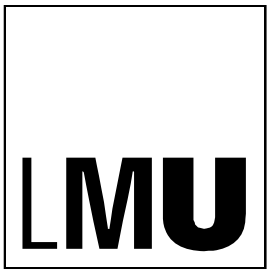


Department of Computer Science
Software Engineering and Programming Languages
Augsburg University

Merging Static Analysis and Model Checking for Improved Security Vulnerability Detection

Wolf-Steffen Rödiger

Master's Thesis in the Elite Graduate Program Software Engineering

Merging Static Analysis and Model Checking for Improved Security Vulnerability Detection

Statische Analyse und Model Checking für eine verbesserte Erkennung von Sicherheitslücken

Autor:	Wolf-Steffen Rödiger
Matrikelnummer:	1116429
Abgabe der Arbeit:	22. Dezember 2011
Erstgutachter:	Prof. Wolfgang Reif
Zweitgutachter:	Prof. Alexander Knapp
Betreuer:	Dr. Ralf Huuck, Ansgar Fehnker

Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 22. Dezember 2011 Wolf-Steffen Rödiger

Acknowledgments

I am very grateful to Ralf Huuck and Ansgar Fehnker for giving me the chance to write my master's thesis at NICTA in Sydney. I would also like to thank the Goanna team for their continuous support: Mark Bradley, Paul Steckler, and Dominic Gurto. Last but not least thanks for the ever so interesting lunch topics to Robert van Glabbeek and Peter Höfner.

A special thanks to my mother for her ongoing support and the steady supply with tea, which made this thesis possible in the first place.

Abstract

Some of the most dangerous and exploitable security problems are caused by insufficiently validated user input. This includes command injections, path traversals, and format string flaws which are part of the 2011 CWE/SANS list of the "Top 25 Most Dangerous Software Errors". These weaknesses have the common characteristic to pass user input obtained from an external source to a vulnerable function without appropriate input validation. I will present a new technique which combines data flow analysis and model checking to find weaknesses of this kind. The data flow analysis tracks the propagation of user input in the program and tags statements which are influenced by it. Model checking is then used in a second step to eliminate false positives and to produce a readable counter-example trace. A summary-based inter-procedural analysis extends the approach to problems which span multiple functions and compilation units. It achieves an average detection rate of 86 % for applicable test cases of the SATE IV benchmark. The analysis evaluates 1,000 lines of code in 2.3 to 7 seconds measured on the basis of four large open source projects. My approach is easily extendable to similar weaknesses and is built into the top notch bug finding tool Goanna.

Contents

1. Introduction

In 1994 it was possible to obtain a root shell on any SGI computer running the operating system IRIX by using the following username in the login screen: [HLV05]

```
FRED; xterm&
```

This is a classic example of a *command injection* vulnerability caused by insufficiently validated user input. The underlying code responsible for this problem is:

```
1 char buf[1024];
2 snprintf(buf, "system lpr -P %s", user_input, sizeof(buf)-1);
3 system(buf);
```

Listing 1.1: Vulnerable code used for user authentication in IRIX (taken from [HLV05]).

IRIX expects the user to provide an alphanumerical username but never enforces that the input indeed meets this requirement. The login procedure does not remove meta-characters from the input string which have a special meaning for the command line. In this case the input `FRED; xterm&` contains a semicolon which separates two commands. It is appended to the string `system lpr -P` in line 2 of the source code snippet. The first part of the resulting string is a command that tries to authenticate the user. However, the part after the semicolon is executed as a separate command. The instruction `xterm&` opens a new shell with the same privileges. An unauthorized user was able to use this input in any IRIX login screen to gain access to a root shell which implies complete control over the affected system.

1.1. Security

Security concerns the protection of systems and information.This includes their confidentiality, integrity, and availability. The increasing importance of the Internet and the interconnection of machines have made security a key topic in computer science.

Security problems are one reason for the popularity of computer security: large numbers of vulnerabilities are reported, potential cyber terrorism threatens states, and privacy issues affect personal information. On the other hand, security technologies like encryption, digital signatures, and digital cash enable new applications.

1.1.1. Objectives

The three basic goals of security are confidentiality, integrity, and availability. However, there are several other security-related objectives. The following definitions are based on a lecture by David Basin and Ueli Maurer [BM09].

Confidentiality Information should be accessible only to those who are authorized to see it. An attacker can exploit a *SQL injection* vulnerability to access confidential data. An example for this is credit card information stored in the database of an internet shop.

Integrity Information should not be modified without proper authentication. Online banking is an example for a target of *man-in-the-middle* attacks. Attackers try to intercept and redirect the communication between customers and their bank. They intend to modify bank transfers to get hold of the money.

Availability Information and computer systems should be available when needed. This is violated by a *denial-of-service* attack which makes a network resource unavailable to its users.

Authenticity A message should originate from the claimed sender. In the previous online banking example the attacker impersonates the bank customer. Authenticity requires that the bank is able to detect illegitimate bank transfers which do not originate from its customer.

Non-repudiation It should be impossible for the sender to deny a message. A bank customer should not be able to deny a bank transfer and reclaim the money from a bank. The bank should be able to proof that the customer is responsible for the money transfer.

Auditability Previous states of a system or certain aspects of it should be reconstructable. This property is important to reconstruct an attack on a system or to reproduce the steps of a program which led to a security failure.

Accountability An entity should be accountable for its actions. This is closely related to non-repudiation and auditability. An entity accounts for an action if it is not able to deny it and if this action is reconstructable.

Privacy Privacy is security applied to personal information. A person should be able to control which personal information is generated, stored, and processed, and by whom. Privacy includes confidentiality of private information but extends to other objectives as well.

Anonymity Anonymity is an aspect of privacy which refers to hiding the identity of a person from other entities. This obviously conflicts with authenticity and non-repudiation.

1.1.2. Measures

Several measures are employed to guarantee the before mentioned objectives. This includes technical solutions like cryptography, organizational issues like security policies, and legal regulations including liability and insurances.

Some of these objectives including confidentiality, integrity, availability, and authenticity are threatened by vulnerabilities in software products. These flaws allow attackers to gain access to secret data, modify it, crash computer systems, and spoof identities. This thesis focuses on static analysis techniques to detect vulnerabilities early in the development process.

1.2. My Approach

Input data is called *tainted* if it is provided by the end user or an unknown third party—e.g., over the network. The goal of my approach is to check if the tainted data can propagate through a program until it reaches a function or statement that is vulnerable to it. These types of vulnerabilities include some of the most common security problems such as command injections (see 3.2.2), format string vulnerabilities (see 3.2.5), and buffer overflows (see 3.2.7).

My approach combines data flow analysis and model checking. A custom data flow analysis propagates the taint information in a program. Afterwards we know which statements can be influenced by user input. Model checking removes invalid paths in a second step. The resulting analysis is fully inter-procedural, flow-sensitive, and sound with respect to its rule set, with the exception of pointer aliasing and global variables.

1.2.1. Integration with Goanna

My approach is implemented as an extension to the bug finding tool Goanna [NIC11] developed at NICTA. Goanna applies static analysis and model checking to perform a full path analysis of C and C++ programs. Programming errors are expressed in computational tree logic (CTL) and are checked against the labeled control flow graph (CFG) of the program. The taint analysis presented in this work takes advantage of several advanced features provided by the Goanna framework apart from its model checking capabilities. This includes the elaborate pre-processing of the source code which generates the abstract syntax tree and CFG. Furthermore, Goanna provides a framework to create a summary-based inter-procedural analysis which I used for my taint analysis.

1.2.2. Contributions

My contribution is a taint analysis which is based on an extensive knowledge base of user input functions, taint transfer functions, and vulnerable functions (see Appendix B). I developed a generic framework for lattice-based data flow analyses and used it to specify several data flow analyses which form the core of the taint analysis. The summary-based framework provided by Goanna allowed me to lift my analysis to an inter-procedural level. I evaluated my approach with the Sate IV benchmark and four large open source projects (see Chapter 8).

1.3. Static Program Analysis

My approach tackles problems similar to the introductory example using two core techniques of static program analysis: data flow analysis and model checking. In this section I will introduce static analysis and explain how it can be applied to detect security vulnerabilities.

A static program analysis evaluates software without executing it. This stands in contrast to a dynamic analysis which executes the code and observes its runtime behavior. Static analysis is performed mostly on the source code and sometimes on the object code of the computer program. It is conducted by an automatic tool which distinguishes it from manually performed code reviews.

Common techniques used in static analysis include model checking and data flow analysis. Model checking decides if a model of a system satisfies a given property. A data flow analysis computes information that reaches the program points of a program.

1.3.1. Areas of Application

Compiler optimizations, software metrics, bug finding, and software security are all areas where static analysis is employed. Static analysis applied to software security aims at finding vulnerable code patterns or control flow paths which lead to vulnerable states. A static analyzer is a valuable tool for security audits as it contains predefined set of rules which are tailored to check for vulnerabilities that are commonly found in software.

1.3.2. Discussion

Static analysis provides many opportunities including the detection of problems early in the development process. However, static program analysis does not solve all problems and has some drawbacks which have to be considered. I will address both advantages and limitations of static program analysis in the following two sections.

Advantages Many programming errors—including missing semicolons, additional parentheses, typing errors, and more—are detected by the compiler at build time and are in turn fixed by the programmer. This quick response to code changes stands in contrast with security vulnerabilities which often remain undetected for years. Moreover, the longer a vulnerability remains in the program the more expensive it can get to fix it [CM04].

Static analyzers promise to identify common security problems before the program is released and even early in development. Many issues stem from coding problems which are repeated very often—an example is the classic null pointer dereference. A static analyzer uses a predefined set of rules to target this kind of previously known problems.

A manual audit can be seen as a first step towards static analysis. The reviewers analyze the source code without running the program and analyze it based on their knowledge of common security problems. However, a code audit is time-consuming and the reviewers need a broad and up-to-date knowledge of possible vulnerabilities. Automatic tools in comparison offer a higher ease of use and do not require the user to have detailed security expertise.

A program can be analyzed even before it reaches a state where it would be reasonable to begin testing. Testing requires the program to be actually run, possibly with drivers and stubs while static analysis can be performed on separate modules or even unfinished code. Testing needs explicit test cases to cover specific code paths. This makes it difficult to find faults which occur only with very specific input data [Bla09]. Static analyzers have the advantage here as they check the source code independently of any particular execution.

