presents first, the test methodology used during our experiments 4.1 and second, the obtained results and constraints 4.2.
The goal of our empirical evaluation is to assess the efficiency of our IF checker in terms of number of detected false-negatives, false-positives, true-positives and execution time. At the same time we want to highlight to what extent our research work is significant. We present results *1)* for the case that the function annotations are hard-coded, and *2)* for the case that code annotations for the functions are automatically loaded. The evaluation was performed using the IE checker presented in Section 3 and the Juliet test cases CWE-524/534/535.

4.1 Methodology

The test cases CWE-526/534/535 were selected because they contain information exposure bugs which we want to detect. These test cases are publicly available in the last version of the Juliet test suite [48]. CWE-526 contains 18 Test Programs (TPr), CWE-534 contains 36 TPr, and CWE-535 contains 36 TPr. For all the test programs contained in CWE-526/534/535 we created a separate Eclipse CDT project. The test programs were then inserted in one Eclipse workspace. In Fig. 3.8 some of the analyzed test programs can be observed.
The IE checker was run automatically for each test program available in the workspace by selecting once all the Eclipse CDT projects available in the workspace and selecting the sub-menu Run C/C++ Code Analysis. We measured the time from the moment of clicking the sub-menu button until all the projects in the workspace were completely analyzed. We also measured the execution time for the test programs belonging to one test case. We report the intermediate execution time (for test programs belonging to one test case), total execution time, number of true positives, false negatives and false positives in Section 4.2. For measuring the time between the moment when the analysis was started and the moment when all the test programs in the workspace were analyzed we used the following time stopping criteria. For determining the total execution time we monitored the event when there was no longer output messages in the console. For determining the intermediate execution times for test programs belonging to one of the test cases CWE-526/534/535 we monitored when all the test programs had a bug icon attached to them.
We new in advance which test programs should have an bug icon attached to them after running the static analysis and which test programs should not have a bug icon (for 5 out of 90 test programs it was not possible to perform the static analysis, this is reported in the next section) attached from previous runs. One such bug icon is presented in Fig. 3.8 above number ① and represents a yellow triangle with an exclamation mark inside.
In addition, we analyzed three simplified C programs in which the prescribed order of operations is broken. They stem from [40] (the programs contain a hard-coded password, which it uses

for its own inbound authentication or for outbound communication to external components), CWE-325 [38] (the software does not implement a required step in a cryptographic algorithm, resulting in weaker encryption than advertised by that algorithm), CWE-666 [39] (the software performs an operation on a resource at the wrong phase of the resource's lifecycle, which can lead to unexpected behaviours). For each of these three programs, the checker described in Chapter 3 was suitably adapted.

4.2 Results and Constraints

Table 4.1, contains the results obtained by analyzing CWE-526, CWE-534 and CWE-535 with our IE checker. Table 4.1 contains the following abbreviations: Test Program (TPr), FP (False-Positives), FN (False-Negatives), True Positives (TP) for programs without the C *goto* statement included, Total True-Positives (TTP) per test case (all programs included), and Total Execution Time in seconds (TET[s]) per test case. The used system was Ubuntu 12.04 LTS, Kernel 3.8.0-35-generic, 64-bit, Intel® Core^{TM} i7-4770 CPU @3.40GHz $\times$ 8, 16 GB RAM.

Test case	**TPr**	**FP**	**FN**	**TP**	**TTP**	**TET[s]**
CWE-526	18	0	0	17	18	30
CWE-534	36	0	0	34	36	46
CWE-535	36	0	0	34	36	45
Total	90	0	0	85	90	121

Table 4.1: IF checker running time results without annotated code.

Table 4.1 presents the timing results for running our tools with hard-coded annotations inside our function models. Our tool found 85 TP out of 90 TP present in the used test cases. We were able to detect all IE bugs. It was not possible to test all test programs available in the test cases because the Codan API is not supporting the building of the CFG for source code containing C *goto* statements, e.g., *goto stop;*.
The test cases CWE-526/534/535 contain 1, 2, and respectively 2 test programs containing the C *goto* statement. In total 5 $(1 + 2 + 2)$ out of 90 test programs were not analyzable. Thus, we reached 94.44% test coverage. We think that if this limitation will be removed from Codan API releases then 100% test coverage is achievable.
In [46], we improved the checker by automatically loading the annotations from source code.

Test case	**TPr**	**FP**	**FN**	**TP**	**TTP**	**TET[s]**
CWE-526	18	0	0	17	18	37
CWE-534	36	0	0	34	36	88
CWE-535	36	0	0	34	36	86
Total	90	0	0	85	90	213

Table 4.2: IF checker running time results with annotated code.