The following quote from Wagner *et al.* [WFBA00] emphasizes this important difference between dynamic testing and static analysis:

> "The fundamental problem with dynamic testing is that the code paths of greatest interest to a security auditor—the ones which are never followed in ordinary operation—are also the ones that are the hardest to analyze with dynamic techniques."

Limitations Static analyzers are limited by their fixed set of rules and their model of a program's semantics. They often ignore complex programming language constructs like function pointers or embedded assembler code.

Static analyzers can not guarantee the absence of bugs when they report no warnings. In fact, there will never be a static analyzer which is able to find all security vulnerabilities in a program without reporting spurious warnings. Static analysis problems including bug finding are generally undecidable. They can easily be reduced to the halting problem which is proven to be undecidable [Tur37]. Static analyzers often make simplifications or use heuristics to overcome this. Unfortunately, either false positives, false negatives, or both are the result.

A *false negative* is a vulnerability which the tool doesn't report and a *false positive* is a warning about a vulnerability which is not actually present in the program. A tool is called *sound* if it warns about every existing vulnerability and *complete* if it reports only warnings which are in fact vulnerabilities. Therefore soundness refers to the absence of false negatives and completeness to the absence of false positives. A tool can never be both sound and complete according to the before mentioned undecidability of the halting problem.

Even though the predefined rule set of a static analysis tool relieves the user of the burden to know about every possible security vulnerability, its output still needs careful evaluation: Several software vulnerabilities were reported[1] where the responsible line of code was annotated to suppress a warning of a security analyzer. Apparently, a programmer erroneously marked a correct warning as a false positive. No static analysis tool can free the user from the task to distinguish between correct warnings and false positives. However, tools can aid the user in this task by providing detailed information for every reported warning.

In general, static analysis tools are limited to security vulnerabilities which are part of the source code of a program. The program's high-level design and architecture are out of its scope. Some very common security problems like weak passwords cannot be addressed by static analysis at all.

1.4. Outlook

Diverse static analysis tools have been constructed which focus on security problems. The methods include lexical analysis, annotation-based analysis, constraint systems, type systems, data flow analysis, and model checking. Each technique is covered with a corresponding tool in Chapter 2. Afterwards, I will compare them with respect to three important properties.

My approach aims at security vulnerabilities caused by user input similar to the example at the beginning of this introduction. This includes format string vulnerabilities, uncontrolled memory allocations, command injections, path traversals, and more. These weaknesses are classified and described in Chapter 3.

I combine data flow analysis and model checking to gain the best of both worlds. The theory behind these static analysis techniques is described in Chapter 4. The integration in the Goanna tool is presented in Chapter 5. Chapter 6 describes the intra-procedural part of my approach which is extended to a fully inter-procedural analysis in Chapter 7. The resulting taint analysis is evaluated and compared to other approaches in Chapter 8. At last, I will draw some conclusions and discuss possible future work in Chapter 9.

[1] Security Advisory 06.26.07 for RealPlayer, see `http://www.securityfocus.com/archive/1/472295`

2. Related Work

This chapter presents several approaches which apply static analysis to the security domain. Each approach is illustrated with at least one representative tool. The different techniques are tailored for different types of vulnerabilities which I will point out accordingly. Afterwards, I will classify the different tools according to their precision, scope, and sensitivity to provide a basic comparison.

2.1. Static Program Analysis for Security

This section describes tools which utilize static analysis techniques to find security vulnerabilities. They differ in the extent and precision of their programming language model. Early methods deal only with the tokenized source code while later techniques consider more aspects of the program's semantics. An analysis can leverage the *abstract syntax tree* (AST) to obtain the basic semantics of a program. Others adopt further technologies which utilize the *control flow graph* (CFG) and *call graph* to achieve even higher precision. The presented techniques include type checking, constraint solving, data flow analysis, and model checking.

2.1.1. Basic Lexical Analysis

One of the first static analyzers specifically addressing security was *ITS4*[1]. Its simple approach is based on lexical analysis. ITS4 basically tokenizes the source files and matches the resulting token stream with predefined patterns. *Flawfinder*[2] and *RATS*[3] are similar tools which syntactically search in the program code for a predefined set of vulnerable functions. While ITS4 and Flawfinder target C and C++, RATS is also able to analyze Perl, PHP, and Python code. All three tools are sound with respect to their rule base but report a high number of false positives which is a direct consequence of their simple approach.

A good example for a common vulnerability which can be addressed by lexical analysis is the use of potentially insecure C functions. These functions are susceptible to buffer overflows—e.g. `strcat()`—and it is recommended (Miller *et al.* [MdR99]) to replace them with safer alternatives—e.g. `strlcat()`. Lexical analysis—while extremely fast in evaluating a program—is prone to report many false positives. To follow up with the example a variable called `strcat` would also trigger a warning since lexical analysis can't distinguish between function names and variable names. Lexical analysis further ignores the flow of values which is crucial to detect vulnerabilities caused by user input. Many security defects will remain undetected as they require a semantic interpretation and by this a more sophisticated approach. All three tools are included in the comparison in Section 8.4.

[1] It's the Software Stupid! Security Scanner (ITS4) presented by Viega *et al.* in [VBKM00] in the year 2000.
[2] Flawfinder developed by David Wheeler in 2001 is available online [Whe11].
[3] Rough Auditing Tool for Security (RATS) published 2001 by Secure Software, Inc. is described in [Sec11].

2.1.2. Annotation-based Analysis

LCLint[4] is an annotation-based static analysis tool. It is designed to detect inconsistencies between LCL specifications and their C implementations. It can detect null pointer dereferences, dead storage, memory leaks, and other programming issues. The analysis is guided by programmer-provided source code annotations.

Splint[5] extends LCLint to the security domain. It is especially designed to discover buffer overflow vulnerabilities but is also extendable to cover other flaws like format string bugs. The tool requires the programmer to add annotations to their program in the form of special C comments. Splint extends the annotation language of LCLint with more expressions to specify function pre- and postconditions and other properties. Splint relies on these annotations when it calculates the minimum and maximum indices used to access buffers. It generates constraints from these annotations and adds them to the abstract syntax tree. The constraints are resolved and for each violation a corresponding warning is reported.

Splint is unsound and incomplete because it reports warnings for correct code and misses some vulnerabilities [LE01]. The biggest barrier to wide-spread adoption of an annotation-based analysis are probably the annotations themselves. The authors of Splint conclude in [EL02] that it might be optimistic to think that programmers are willing to add annotations to their programs: "The effort involved in annotating programs is significant and limits how widely these techniques will be used in the near future."

2.1.3. Constraint-based Analysis

BOON[6] performs an integer range analysis to identify buffer overflows caused by string manipulation function. Each string is modeled with two numbers: the amount of memory allocated and the number of bytes currently used. All functions of the C standard library which manipulate strings are modeled in a constraint language based on these ranges.

During the analysis the tool generates integer range constraints for each statement of a program. The corresponding constraint system is solved afterwards by finding a minimal bounding box solution that encloses all possible execution paths. Warnings are reported for all statements which might violate the constraints. The author admits its "many false alarms" [WFBA00, p. 4] which are caused by the flow- and context-insensitive nature of the approach (see Section 2.2.3 for details).

ARCHER[7] is a static analysis tool which checks for memory access errors. It operates on the control flow graph and uses an approximation of the call graph for an inter-procedural analysis. The tool creates constraints from conditions and checks if a memory access is unsafe with respect to these constraints. The constraints are solved by a linear constraint solver.

The analysis developed by Xie *et al.* uses statistical code analysis to infer the functions that it should analyze. The approach is context-sensitive and performs an alias analysis for buffers. The authors state that they do not consider string operations. Zitser argues in [ZLL04, p. 98] that ARCHER is further restricted as it ignores function pointers, uses heuristics to analyze loops, and considers only simple range constraints.

[4] David Evans *et al.* presented LCLint 1994 in [EGHT94].
[5] Secure Programming Lint (Splint) was introduced by David Larochelle and David Evans 2001 in [LE01].
[6] Buffer Overrun detectiON (BOON) was introduced in the year 2000 by Wagner *et al.* in [WFBA00].
[7] ARray CHeckER (ARCHER) was described by Xie *et al.* 2003 in [XCE03].

2.1.4. Type-based Analysis

cqual[8] is a framework that allows the programmer to extend the type system of C with user-defined type qualifiers. It employs a constraint-based type-inference engine to propagate the qualifiers. Afterwards type checking is used to reason about the program. One example given in [FFA99] is const inference which allows the user to find the maximum number of const annotations for the variables of an arbitrary C program.

Shankar *et al.* [STFW01] extended the framework to the domain of software security in general and to find format string vulnerabilities in particular. Apart from format string vulnerabilities, cqual is also able to check for year 2000 bugs and deadlocks.

The programmer has to annotate variables as *tainted* or *untainted*. These two type qualifiers indicate wether a variable is influenced by user input. The C source code and additional configuration files containing pre-defined annotations for system libraries are used to generate an annotated abstract syntax tree. cqual traverses the AST and propagates the annotations with type inference. A following type check reveals format string vulnerabilities as type inconsistencies: a tainted variable is used where an untainted variable is expected. The programmer has to annotate format string functions beforehand to accept only variables of type *untainted* as their format string specifier.

The analysis is inter-procedural and can handle pointers. However, the approach is flow-insensitive which means that a tainted variable is considered to be tainted at every statement of a function. This can lead to taint flowing backwards from a user input function to a vulnerable function which results in a false positive. cqual is sound modulo some restrictions with casts. I have selected it for the evaluation in Section 8.4.

2.1.5. Data Flow Analysis

Vulncheck[9] is an extension for the popular open source compiler gcc. It augments gcc with a data flow analysis which is used to track tainted data. Vulncheck focuses on security vulnerabilities including uncontrolled memory allocations, buffer overflows, invalid array accesses, format string flaws, and insecure C functions. It facilitates a standard implementation of Patterson's algorithm to perform a value range propagation. The resulting intervals for integer variables are used to determine if a possible buffer overflow vulnerability is exploitable. The analysis is flow-sensitive and respects the order of statements in the program. Vulncheck performs only an intra-procedural analysis and cannot find vulnerabilities which span several functions. This limitation stems from the integration in gcc which does not support inter-procedural client analyses. I selected Vulncheck for the tool comparison in Section 8.4.