Table 4.2 presents the timing results of running the same test programs as above through the improved checker version.

These running times are higher due to the overhead of loading and interpreting the annotated library in which sinks and sources are annotated for each of the test programs. This library gets loaded each time the plug-in is started—which is due to the current Eclipse plug-in implementation. If the annotated library would be loaded only once, the overhead would be approximately only 1.1%. More details on the running times of the improved checker are found in [46].

Chapter 5: Conclusion and Future Work

5.1 Conclusion

We successfully proved that our IE checker can be used for detecting IE bugs and at the same time we have shown to what extent our work is significant by comparing it with related research work. Comparison with related work is not presented in this report, only in the original paper (see [44]).
The Codan API was used for parsing source files, dealing with project resources and interpreting C/C++ code. Bug location marking was easily implemented using the markup capabilities offered by the Codan API.
The AST traversing mechanisms offered the possibility to focus more on static analysis rather than on re-implementing utility functions for manipulating AST nodes. Building of CFG for test programs containing C *goto* statements should be possible in future Codan API releases, thus, removing one of the current implementation constrains.
We successfully used the SAE engine for propagating symbolic variables tags based on explicit information flow. The SAE engine was easily extendable and offered the possibility of plugging our IE checker in the existing language interpreter. The computational overhead introduced by SMT-LIB [6] statement construction and the calling of an external command line tool could be avoided by using an SMT solver which has an API compatible with our development language [16] or [20].
The static definition of *sinks*, *sources* and *confidential* variables can be made more user-friendly by going via a UML model of the code as we are going to sketch now. First, one can automatically transform source code into UML state chart diagrams. Second, one can attach extra security information in these diagrams to the methods and attributes. Third, one can automatically generate source code with auxiliary assertions stemming from the prior UML description. These areas are going to be expanded on in the SIBASE working package 5.1.3 [45]. Moreover, the definition of *sinks*, *sources* and *confidential* variables can be further automated by providing annotated test cases or libraries containing this information. Alternatively, one can load a configuration file containing the specifications.
We think that static analysis is worthwhile to be used for detecting IE bugs related to security concerns and future research in this area is needed.

5.2 Future Work

In future we plan to do research in the area of annotating whole C/C++ libraries and reusing these annotated libraries during static analysis for trust-boundaries definition and symbolic variable

tainting. We envisage that advanced checks for *sinks* and *sources* on a potential bug prone path could reduce the number of candidate paths.
The process of manually determining which function models (trust-boundaries) could produce IE leaks could be automated by preprocessing the *source* code and determining which functions could represent potential candidates for *sinks* and *sources*. This could be based on a previously annotated C/C++ library with annotation tags attached to function headers. Information propagation could be used in order to detect other potential *sink*, *sources*, etc., thus, fully automating the process of trust-boundary definition and initial variable tainting.

Chapter 6: Publication List

1. Paul Muntean, Claudia Eckert, and Andreas Ibing. Context-sensitive detection of information exposure bugs with symbolic execution. In *International Workshop on Innovative Software Development Methodologies and Practices* (*InnoSWDev*), Nov. 2014, ACM, [44].

2. Paul Muntean, Adnan Rabbi, Andreas Ibing, and Claudia Eckert. Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code. In *IEEE International Workshop on Model-Based Verification & Validation* (*MVV*), 2015, IEEE, [46].

Bibliography

[1] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded Model Checking of Software Using SMT Solvers Instead of SAT Solvers. In *Proceedings of the 13th International Conference on Model Checking Software*, SPIN, pages 146–162, Berlin, Heidelberg, 2006. Springer-Verlag.

[2] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2011.

[3] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[4] Nathaniel Ayewah and William Pugh. The Google FindBugs Fixit. In Paolo Tonella and Alessandro Orso, editors, *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis (ISSTA)*, pages 241–252. ACM, July 2010.

[5] Thomas Ball and Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In Matthew B. Dwyer, editor, *Model Checking Software, 8th International SPIN Workshop*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122. Springer, 2001. Online available at: `http://dx.doi.org/10.1007/3-540-45139-0_7`.

[6] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, the University of Iowa, 2010. Online available at: `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf`.

[7] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A Static Analyzer for Finding Dynamic Programming Errors. *Software - Practice and Experience (SPE)*, 30(7):775–802, 2000.

[8] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Richard Draves and Robbert van Renesse, editors, *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX Association, December 2008. Online available at: `http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf`.

[9] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically Generating Inputs of Death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 322–335. ACM, 2006.

[10] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, 56(2):82–90, 2013.

[11] Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet. Taint Dependency Sequences: A Characterization of Insecure Execution Paths Based on Input-Sensitive Cause Sequences. In *Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 371–380. IEEE Computer Society, 2010.