Livshits *et al.* [LL05] proposed a static analysis approach based on a pointer analysis which finds vulnerabilities in Java web applications. They employ a tainted object propagation which is a special data flow analysis to find sink objects which can be derived from source objects. Their method can find SQL injection, cross-site scripting, HTTP response splitting, path traversal, and command injection attacks. They proposed a similar approach for C programs earlier [LL03] which is also based on an alias analysis and can find buffer overflows and format string bugs.

[8] Foster *et al.* presented cqual in 1999 [FFA99].

[9] Vulncheck was developed by Alexander Sotirov and presented in his master's thesis in 2005 [Sot05].

Jovanovic *et al.* [JKK06] developed *Pixy*, a static analysis tool which detects cross-site scripting vulnerabilities in PHP scripts. They employ a data-flow-based taint analysis. Pixy also performs an alias analysis to handle pointers appropriately.

Several *dynamic* data flow analyses have been proposed as well. A dynamic taint analysis executes a program and observes which computations are affected by user input. Flaws are detected at runtime when tainted data is used in vulnerable functions. Schwartz *et al.* describe the fundamentals of dynamic taint analysis in [SAB10]. An example for a dynamic taint analysis is the approach by Haldar *et al.* [HCF05] which targets command injections, parameter tampering, cookie poisoning, and cross-site scripting in Java web applications.

Chang *et al.* [CSL08] describe a similar dynamic taint analysis which targets format string attacks, command injections, and privilege escalation in C programs. The system is based on the *Broadway* compiler which integrates a data flow analysis into a program at compile time. This small library tracks the taint status of variables during runtime and detects vulnerabilities which are caused by untrusted data.

2.1.6. Model Checking

MOPS[10] uses model checking techniques to check for the violation of rules defined as temporal safety properties. The intended audience of the tool are developers implementing security-critical software and auditors conducting code reviews.

Users can define their own temporal safety properties though this might be infeasible for users unfamiliar with finite state automata. Temporal safety properties are used to reason about the order of security-relevant operations. MOPS has been used to detect file access race conditions, vulnerable chroot jails, and privilege management errors as reported in [CM04].

The program under evaluation is represented as a push-down automaton and the temporal safety properties are represented as finite state automata. Model checking is used to determine if a state can be reached that violates a security goal.

MOPS takes the control flow into account but neglects data flow. It therefore allows only to express properties that concern the order of operations. Due to this restriction the tool cannot reason about the influence of user data and is unable to find related vulnerabilities like format string problems. The approach is sound except for execution paths originating from function pointers, signal handlers, and jumps. The analysis is inter-procedural.

2.2. Comparison

In this section I will compare those of the before mentioned tools which target security vulnerabilities in C code. The comparison focuses on three key attributes of the approaches: precision, scope, and sensitivity. A tool's precision can be measured by its soundness and completeness. Figure 2.1 classifies the approaches accordingly. A tool's scope captures how it handles the interactions of multiple functions and pointer aliasing. The approaches are summarized with respect to their scope in Figure 2.2. A tool's sensitivity depends on how it models the data flow and function calls. Figure 2.3 depicts which tools are flow- and context-sensitivity.

[10] MOdelchecking Programs for Security properties (MOPS) published by Chen *et al.* 2002 in [CW02].

	incomplete	complete
sound	ITS4, Flawfinder, RATS, cqual, MOPS	*impossible*
unsound	BOON, Splint, ARCHER, Vulncheck	—

Figure 2.1.: Comparison of static analysis tools regarding soundness and completeness.

2.2.1. Precision

The precision of a static analyzer can be captured by the number of false positives it reports and the number of existing issues it fails to report. Previous evaluations have tested the detection rate of tools with respect to synthetic test cases [Kra05] and existing vulnerabilities in software products [ZLL04]. I will restrict the comparison in this section to soundness and completeness—the absence of false negatives respectively false positives. The comparison is summarized in Figure 2.1.

ITS4, Flawfinder, RATS The lexical analysis tools are sound as they report all vulnerabilities according to their rule set. ITS4, Flawfinder and RATS are incomplete because they report a high number of false positives.

Splint Larochelle *et al.* report in [LE01, p. 178] that their tool is unsound as well as incomplete. They instead focused on reducing the number of false reports and tried not to miss too many real vulnerabilities.

BOON Wagner *et al.* states in [WFBA00, p. 4] that BOON is unsound as a result of its imprecision. The tool ignores aliasing, function pointers, and unions [WFBA00, p. 7]. The analysis is incomplete as it reports false positives. However, the authors explain in [WFBA00, p. 4] that the main design goal of the tool is to lower the number of unsafe string operations which a programmer has to check manually. They claim that BOON reduces this number by an order of magnitude.

ARCHER ARCHER approximates the call graph because it does not track function pointers [ZLL04, p. 98] and it ignores string operations [Kra05, p. 71]. The approach is therefore unsound. Like the other tools it reports false positives and hence is incomplete as well.

cqual Shankar *et al.* describe in [STFW01, p. 211] that for soundness the user is required to annotate all potentially-vulnerable varargs functions. The approach is sound modulo the handling of specific uncommon casts. cqual is incomplete because it reports false positives.

Vulncheck The approach by Sotirov [Sot05] is unsound because it misses vulnerabilities which span several functions. Vulncheck is incomplete because it reports false positives.

MOPS The approach by Chen *et al.* is sound with the exception of function pointers, signal handlers, and non-local jumps [CW02, p. 243] which introduce new execution paths it does not consider. The tool is incomplete as it reports false positives.

inter-procedural	BOON, MOPS	cqual, ARCHER
intra-procedural	ITS4, Flawfinder, RATS, Splint, Vulncheck	—
	no aliasing	aliasing

Figure 2.2.: Static analysis tools classified by scope.

2.2.2. Scope

The scope of an analysis can be measured in two dimensions. The first dimension captures if an analysis considers problems which involve several functions. An *intra-procedural* analysis focuses on one function independent of the others. An *inter-procedural* analysis takes the interaction of different functions into account.

The second dimension measures how the approaches model pointers. Some tools perform a separate pointer alias analysis to achieve higher precision while others ignore pointers completely. Figure 2.2 categorizes the approaches regarding these two dimensions.

ITS4, Flawfinder, RATS The three tools are based on pattern matching and neither model pointer aliasing nor the interaction between functions.

Splint Splint does not facilitate the call graph and is primarily an intra-procedural approach. It is able to perform a limited inter-procedural analysis locally when the programmer provides additional annotations [LE01, p. 183]. Splint seems not to perform a dedicated alias analysis.

BOON The approach models function calls and thus is inter-procedural [ZLL04, p. 100]. Wagner *et al.* provide an example of a buffer overflow found by BOON which is caused by the interaction of several functions [WFBA00, p. 10]. BOON cannot detect buffer overflows caused by pointer arithmetics because it does not consider pointer aliasing [WFBA00, p. 7].

ARCHER The tool uses a bottom-up inter-procedural analysis to propagate information across procedure boundaries [XCE03, p. 334]. ARCHER uses points-to information which is derived using a simple per-path alias analysis algorithm [XCE03, p. 332].

cqual The type-checking approach by Shankar *et al.* is inherently inter-procedural since it matches the type of a formal function parameter with the type of an actual argument. They present a vulnerability detected by cqual which spans several functions [STFW01, p. 202]. cqual also considers pointer aliases as their type has to match exactly [STFW01, p. 205].

Vulncheck The data flow analysis used by Vulncheck is intra-procedural and analyzes functions in isolation [Sot05]. The approach does not perform an alias analysis.

MOPS Chen *et al.* claim in [CW02, p. 244] that their approach is fully inter-procedural. MOPS checks if any execution path through a program violates a security property, including execution paths that span several functions. It ignores most of the data flow [CW02, p. 239] and hence does not need to model pointer aliasing.

	context-insensitive	context-sensitive
flow-sensitive	MOPS, Vulncheck	Splint, ARCHER
flow-insensitive	ITS4, Flawfinder, RATS, BOON, cqual	—

Figure 2.3.: Static analysis tools ranked with respect to flow- and context-sensitivity.

2.2.3. Sensitivity

Sensitivity describes how an analysis models functions calls and the data flow of a program. An analysis is called *context-insensitive* if it does not distinguish between different calls to the same a function. Instead the call sites are merged and the function is analyzed once with this combined information. A *context-sensitive* analysis avoids the cross-over of information from one call site to another and maintains the proper calling context. An analysis is *flow-sensitive* if it considers the order of statements in the program. If the order is unimportant for the analysis it is called *flow-insensitive*. Figure 2.3 compares the different tools with respect to their sensitivity.

ITS4, Flawfinder, RATS A lexical analysis ignores different call-sites and the flow of data. ITS4, Flawfinder and RATS are therefore context- and flow-insensitive.

Splint Splint records the locations of expressions used as actual arguments in its constraints [LE01, p. 181]. This is an indicator for a context-sensitive analysis. Splint is flow-sensitive as it employs a standard compiler data flow analysis [LE01, p. 183].

BOON The tool merges the information from all call sites of the same function and hence is context-insensitive. Wagner *et al.* pursue a flow-insensitive approach. [WFBA00, p. 7]

ARCHER The constraint-based analysis by Xie *et al.* summarizes functions in constraints and these are evaluated at all call sites with the correct context [XCE03, p. 332]. The approach is therefore context-sensitive. The approach is based on a flow-sensitive data flow analysis [XCE03, p. 330].

cqual The type-based approach by Shankar *et al.* is context-insensitive as well as flow-insensitive [CW02, p. 243]. A tainted variable is considered to be tainted in the whole program even before the statement that tainted it.

Vulncheck The taint analysis by Sotirov does not consider call sites and is hence context-insensitive. Vulncheck performs a data flow analysis to determine statements where variables are tainted and as a result is flow-sensitive.

MOPS The approach by Chen *et al.* is context-insensitive as it is only concerned with the order of function calls and not their calling context. MOPS is flow-sensitive because the order of statements is important for the analysis [CW02, p. 235].

3. Security Vulnerabilities

This chapter presents several taxonomies for security vulnerabilities. They were developed to classify security flaws into categories based on specific characteristics. Different taxonomies focus on different properties of the vulnerabilities. My approach targets flaws which are caused by user input. The second part of this chapter describes weaknesses of this kind.

3.1. Taxonomies

A classification of security flaws leads to a better understanding and can also serve as a common vocabulary for vulnerabilities. This facilitates the exchange of thoughts about security problems and avoids misunderstandings. Several classifications have been proposed in the security domain since the 1970s. I will first introduce a classification which is of historical significance and afterwards focus on those which influenced the Common Weakness Enumeration (CWE). The U.S. National Institute of Standards and Technology (NIST) uses the CWE taxonomy to evaluate software security tools. I will use this benchmark for the evaluation of my approach in Chapter 8.

3.1.1. Taxonomy of Integrity Flaws (1976)

One of the earliest taxonomies for security defects originates from the Research Into Secure Operating Systems (RISOS) project [ACD+76] conducted at the Lawrence Livermore Laboratories. The project's goal was to advise computer and systems managers to understand security issues and to help them estimate the effort needed to improve security features. Abbott *et al.* mention the need to protect data concerning the nation's defense as one reason for the project. The "Privacy Act of 1974" served as an additional motivation since it demands that data of individuals collected by government agencies should be sufficiently protected.

They proposed the *Taxonomy of Integrity Flaws* which included even the physical protection of computer systems. The following list summarizes their classification of operating systems security flaws. This part of their taxonomy consists of the following seven categories:

Incomplete Parameter Validation Addresses the insufficient validation of user input. An attacker could exploit this to access restricted data or even crash the system.

Inconsistent Parameter Validation Similar but concentrates on cases where different parts of the system make different assumptions about what constitutes valid input. Even when all parts validate input correctly with respect to their assumptions, security flaws can still occur in the interaction of those parts when their assumptions are inconsistent.

Implicit Sharing of Privileges/Confidential Data Flaws are part of this category whenever information is not correctly isolated between users or users and the operating system.

Asynchronous Validation/Inadequate Serialization Flaws that are caused by the existence of a timeframe between the validation and the actual use of data. An attacker could change the data after the validation but before its use.

Inadequate Identification/Authentication/Authorization Vulnerabilities where users counterfeit their identity and circumvent password checks.

Violable Prohibition/Limit Security issues concerning the violation of upper or lower limits. The operating system should enforce limits—failing this the system might crash or data could be lost. This category includes buffer overflows.

Exploitable Logic Error For example incorrect error handling routines where illegal actions are performed before an error condition is signaled.

Their final report provides examples for each flaw category. They also classified 17 actual flaws into the seven categories. Those vulnerabilities affect three operating systems: IBM OS/MVT, UNIVAC 1100, and TENEX.

The first two of the seven categories explicitly deal with flaws caused by insufficiently validated user input. Already in the 1970s this was seen as one of the primary reason for security flaws.

3.1.2. A Taxonomy of Computer Program Security Flaws (1994)

The taxonomy of Landwehr *et al.* [LBMC94] focuses on flaws that are detected in released software. They believe that an organization of security problems can help others to focus their efforts to find security problems or even prevent the introduction of flaws into software.

The *Taxonomy of Computer Program Security Flaws* is centered around three questions:

1. How did the flaw enter the system?
2. When did it enter the system?
3. Where in the system is it manifest?

These three questions correspond to three subsections of the classification. They are the three perspectives from which each vulnerability can be observed: genesis (how), time of introduction (when), and location (where).

Those three dimensions are organized in a structure of subcategories of varying depth. Flaws by genesis are broken down into intentional, and inadvertent, where the intentional category is further split up into malicious and non-malicious. Defects by time are broken down into development, maintenance, and operation, while the development class is itself partitioned into requirements, source code, and object code. Vulnerabilities by location are broken down into software and hardware, where software is redivided into operating system, support, and application.

Landwehr *et al.* collected 50 actual flaws from the literature and classified them into their taxonomy. Their work was later continued by Viega in his *CLASP Application Security Process* [Vie05]. He added several perspectives in addition to genesis, time, and location including consequence, platform, required resources, severity, likelihood of exploit, and avoidance.

3.1.3. The 19 Deadly Sins of Software Security (2005)

The 19 Deadly Sins of Software Security is a list of software security problems which was published as a book by M. Howard *et al.* in 2005 [HLV05]. It comprises 19 common security defects which are illustrated with code examples.

1. Buffer overruns
2. Format string problems
3. Integer overflows
4. SQL injections
5. Command injections
6. Failure to handle errors
7. Cross-site scripting
8. Failure to protect network traffic
9. Use of magic URLs and hidden forms
10. Improper use of SSL
11. Use of weak password-based systems
12. Failure to store and protect data securely
13. Information leakage
14. Trusting network address resolution
15. Improper file access
16. Race conditions
17. Unauthenticated key exchange
18. Failure to use cryptographically strong random numbers
19. Poor usability

The list ranges from code-level problems like buffer overflows to high-level issues like weak passwords. The book is based on a list developed by the U.S. Department of Homeland Security. The 19 entries are claimed to account for 95 % of all security issues [HLV05]. It does not use subcategories or different dimensions to classify the security flaws further. A recent edition adds five new issues and is therefore named "24 Deadly Sins of Software Security" [HLV09].

3.1.4. Seven Pernicious Kingdoms (2005)

The taxonomy presented by Tsipenyuk *et al.* [TCM05] is tailored for "automatic identification using static source code analysis techniques." They view such tools as an effective teaching mechanism for developers. Their taxonomy focuses on code-level security problems which excludes organizational issues.

Seven Pernicious Kingdoms is based on a simple hierarchy consisting of categories—called *kingdoms*—and the specific types of coding errors—named *phyla*. Phyla are part of the same kingdom if they share similar characteristics.

Their main goal was to design a simple and intuitive taxonomy with only a small number of categories. The classification is designed to be precise and consistent so that a vulnerability is mapped to exactly one category. Other design goals include the adaptability and extensibility of the classification as well as the suitability for source code analysis tools.

The seven kingdoms of the taxonomy ordered by importance for software security are:

1. **Input Validation and Representation** User input is trusted without proper validation. This kingdom contains buffer overflows, command injections, and SQL injections.
2. **API Abuse** Violations of the implicit or explicit contract between caller and callee. This covers the chroot-jail problem and unchecked return values.
3. **Security Features** Misuse of security features. This kingdom is concerned with insecure randomness and inadequate password management.
4. **Time and State** Problems related to threads, processes, and timing. This includes deadlocks and race conditions.
5. **Errors** Insufficient error handling and errors which reveal sensitive information.
6. **Code Quality** Poor code quality is claimed to lead to security problems. This kingdom comprises the double free problematic, memory leaks, and null dereferences.
7. **Encapsulation** Weak boundaries between trusted and untrusted data.

*. **Environment** This section focuses on security relevant issues outside the source code level and is therefore separated from the other kingdoms.

Tsipenyuk *et al.* see their taxonomy as an alternative to highly specific collections of security problems like the Common Vulnerabilities and Exposures (CVE) [Mit11a] or taxonomies of attack patterns which they believe are difficult to facilitate for static analysis.

3.1.5. Common Weakness Enumeration Specification (2005)

The *Common Weakness Enumeration Specification* (CWE) [Mit11b] is a dictionary of software weakness types which is maintained by the Mitre Corporation. It is designed as a standard nomenclature for software security weaknesses and to help assess the coverage of different software security tools.

Mitre's first attempt at a software security taxonomy was the PLOVER list which was in turn based on the list of Common Vulnerabilities and Exposures (CVE) [Mit11a]. The U.S. National Institute of Technology (NIST) conducted the Software Assurance Metrics and Tool Evaluation (SAMATE) project to extend PLOVER. The result of this effort is the Common Weakness Enumeration Specification. It incorporates structural elements and examples of the before mentioned taxonomies CLASP (which is turn based on the Taxonomy of Computer Program Security Flaws), The 19 Deadly Sins of Software Security and Seven Pernicious Kingdoms.

Each CWE entry represents a single cause for one class of software vulnerabilities. They are organized in a hierarchical structure which allows several levels of abstraction (see Figure 3.1). Entries in higher levels (e.g. Input Validation) are categories for specific issues below them (e.g. Command Injection). The major benefit Mitre sees in this specification are the standardized identifiers and descriptions for the listed weakness types. Mitre claims that CWE identifiers have been adopted as a common language to refer to security defects.

Every year the SANS Institute and the Mitre corporation rank the weaknesses from the CWE list to obtain the CWE/SANS Top 25 Most Dangerous Software Errors which is available online. The list is designed as "a tool for education and awareness" [SM11].

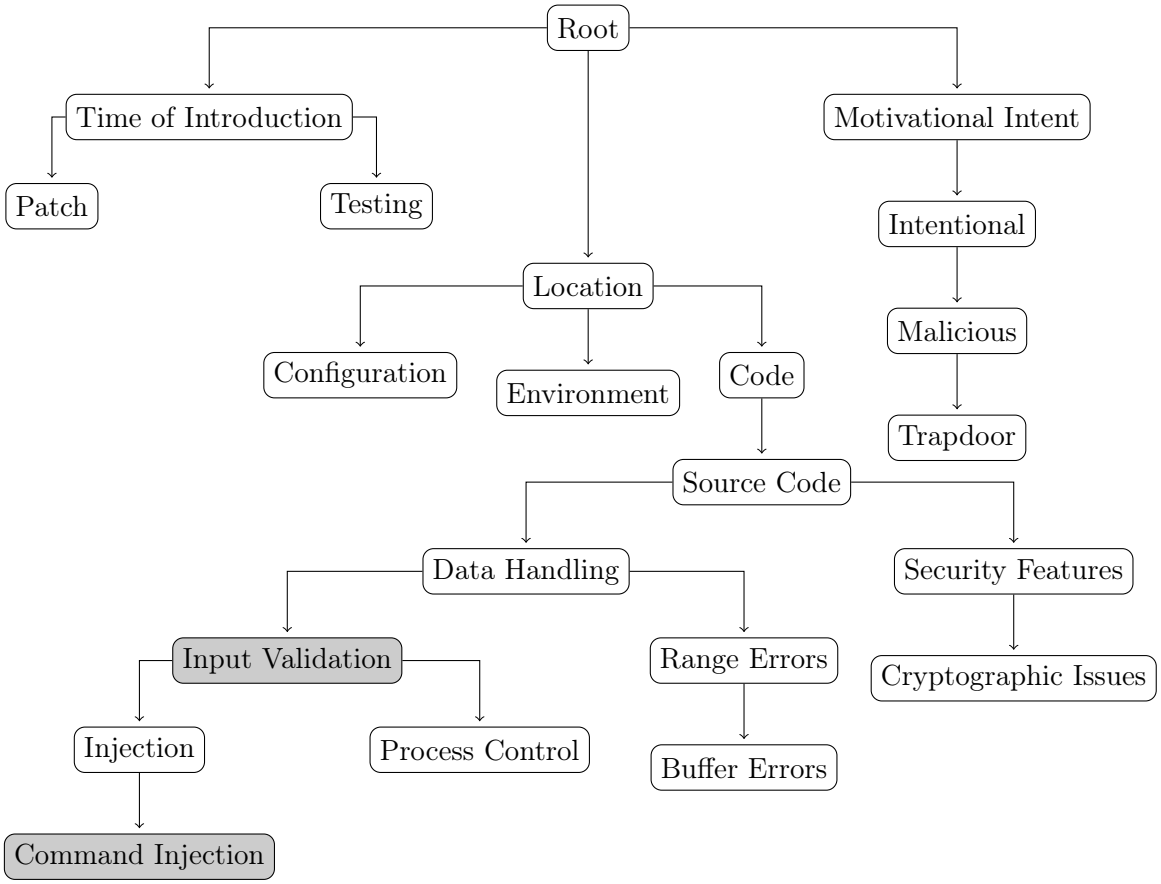


Figure 3.1.: Fragment of the Common Weakness Enumeration inspired by a figure in [Nat11a].