[12] Richard M. Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF)*, pages 186–199. IEEE Computer Society, 2009.

[13] Roderick Chapman and Adrian Hilton. Enforcing Security and Safety Models with an Information Flow Analysis Tool. *ACM SIGAda*, 24(4), 2004.

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-Vivo Multi-Path Analysis of Software Systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2011.

[15] Stephen Chong, Andrew Clifford Myers, Nate Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java + Information Flow, July 2006. Online available at: `http://www.cs.cornell.edu/jif/`.

[16] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: an Interpolating SMT Solver. In *SPIN*, pages 248–254, 2012.

[17] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT 5 SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.

[18] Ellis Saul Cohen. Information Transmission in Sequential Programs. In Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, editors, *Foundations of secure computation*, pages 301–339, Gerogia Tech, Atlanta, Georgia 30332, US, December 1978. Online available at: `https://smartech.gatech.edu/bitstream/handle/1853/40598/g-36-619_142482.pdf?sequence=1`.

[19] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 57–68, 2002.

[20] Leonardo de Moura and Nikolaj. Bjørner. Z3: An efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[21] Dorothy Elizabeth Robling Denning. A Lattice Model of Secure Information Flow. *Communications of the ACM*, 19(5):236–243, 1976.

[22] Dorothy Elizabeth Robling Denning and Peter James Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):503–513, 1977.

[23] Jeffrey Stewart Fenton. Memoryless Subsystems. *Computer Journal*, 17(2):143–147, May 1974.

[24] Jeffrey S. Foster. CQual, 2016. Online available at: `http://www.cs.umd.edu/~jfoster/cqual/`.

[25] Jeffrey S. Foster, Tachio Terauchi, and Alexander Aiken. Flow-Sensitive Type Qualifiers. In Jens Knoop and Laurie J. Hendren, editors, *Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM, 2002.

[26] Marco Guarnieri, Paul El-Khoury, and Gabriel Serme. Security Vulnerabilities Detection and Protection using Eclipse. In *6th Workshop of the Italian Eclipse Community (ECLIPSE-IT)*, September 2011.

[27] Christian Hammer, Jens Krinke, and Gregor Snelting. Information Flow Control for Java Based on Path Conditions in Dependence Graphs. In *In IEEE International Symposium on Secure Software Engineering (ISSSE)*, pages 1–10. IEEE, 2006.

[28] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, ISBN: 978-0-521-89957-4, 2009.

[29] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software Verification with BLAST. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[30] David Hovemeyer and William Pugh. Finding Bugs is Easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[31] Sebastian Hunt and David Sands. On Flow-sensitive Security Types. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 79–90. ACM, January 2006.

[32] Andreas Ibing. SMT-Constrained Symbolic Execution for Eclipse CDT/Codan. In *Workshop on Formal Methods in the Development of Software (WS-FMDS)*, 2013.

[33] Andreas Ibing. SIBASE Report TP 5.2 – AP 5.2.1: Symbolic Execution Engine, December 2014. Available as file `AP521_SymbolicExecutionEngine.pdf` of the SIBASE project.

[34] James Cornelius King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19:385–394, July 1976.

[35] Dharmender Singh Kushwaha and A. K. Misra. Software Test Effort Estimation. *SIGSOFT Softw. Eng. Notes*, 33(3):6:1–6:5, May 2008. Online available at: `http://doi.acm.org/10.1145/1360602.1361211`.

[36] Elena Laskavaia. Codan – C/C++ Static Analysis Framework for CDT], journal = EclipseCon, year = 2011, note = Online available at: `http://www.eclipsecon.org/2011/sessions/index0a55.html?id=2088`.

[37] CWE-200: Information Exposure. Online available at: `http://cwe.mitre.org/data/definitions/200.html`.

[38] Mitre. CWE-325: Missing Required Cryptographic Step. Online available at: `http://cwe.mitre.org/data/definitions/325.html`.

[39] Mitre. CWE-666: Operation on resource in wrong phase of lifetime. Online available at: `http://cwe.mitre.org/data/definitions/666.html`.

[40] Mitre. CWE-259: Use of hard-coded password, 2014. Online available at: `http://cwe.mitre.org/data/definitions/259.html`.

[41] Scott Moore and Stephen Chong. Static Analysis for Efficient Hybrid Information-flow Control. In *Proceedings of the IEEE 24th Computer Security Foundations Symposium (CSF)*, pages 146–160, 2011.