3.1.6. Software Assurance Metrics and Tool Evaluation (2005)

The U.S. National Institute of Standards and Technology (NIST) started the *Software Assurance Metrics and Tool Evaluation* (SAMATE) project in 2005. The SAMATE project aims "to help develop standard evaluation measures" [Bla05]. In 2007 the NIST decided "to work on static source code security analyzers" [Bla07]. They determined that a standard developed by NIST would assure developers that static analyzers are useful.

In the next step they defined what a static analyzer should do. But to assess the tools and evaluate which tools find what flaws a taxonomy of security weaknesses was needed. This effort led to the creation of Mitre's Common Weakness Enumeration (CWE) [Mit11b]. The enumeration of weaknesses alone is not sufficient to test static analyzers. The Standard Reference Dataset (SRD) [Nat11b] project consequently developed test cases for known security flaws including synthetic test cases as well as code from actual software products. This dataset is available online and provides a repository of flawed software for the evaluation and development of software assurance tools.

The forth installment of the Static Analysis Tool Exposition (SATE) is currently taking place. Goanna participates equipped with a component which facilitates the approach presented in this work. Some preliminary results are presented in Chapter 8.

3.2. Targeted Weaknesses

My approach targets several security weaknesses which all share the characteristic to pass user input to vulnerable functions. They differ in the set of vulnerable functions as well as the side conditions which determine if a vulnerability is exploitable. This section describes these weaknesses with code examples, common attacks, and a collection of related real world vulnerabilities. Appendix B.2 lists the functions that are considered for each category.

3.2.1. CWE 22: Path Traversal

A program uses external input to construct a path to a file without validating that it does not contain malicious characters. Attackers can use special character sequences to access files outside the intended directory. They might be able to arbitrarily traverse the file system hierarchy and view, edit, overwrite, or delete files they are not supposed to access.

This CWE entry is split up into *CWE 23 Relative Path Traversal* and *CWE 36 Absolute Path Traversal*. In the first case an attacker uses the character sequence `../` to traverse to the parent directory of the current folder. In the second case the program expects a file name relative to the current folder but the attacker provides an absolute path like `/etc/passwd`. This allows the attacker to access specific files anywhere on the system.

Code Example Figure 3.2 depicts the filesystem hierarchy for the code example in Listing 3.1. The program initializes the `path` variable with the directory `/tmp/sandbox/` in line 5. The following line appends the first command line argument `argv[1]` to this path. Then the file which is identified by the newly constructed path is opened in line 7 and subsequently printed out character by character in line 10. Notably, the resulting path is never checked for path-traversal directives like `../`.

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main(int argc, char *argv[]) {
5    char path[100] = "/tmp/sandbox/";
6    strncat(path, argv[1], 50);
7    FILE *file = fopen(path, "r");
8    char c;
9    while((c = fgetc(file)) != EOF) {
10     printf("%c", c);
11   }
12   fclose (file);
13   return 0;
14 }
```

Listing 3.1: Example code with a path traversal vulnerability.

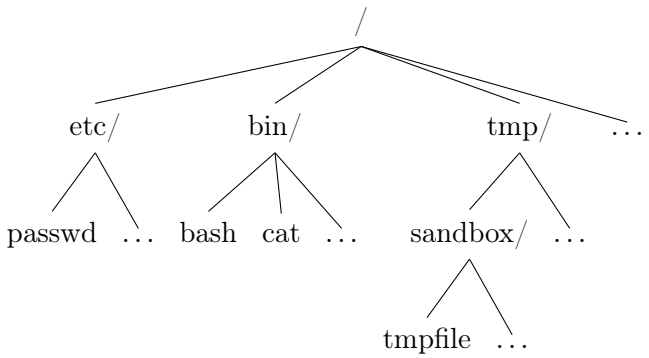


Figure 3.2.: Relevant part of the filesystem hierarchy for Listing 3.1.

Attack In this example program the user should only be able to display the contents of files located inside the `/tmp/sandbox` directory. However, an attacker could provide the input string `../../etc/passwd`, by this traverse the file system hierarchy and display the operating system's passwords file instead.

The `passwd` file allows the attacker to display all user accounts of the system. In older versions of Unix operating systems this file also contains password hashes. A look-up in a *rainbow table*[1] could reveal the actual passwords to the attacker. A missing validation of a filename could allow a malicious user to compromise the whole system.

Real World Examples Software products which were subject to this weakness include several FTP servers (CVE-2009-4194, CVE-2009-4053, CVE-2009-0244), content management systems (CVE-2009-4581, CVE-2008-5748), and chat clients (CVE-2010-0013).

3.2.2. CWE 78: OS Command Injection

The operating system command injection weakness was ranked second in the 2011 edition of the "Top 25 Most Dangerous Software Errors" [SM11].

Attackers can exploit a command injection weakness to alter a program's course of execution. There are very different manifestations of command injections ranging from SQL injections over dynamic evaluation, cross-site scripting, and file inclusion to operating system injections. In the case of C and C++ the latter is the most common variant.

A shell injection—another common term for an operating system command injection—often exists when a program passes user input to another executable. This weakness is caused by the use of insufficiently validated user input to construct a system command.

Attackers can exploit meta-characters of the shell to carefully craft an input string which allows them to execute arbitrary commands with the privileges of the affected program. An overview of several common shell meta-characters, which could be used to such an effect, is depicted in Figure 3.3.

[1] A *rainbow table* is used to store all possible strings which correspond to a cryptographic hash value. The table contains a complete mapping from hash values to plaintext passwords as long as the password doesn't exceed a length limit and only uses certain characters.

Input String	Description
`> /some/file`	Redirects the output of the preceding command to the specified file and by this overwrites the file if the program has sufficient privileges.
`< /some/file`	Sets the declared file as the input for the preceding command.
`; command`	Executes the attackers command after the preceding command.
`&& command`	Executes the attackers command if the previous command returned with a 0 status code indicating success.
`|| command`	Executes the attackers command if the previous command returned with a non-zero status code indicating failure.
`| command`	Pipes the output of the preceding command to the attackers command.
`` `command` ``	Executes the attackers command first and then passes the output as arguments to the previous command.

Figure 3.3.: Shell meta-characters used to exploit command injection vulnerabilities.

Code Example The following example program is a very simple wrapper for the Unix `cat` executable which takes a filename and returns its contents. The program is very simple but nevertheless serves as an example for the command injection vulnerability.

In line 5 the `cmd` string is initialized with the absolute path to `cat`. The user input obtained from a call to `scanf()` in line 7 is appended to the command string in line 8. Finally, the full command string is executed with the `system()` function in line 9.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[]) {
5   char cmd[100] = "/bin/cat ";
6   char filename[50] = "";
7   scanf("%50s", filename);
8   strcat(cmd, filename);
9   system(cmd);
10 }
```

Listing 3.2: Example code with a command injection vulnerability.

Attack If an attacker passes the argument `file.txt; rm -r -f /` to the program it will not only print the contents of the file but also remove all files from the system provided it has sufficient privileges. The attacker concatenated the command `rm -r -f /` with the shell meta-character `;` to the filename `file.txt`. This executes the attackers command after the preceding command `cat file.txt`. The same flaw can also be used to overwrite important system files by providing input of the form `file.txt > /etc/passwd`.

Real World Examples Known examples for command injection vulnerabilities include the Apache web server (CVE-2002-0061), the Kerberos FTP client (CVE-2003-0041), and the dynamic web language PHP (CVE-2001-1246). These in-the-wild examples are naturally more complicated than the code example. However, they share the same characteristic data flow: Unvalidated user input is used to construct a shell command which is subsequently executed with a function like `system()` or `execl()`.

3.2.3. CWE 114: Process Control

A process control weakness includes a call to a function like `LoadLibrary()` which loads a dynamic library and executes the contained code. This specific function is part of the Windows operating system but similar functions exist for other platforms. If the attacker can influence which library is loaded they can inject malicious code which is executed with the privileges of the affected program.

One variant of the process control weakness was present in over 50 common Windows programs [US-11]. The problem was the use of a relative path to identify the DLL file in a call to the `LoadLibrary()` function. In this case the current working directory is searched first for the library file and the system directories are considered afterwards. If the attacker controls the working directory they can simply put a file in it which is named similar to a system library. Consequently, the attackers file would be loaded due to the relative path.

Consider the following situation: An attacker sends a link to a remote network share which contains several media files. Additionally, the program which is associated to their filetype is vulnerable to a process control weakness. The directory contains a malicious library placed there by the attacker. At the moment the user opens one of the media files the malicious library will be loaded and executed.

Since my approach is designed for weaknesses caused by user input it cannot check for this manifestation of a process control weakness. Instead it concentrates on cases where the attacker is able to manipulate the filename of the library because of missing input validation.

Code Example The following example program writes the content of the environment variable `DLL_PATH` into the variable `dllname` in line 6. In the next line the system loads and executes the dynamic library which is identified by this path.

```
1 #include <stdlib.h>
2 #include <winbase.h>
3
4 int main(int argc, char *argv[]) {
5   char *dllname;
6   dllname = getenv("DLL_PATH");
7   LoadLibrary(dllname);
8   return 0;
9 }
```

Listing 3.3: Example code with a process control vulnerability.

Attack Attackers could change the environment variable before the program executes and by this load an arbitrary library file. This allows them to execute malicious code with the privileges of the exploited program. Consequently, environment variables are considered external input by the taint analysis.