[42] Michał Moskal, Nikolai Tillmann, Manuel Fahndrich, and Peli de Halleux. TouchDevelop - Programming Cloud-Connected Mobile Devices via Touchscreen. Technical Report MSR-TR-2011-49, Microsoft, April 2011. Online available at: `https://www.microsoft.com/en-us/research/publication/touchdevelop-programming-cloud-connected-mobile-devices-via-touchscreen/`.

[43] Paul Muntean. SIBASE Report TP 5.1 – AP 5.1.2: Modeling of information-flow restrictions, Technical University of Munich, Technical Report, TUM-I1416, September 2014, September 2014. Online available at: `https://mediatum.ub.tum.de/doc/1244626/1244626.pdf`.

[44] Paul Muntean, Andreas Ibing, and Claudia Eckert. Context-Sensitive Detection of Information Exposure Bugs with Symbolic Execution. In *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices (InnoSWDev)*, pages 84–93. ACM, November 2014.

[45] Paul Muntean and Alexander Malkis. SIBASE Technical Report TP 5.1 – AP 5.1.3: Automatic Security Checks on the Model Level, 2016. (In preparation.).

[46] Paul Muntean, Adnan Rabbi, Andreas Ibing, and Claudia Eckert. Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code. In *Model-Based Verification and Validation (MVV)*, pages 128–137. IEEE, 2015.

[47] Andrew Clifford Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, January 1999.

[48] National Institute of Standards and Technology (NIST). Juliet test suite v1.2 for C/C++. Online available at: `http://samate.nist.gov/SRD/testsuites/juliet/Juliet_Test_Suite_v1.2_for_C_Cpp.zip`.

[49] National Vulnerability Database (NVD). Online available at: `https://web.nvd.nist.gov/view/vuln/search-results?adv_search=true&pub_date_start_month=9&pub_date_start_year=2014&pub_date_end_month=9&pub_date_end_year=2015&cve_id=`.

[50] OWASP. OWASP top 10, 2007. Online available at: `https://www.owasp.org/index.php/Top_10_2007`, retrieved on 11 Nov 2015.

[51] Marco Pistoia, Robert James Flynn, Larry Koved, and Vugranam C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In *Proceedings on the 19th European Conference European Conference on Object-Oriented Programming (ECOOP)*, pages 362–386, 2005.

[52] Bill Pugh, Andrey Loskutov, and Keith Lea. FindBugs, 2015. Online available at: `http://findbugs.sourceforge.net`.

[53] Xiao Qu and Brian Robinson. A Case Study of Concolic Testing Tools and their Limitations. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 117–126. IEEE Computer Society, September 2011.

[54] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium (CSF)*, pages 186–199. IEEE Computer Society, July 2010. Online available at: `http://doi.ieeecomputersociety.org/10.1109/CSF.2010.20`.

[55] Andrei Sabelfeld and Alejandro Russo. From Dynamic to Static and Back: Riding the Roller Coaster of Information-flow Control Research. In *International Conference on Perspectives of System Informatics*, 2009.

[56] Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-Input Dependence Analysis via Graph Reachability. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 25–34. IEEE Computer Society, 2008.

[57] Vincent Simonet. The Flow Caml System: Documentation and User's Manual. Technical report, INRIA, July 2003.

[58] Vincent Simonet. The Flow Caml System. Software Release, July 2003. Online available at: `http://www.normalesup.org/~simonet/soft/flowcaml/`.

[59] Gregor Snelting, Torsten Robschink, and Jens Krinke. Efficient Path Conditions in Dependence Graphs for Software Safety Analysis. *ACM Transactions on Software Engineering Methodology*, 15(4):410–457, 2006.

[60] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):167–187, 1996.

[61] Oleg Šelajev, Erkki Lindpere, Sigmar Muuga, and Oliver White. Using Eclipse for Java development: A helpful introduction to the world's most-used IDE, October 2013. Online available at: `https://zeroturnaround.com/rebellabs/using-eclipse-for-java-development`.

[62] Chris Wysopal. Mobile App Top 10 List, 2010. Online available at: `https://www.veracode.com/blog/2010/12/mobile-app-top-10-list`.

[63] Xusheng Xiao, Nikolai Tillmann, Manuel Fahndrich, Jonathan de Halleux, and Michał Moskal. Transparent Privacy Control via Static Information Flow Analysis. Technical Report MSR-TR-2011-93, Microsoft, August 2011.

[64] xTend, a Programming Language on Top of Java. Online available at: `http://www.eclipse.org/xtend`.

[65] xText, a Framework for Development of Programming Languages and Domain-specific Languages. Online available at: `http://www.eclipse.org/xtext`.

[66] Steve Zdancewic and Andrew Clifford Myers. Robust Declassification. In *Proceedings of the IEEE Computer Security Foundations Workshop (CSFW)*, pages 15–23, 2001.