3.2.4. CWE 129: Improper Validation of Array Index

CWE 129 describes a weakness where user input is used to access an array. The array index derived from the user input was not checked to be in the bounds of the array. The following array access could therefore cause problems and even constitute a security vulnerability.

Code Example The following program uses network input to access an array. In line 6 data is received from a socket. It is written into the character array `buffer`. In line 8 this string is interpreted as an integer with the `sscanf()` function. In the next line the integer is used as an index to access the `options` array.

```
1 #include <stdio.h>
2
3 char *get_option(int socket, char *options[]) {
4   char buffer[1024];
5   int index, success;
6   success = recv(socket, buffer, sizeof(buffer) - 1, 0);
7   if (!success) return;
8   sscanf(buffer, "%d", &index);
9   return options[index];
10 }
```

Listing 3.4: Example code with an array index vulnerability.

Attack The integer received over the network could be smaller than zero or larger than the size of the options array. In both cases the array would be accessed outside its bounds. An attacker could trigger this on purpose to compromise an affected application.

Real World Examples Examples where an unvalidated array index led to other security problems include two mail clients (CVE-2001-1009, CVE-2003-0721), the MIT Kerberos security protocol (CVE-2004-1189), and the Linux kernel (CVE-2005-2456).

3.2.5. CWE 134: Uncontrolled Format String

A very subtle security problem is the format string vulnerability. It is caused by design decisions for the C standard library which favors efficiency over security. A typical usage of a format string would be `printf("%s", message)`. The argument `"%s"` is a format string

Token	Description	Example
`%d`	Signed decimal integer.	`printf("%d", 42);`
`%f`	Decimal floating point.	`printf("%.2f", 2.718281828);`
`%s`	String of characters.	`printf("%s", "Daniel Waterhouse");`
`%x`	Unsigned hexadecimal integer.	`printf("%x", 0x2A);`
`%n`	Stores current character count.	`printf("%n", &count);`

Figure 3.4.: Format tokens used by format string functions.

that specifies the number and types of the following arguments—in this case one argument is expected which should point to the address of a character array.

The format string is not checked to match with the actual count and type of the arguments. The following call can therefore be dangerous: `printf(input)`. If the string `input` contains format tokens such as `%x` then `printf()` will still try to read the next entry from the stack and print it as an hexadecimal value. Different format tokens correspond to different types of variables as depicted in Table 3.4.

What might be surprising is that an attacker with control over the contents of the format string can use this to read and write arbitrary memory locations. Crucial for this is the special format token `%n`. It interprets the top of the stack as an address to an integer variable and stores the number of characters printed so far into this variable. Intricate techniques have been developed to exploit format string functions to compromise the security of a program. I refer to [New00] for a detailed description.

Code Example The following program authenticates the user with a password. The user is prompted for a name and password in line 6. The user's input and the correct password are passed to another function in line 7. Line 10 prints the raw user name when the login failed.

```
1 #include <stdio.h>
2
3 void login() {
4   int level; char pwd[50];
5   char *password = "SECRET";
6   scanf("%s %s", user, pwd);
7   check(pwd, password, &level);
8   if (level < 0) {
9     printf("Login failed:");
10    printf(user);
11  }
12  do_powerful_stuff(level);
13 }
```

Listing 3.5: Example code with a format string vulnerability.

Attack With the use of format tokens inside the user name an attacker is able to read the secret password and even circumvent the password check by writing to the `level` variable. The attacker could provide the user name `%x.%s` which would print the password. The input `%x.%32d.%n` would assign the value 42 to the variable `level`.

It is easy to prevent format string attacks by using constant format strings instead of user-controlled variables. Format string bugs are an important security problem due to the variety of different format string functions and the ignorance of many developers.

Real World Examples Documented format string vulnerabilities include virtual machines (CVE-2010-1139), mail servers (CVE-2011-2475), web servers (CVE-2010-2271, CVE-2009-4769), print servers (CVE-2010-0393), file servers (CVE-2010-0388, CVE-2009-1886), ftp servers (CVE-2009-4775), and database systems (CVE-2009-2446, CVE-2008-5440).

3.2.6. CWE 427: Uncontrolled Search Path Element

The operating system uses the search path to determine the correct executable for a command. If attackers are able to modify the search path and add custom paths to it they could force a program to execute malicious code.

Code Example This simplified example allows the user to directly set search path elements. In line 5 user input is stored in the variable `input`. It is used in line 6 to set environment variables. In the next line the `cat` command is executed to display the contents of a file.

```
1 #include <stdlib.h>
2
3 int main(int argc, char *argv[]) {
4   char *input;
5   scanf("%s", input);
6   dllname = putenv(input);
7   system("cat text.txt")
8   return 0;
9 }
```

Listing 3.6: Example code with a search path vulnerability.

Attack An attacker could add a directory to the search path which contains an alternative binary for the `cat` command. This malicious program would be executed when the program calls `cat` in line 7.

Real World Examples Programs vulnerable to a modified search path include text editors (CVE-2010-3402, CVE-2001-0289), data encryption tools (CVE-2010-3397), web browsers (CVE-2010-3131), database systems (CVE-2002-1576, CVE-2001-0943), operating systems (CVE-1999-1318, CVE-2002-2040), and debugging tools (CVE-2005-1705).

3.2.7. CWE 789: Uncontrolled Memory Allocation

A memory allocation is uncontrolled when unvalidated user input is used to determine the amount of memory to allocate. An attacker could modify the input to allocate enormous chunks of memory which would exhaust the resources of the system. A different but equally dangerous attack would be to alter the input with the effect that insufficient memory is allocated. This could lead to a successive buffer overflow.

Other weaknesses of the program can improve the attacker's chance to exploit an uncontrolled memory allocation. For example if the allocation size is checked to be smaller than a given boundary the attacker could still provide negative input which would be implicitly cast to a large positive number (this corresponds to the weaknesses *CWE 194: Unexpected Sign Extension* and *CWE 195: Signed to Unsigned*).

Another possibility is input which causes an integer overflow. The allocated size would then be too small which would in turn cause a buffer overflow (this is an instance of the weakness described in *CWE 680: Integer Overflow to Buffer Overflow*).

Code Example The example program receives input over the network in line 6 and interprets it as an integer in line 7. This number is used as the allocation size in line 9. The newly allocated memory is the destination buffer for a string copy operation in line 10.

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[]) {
5   int num;
6   recv(socket, buffer, sizeof(buffer) - 1, 0);
7   sscanf(buffer, "%d", &num);
8   if (num > 100) return -1;
9   char *string = malloc(num);
10  strcpy(string, other_string);
11 }
```

Listing 3.7: Example code with a memory allocation vulnerability.

Attack The program checks that the network input is smaller than 100 to avoid resource exhaustion. However, the string received over the network could be a negative number which would be implicitly cast to a big unsigned number in the call to `malloc()`. This would still lead to an improper use of system resources. `num` could also be smaller than the length of the character sequence `other_string`. This would cause a buffer overflow in line 10.

Real World Examples Uncontrolled memory allocations have caused security problems in database systems (CVE-2008-1708), storage servers (CVE-2008-0977), and instant messaging clients (CVE-2004-2589).

4. Background

My approach is based on two techniques from the field of static analysis: data flow analysis and model checking. I will first introduce *data flow analysis* which I use to track potentially malicious user input. Next, I will give a brief summary of *syntactic model checking* which refines the results of the data flow analysis by generating counter-example paths.

4.1. Data Flow Analysis

Data flow analysis was introduced by Gary A. Kildall in [Kil73]. Today it is a standard technique used for instance in compilers to optimize programs. The general idea is to compute for each elementary program statement a set of information which may reach it. The information could be variable names, arithmetic expressions, or values of variables. This depends on the objective of the specific analysis.

This section explains the principles of data flow analysis with two examples: the *Reaching Definitions Analysis* and the *Very Busy Expressions Analysis*. They demonstrate the range of possible data flow analyses. Afterwards I will describe the formal generalization of the different types of data flow analyses into lattice-based Monotone Frameworks. During the course of this chapter I will explain the differences between forward and backward analyses as well as may and must analyses. The presentation of this section is based on the the second edition of the book *"Principles of Program Analysis"* by Nielson *et al.* [NNH05].

4.1.1. Reaching Definitions Analysis

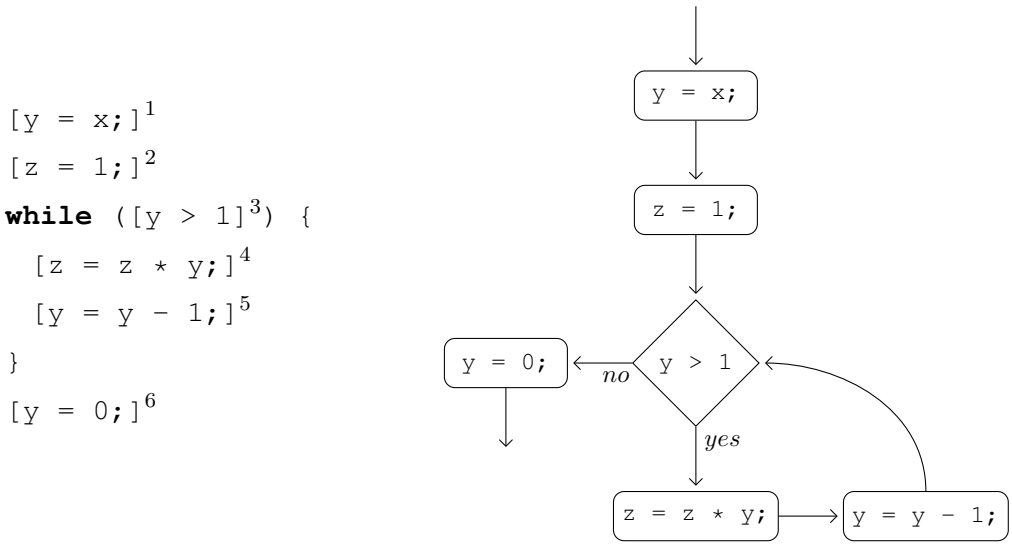
Figure 4.1 depicts the factorial program which computes the factorial of `x` and stores the result in `z`. Each basic statement is uniquely labeled so I can refer to it easily. This program will serve as an example for the analysis. Next to the program is its *control flow graph* (CFG) where the statements are represented as nodes and the transitions as edges.

The *Reaching Definitions Analysis* computes the set of variable assignments which are available at a program point:

> The assignment k of the form `[x = a;]`^{k} *may* be available at the entry of statement l if there is a path of execution where x was last assigned a value at k when statement l is reached.

This is useful to determine the connection between statements that define the value of a variable and statements that use this value of the variable.

In this example the assignment `[z = 1;]`^{2} reaches the statement `[z = z * y;]`^{4}—for brevity I will shorten this to "(z, 2) reaches the entry of 4". (z, 2) does not reach the the entry of 5 since the assignment at 4 redefines `z`. Figure 4.2 includes the reaching definition information for all statements of the program.



(a) Factorial program (b) Control flow graph

Figure 4.1.: Factorial program with corresponding control flow graph taken from [NNH05].

Equation System An analysis like reaching definitions can be formally specified by a number of equations which are specific to the analyzed program. These equations belong to two classes: An equation of the first kind defines what a single statement adds to and removes from the incoming set of information to create the outgoing set of information. It models how the statement changes the available information when executed. An equation of the second kind describes how the incoming information is obtained from the predecessors in the CFG. I denote the incoming set of information for statement k as IN_k and the outgoing set as OUT_k. For the factorial program from Figure 4.1a the equations of the first class are:

$$\mathrm{OUT}_1 = \mathrm{IN}_1 \setminus \{(y,l) \mid l \in \mathrm{labels}\} \cup \{(y,1)\}$$
$$\mathrm{OUT}_2 = \mathrm{IN}_2 \setminus \{(z,l) \mid l \in \mathrm{labels}\} \cup \{(z,2)\}$$
$$\mathrm{OUT}_3 = \mathrm{IN}_3$$
$$\mathrm{OUT}_4 = \mathrm{IN}_3 \setminus \{(z,l) \mid l \in \mathrm{labels}\} \cup \{(z,4)\}$$
$$\mathrm{OUT}_5 = \mathrm{IN}_4 \setminus \{(y,l) \mid l \in \mathrm{labels}\} \cup \{(y,5)\}$$
$$\mathrm{OUT}_6 = \mathrm{IN}_5 \setminus \{(y,l) \mid l \in \mathrm{labels}\} \cup \{(y,6)\}$$

There is clearly a difference between the equation for statement 3 and all the others. This is because statement 3 is a condition which does not define a variable. Therefore the set of outgoing information is the same as the incoming. Only assignments alter the reaching definition information.

In case of an assignment the outgoing set contains everything from the incoming set except the previous definitions of the variable which is being assigned. These definitions can't reach a following program point on a path through the current statement because the variable is redefined. In addition to removing old definitions of the assigned variable, a new definition has to be added to the set of outgoing information.

k	IN_k	OUT_k
1	(x, ?), (y, ?), (z, ?)	(x, ?), (y, 1), (z, ?)
2	(x, ?), (y, 1), (z, ?)	(x, ?), (y, 1), (z, 2)
3	(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)	(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)
4	(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)	(x, ?), (y, 1), (y, 5), (z, 4)
5	(x, ?), (y, 1), (y, 5), (z, 4)	(x, ?), (y, 5), (z, 4)
6	(x, ?), (y, 1), (y, 5), (z, 2), (z, 4)	(x, ?), (y, 6), (z, 2), (z, 4)

Figure 4.2.: Solution of the Reaching Definitions Analysis for the factorial program.

The second class of equations describes the set of incoming information for each statement. This is the combination of the sets of outgoing information from all its possible predecessors in the CFG. The general form of this equation for statement k with predecessors $\text{pred}(k)$ is:

$$\text{IN}_k = \bigcup_{i \in \text{pred}(k)} \text{OUT}_i \tag{4.1}$$

The information from the predecessors is combined with the set union operator. Data flow analyses differ in the operator they use to join the incoming information at a program point as we will see when I present the Very Busy Expressions Analysis. For the factorial program the equations describing the incoming information are:

$$\begin{aligned}
\text{IN}_2 &= \text{OUT}_1 \\
\text{IN}_3 &= \text{OUT}_2 \cup \text{OUT}_5 \\
\text{IN}_4 &= \text{OUT}_3 \\
\text{IN}_5 &= \text{OUT}_4 \\
\text{IN}_6 &= \text{OUT}_5
\end{aligned}$$

There is one additional equation which is concerned with the initial statement of the program. The value of a variable is undefined until an assignment defines it. The special label `?` is introduced to mark uninitialized variables. The resulting equation for the incoming information of the first statement is:

$$\text{IN}_1 = \{(v, ?) \mid v \in \text{vars}\} = \{(x, ?), (y, ?), (z, ?)\}$$

Least Solution Together these two classes of equations define how the information flows through the program. We can obtain the actual twelve sets $\text{IN}_1, \ldots, \text{IN}_6$ and $\text{OUT}_1, \ldots, \text{OUT}_6$ by solving these equations. Section 4.1.4 describes an efficient algorithm for this task.

The solution in Figure 4.2 is the *least* solution to the equation system and by this the one we are interested in. It contains the fewest pairs of reaching definitions that is consistent with the program. We could add additional pairs of reaching definitions without making the analysis semantically unsound, but this would make the analysis less usable. Those superfluous definitions still "*may* reach the statement" but we actually know that they can't.

4.1.2. Very Busy Expressions Analysis

The *Very Busy Expressions Analysis* determines the set of very busy expressions for the statements of a program:

> An arithmetic expression *must* be *very busy* at the entry of a program point when on every possible execution path the expression is always used before any of its variables are redefined.

Consider the example program depicted in Figure 4.3: The expressions `a-b` and `b-a` are very busy at the entry of the condition statement. Both are used regardless of the outcome of the condition and neither `a` nor `b` are redefined on a path before their use. The program could be optimized based on this very busy expressions information by precomputing and storing the value of `a-b` and `b-a` before the condition.

Forward and Backward Analyses Very busy expressions are easily computed by starting at the expression and traversing the CFG *backwards* until a variable in the expression is redefined. In this case the expression looses its very busy status by definition and has to be removed from the set of information. The Very Busy Expressions Analysis is therefore defined as a *backward* analysis. This is different to the Reaching Definitions Analysis which propagates the information *forward* with respect to the CFG. A forward analysis combines the outgoing information of the *predecessors* of the current program point:

$$\mathrm{IN}_k = \bigotimes_{i \in \mathrm{pred}(k)} \mathrm{OUT}_i$$

A backward analysis focuses on the *successors* instead and combines their outgoing information:

$$\mathrm{IN}_k = \bigotimes_{i \in \mathrm{succ}(k)} \mathrm{OUT}_i$$

May and Must Analyses The direction is not the only difference between the two analyses. The goal of the Very Busy Expressions Analysis is to find all expressions that *must* be busy at the entry of a program point. Whereas the Reaching Definitions Analysis targets at determining all assignment which *may* reach a statement. A may analysis combines the outgoing information of the predecessors with the set *union* operator to collect the data flow values which reach the program point via any execution path:

$$\mathrm{IN}_k = \bigcup_i \mathrm{OUT}_i$$

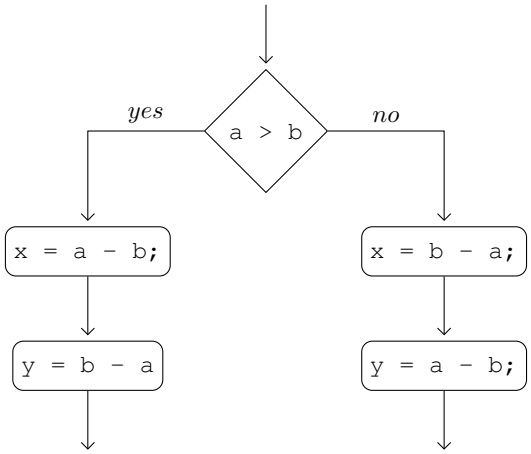
A must analysis uses the set *intersection* operator to obtain only those data flow values which reach the program point independent of the execution path:

$$\mathrm{IN}_k = \bigcap_i \mathrm{OUT}_i$$

Next, I will describe a different way to define the equation system for a data flow analysis.

```
if ([a > b]^1) {
  [x = a - b;]^2
  [y = b - a;]^3
} else {
  [x = b - a;]^4
  [y = a - b;]^5
}
```

(a) Example program



(b) Control flow graph

Figure 4.3.: Example program with corresponding control flow graph taken from [NNH05].

KILL and GEN Sets Instead of listing all equations for the outgoing information as I did for the Reaching Definitions Analysis, I will now express the equation system in terms of *KILL and GEN sets*. This is a common approach to define a data flow analysis.

In the case of a Very Busy Expressions Analysis an expression is *killed* by a statement if it defines a variable which is contained in the expression. I use KILL_k to denote the set of expressions killed by statement k. An expression is *generated* when it is contained in the current statement. The set of expressions produced by statement k is called GEN_k.

The outgoing information for node k can then be described in terms of the incoming information and the sets KILL_k and GEN_k:

$$\mathrm{OUT}_k = (\mathrm{IN}_k \setminus \mathrm{KILL}_k) \cup \mathrm{GEN}_k \tag{4.2}$$

From the set of incoming information all the killed information is removed and the generated information is added. This looks very similar to the equation system listed in Section 4.1.1. Indeed, I could describe the Reaching Definitions Analysis with KILL and GEN sets, too. In case of the Very Busy Expressions Analysis the sets KILL_k and GEN_k for statement k are defined as:

$$\mathrm{GEN}_k = \begin{cases} \mathrm{AExpr}(a), & \text{if } k \text{ is an assignment } \texttt{[x = a;]} \\ \mathrm{AExpr}(b), & \text{if } k \text{ is a boolean condition } \texttt{[b]} \\ \varnothing, & \text{otherwise} \end{cases}$$

$$\mathrm{KILL}_k = \begin{cases} \{a' \in \mathrm{AExpr} \mid x \in \mathrm{vars}(a')\}, & \text{if } k \text{ is an assignment } \texttt{[x = a;]} \\ \varnothing, & \text{otherwise} \end{cases}$$

Here $\mathrm{AExp}(a)$ and $\mathrm{AExp}(b)$ stand for the arithmetic expressions contained in statement a respectively boolean condition b. AExpr is the set of all arithmetic conditions used in the current program.

k	IN_k	OUT_k
1	{a-b, b-a}	{a-b, b-a}
2	{a-b, b-a}	{a-b}
3	{a-b}	∅
4	{a-b, b-a}	{a-b}
5	{a-b}	∅

Figure 4.4.: Solution of the Very Busy Expressions Analysis for the example program.

Largest Solution The following two equations describe the sets of incoming and outgoing information for any backward must analysis. Together with the previous definitions of KILL_k and GEN_k they define the equation system for the Very Busy Expressions Analysis:

$$\text{OUT}_k = (\text{IN}_k \setminus \text{KILL}_k) \cup \text{GEN}_k$$

$$\text{IN}_k = \bigcap_{i \in \text{succ}(k)} \text{OUT}_i$$

We are interested in the largest solution of this equation system. We could always remove very busy expressions without rendering the solution unsound because we are interested in the set of expressions which *must* be very busy. However, it is more useful to know all expressions that are very busy. The largest fixed point solution for the example program is depicted in Figure 4.4.

4.1.3. Monotone Frameworks

The different types of data flow analyses—may and must, forward and backward—can be generalized in a Monotone Framework. The advantage of such a framework includes the possibility to design generic algorithms which can be instantiated for a specific analysis and solve the corresponding data flow equations. A data flow analysis is characterized by

1. A set of *flow values* L
2. A binary *meet operator* $\sqcup$
3. A set of *transfer functions* F

Flow Values The flow values are represented as a lattice L. A lattice $(L, \preccurlyeq)$ is a partially ordered set L of program facts with a binary operator $\preccurlyeq$ that has to be

- *reflexive:* $\forall x \in L. x \preccurlyeq x$,
- *transitive:* $\forall x, y, z \in L. x \preccurlyeq y \wedge y \preccurlyeq z \Rightarrow x \preccurlyeq z$, *and*
- *anti-symmetric:* $\forall x, y \in L. x \preccurlyeq y \wedge y \preccurlyeq x \Rightarrow x = y$.

L is a *complete* lattice if each subset has a least upper and a least lower bound.

	Reaching Definitions	Very Busy Expressions
L	$\wp(\text{vars} \times \text{labels})$	$\wp(\text{AExpr})$
$\preccurlyeq$	$\supseteq$	$\subseteq$
$\sqcup$	$\cup$	$\cap$
$\top$	$\varnothing$	AExp
$\bot$	vars × labels	$\varnothing$

Figure 4.5.: Reaching Definitions and Very Busy Expressions as Monotone Frameworks.

Meet Operator The *meet operator* $\sqcup : \wp(L) \to L$ combines flow values from different paths which merge at a program point. The meet operator $\sqcup$ has to be

- *a closure on L:* $\forall x, y \in L.\, x \sqcup y \in L$,
- *commutative:* $\forall x, y \in L.\, x \sqcup y = y \sqcup x$, *and*
- *associative:* $\forall x, y, z \in L.\, (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$.

The meet operator is closely related to the partial order $\preccurlyeq$ on lattice L:

$$\forall x, y \in L.\, x \preccurlyeq y \Leftrightarrow x \sqcup y = x$$

Transfer Functions The last component of the framework is the set of *transfer functions*. The transfer function f_l models the ability of statement l to modify its local information based on the program point's semantic approximation.

$$F = \{f_l : L \to L \mid l \in \text{labels}\}$$

The transfer functions map the program behavior onto lattices. It is important for the existence of a fixed point that each transfer function is *monotone*.

Discussion The elements of lattice L represent the flow values for a data flow analysis, i.e., the in and out sets. The Reaching Definitions Analysis operates on sets of pairs of variables and labels; the Very Busy Expressions Analysis deals with sets of arithmetic expressions.

A complete lattice L has a unique greatest element $\top$ which represents the *best case* information for a data flow analysis. In the case of a may analysis this is the empty set and for a must analysis this is the set containing all information.

A complete lattice L also has a unique least element $\bot$ which means that the data flow analysis gained no information for this program point. For a may analysis this the set containing all information and for a must analysis this is the empty set.

The meet operator $\sqcup$ merges the information of preceding program points. A may analysis uses the set union $\cup$ as its meet operator and a must analysis the set intersection $\cap$.

The set x is a conservative approximation of y if $x \preccurlyeq y$. For a may analysis $\preccurlyeq$ is the superset operator $\supseteq$ and for a must analysis the subset operator $\subseteq$.

Figure 4.5 phrases the Reaching Definitions Analysis and the Very Busy Expressions Analysis as lattice-based Monotone Frameworks.

4.1.4. Work List Algorithm

The following algorithm calculates the fixed point solution for any data flow analysis which is described as a Monotone Framework. The algorithm takes the control flow graph, a direction d which is either *forward* or *backward*, the lattice L of flow values with least element $\bot$, the meet operator $\sqcup$, and the transfer functions F as input. It returns the fixed point solution, i.e., the sets of incoming and outgoing information for every node of the control flow graph.

Algorithm 1 Work list algorithm for lattice-based data flow analyses.

Input: Control flow graph, direction d, Lattice L with least element $\bot$, meet operator $\sqcup$, and transfer functions F
Output: IN and OUT sets for every node of the control flow graph

```
for all nodes n in the control flow graph do
  IN_n ← nil
  OUT_n ← nil
end for
if direction d is forward then
  s ← first node of the control flow graph
else
  s ← last node of the control flow graph
end if
w.enqueue s
while work list w is not empty do
  n ← w.dequeue
  OUT'_n ← OUT_n
  IN_n ← ⊥
  for all p ∈ pred_d(n) do
    IN_n ← IN_n ⊔ OUT_p
  end for
  OUT_n ← f_n(IN_n) with f_n ∈ F
  if OUT_n ≠ OUT'_n then
    for all s ∈ succ_d(n) do
      if s ∉ w then
        w.enqueue s
      end if
    end for
  end if
end while
```

The algorithm starts with setting all in and out sets to undefined. This is important for a later test which decides if the out set has changed. The algorithm needs to distinguish the empty set from an undefined set to ensure that every node is evaluated at least once.

In the next step the start node s is defined. This is either the first or the last node of the control flow graph depending on the direction d of the analysis. I assume without loss of generality that every control flow graph has both a unique first and last node.

The algorithm calculates the in and out sets for the current node n which is dequeued from the work list w. First the current out set is saved for a later comparison. Then the in set is set to the least element $\perp$ of the lattice L. Now all predecessors (with respect to the direction d) of the current node are iterated. Their sets of outgoing information are combined with the meet operator $\sqcup$ which yields IN_n. The set OUT_n is obtained by applying the flow function f_n for the current node n to the set of incoming information IN_n.

At last, the newly calculated set of outgoing information OUT_n is compared to the previous set OUT'_n. The successors of the current node (again with respect to the direction of the analysis) depend on its set of outgoing information. Every time this set changes the successors' sets of information have to be calculated again. For this they are added to the end of the work list (if they are not already part of it). During the analysis the set of outgoing information changes for every node at least from undefined to defined. This ensures that every node is evaluated at least once.

The algorithm stops when the work list w is finally empty. The resulting sets represent the fixed point solution for the given data flow analysis and the current program.

4.1.5. Limitations

Standard data flow analysis has a number of limitations. The analysis is sound but the over-approximation which occurs when the meet operator is applied to merge paths at a program point can be significant. Furthermore, a data flow analysis does not keep track from where flow values originate. It can therefore not present an example trace for the taint flow from a tainted source to a vulnerable sink. A data flow analysis can only state that a vulnerability occurs at a specific statement not how it was caused.

Most importantly, there is no good approach to refine a data flow analysis. A more detailed lattice of facts requires a new set of transfer functions. Another possibility is to modify the criteria when to apply the meet operator. However, this does not match the advances made in recent years in the area of model checking.

4.2. Syntactic Model Checking

Model checking is an automatic verification technique used for checking properties of hardware and software models. More formally, model checking verifies whether a model M satisfies a property ϕ which would be denoted as $M \vDash \phi$.

A common model used is the *Kripke structure* and the properties are often expressed in one of several forms of temporal logic. I focus here on the *Computational Tree Logic* (CTL) [BAMP81] which is described in one of the next sections. CTL is used to specify properties of the Kripke structure reasoning about paths through its states and their annotations.

Goanna uses model checking to solve static program analysis problems in general and to find bugs in software in particular. The details of how Goanna uses model checking are described by Fehnker *et al.* in [FHJ+07]. The basic idea is to generate the control flow graph (CFG) for a C/C++ program and to label the CFG with occurrences of syntactic constructs. The labeled CFG can be seen as a Kripke structure. CTL is used to specify interesting properties for the program using these labels. The Kripke structure and the formulae are passed to a model checker which generates a counter-example if one of the formulae is violated.

4.2.1. Kripke Structures

A *Kripke structure* is a variation of a non-deterministic finite automaton introduced by Saul Kripke in [Kri63]. Let AP be a set of atomic propositions. A Kripke structure over AP is defined as a 3-tuple $M = (S, E, \mu)$ consisting of

- a set of states S,
- a transition relation $E \subseteq S \times S$ on these states, and
- a labeling function $\mu : S \to \wp(AP)$.

The labeling function μ defines for each state $s \in S$ the set of atomic propositions $\mu(s)$ which hold in s.

4.2.2. Computational Tree Logic

Computational Tree Logic (CTL) is a temporal logic. It models time as a tree-like structure and is used to specify temporal safety and liveness properties. CTL allows the usual logical operators $\neg$, $\vee$, $\wedge$, $\Rightarrow$, and $\Leftrightarrow$ together with the boolean constants *true* and *false*.

CTL has two additional kinds of operators: path quantifiers **A** (all) and **E** (exists) and temporal operators **X** (next), **G** (globally), **F** (finally), **U** (until), and **W** (weak until). Path quantifiers determine the paths on which a property has to hold. This is either $\mathbf{A}\,\phi$ which requires ϕ to hold on every possible path or $\mathbf{E}\,\phi$ which claims that there has to be at least one path which satisfies ϕ.

A temporal operator specifies in which states of a path ϕ has to hold. The path formulae $\mathbf{X}\,\phi$, $\mathbf{G}\,\phi$, and $\mathbf{F}\,\phi$ mean that ϕ has to hold in the next state, in all states, or in some state, respectively. The formula $\phi\,\mathbf{U}\,\psi$ expresses that ϕ holds until ψ holds. This requires that ψ will eventually hold. Whereas $\phi\,\mathbf{W}\,\psi$ includes the possibility that ψ never holds, in which case ϕ has to hold forever. In CTL a temporal operator is always immediately preceded by a path qualifier.

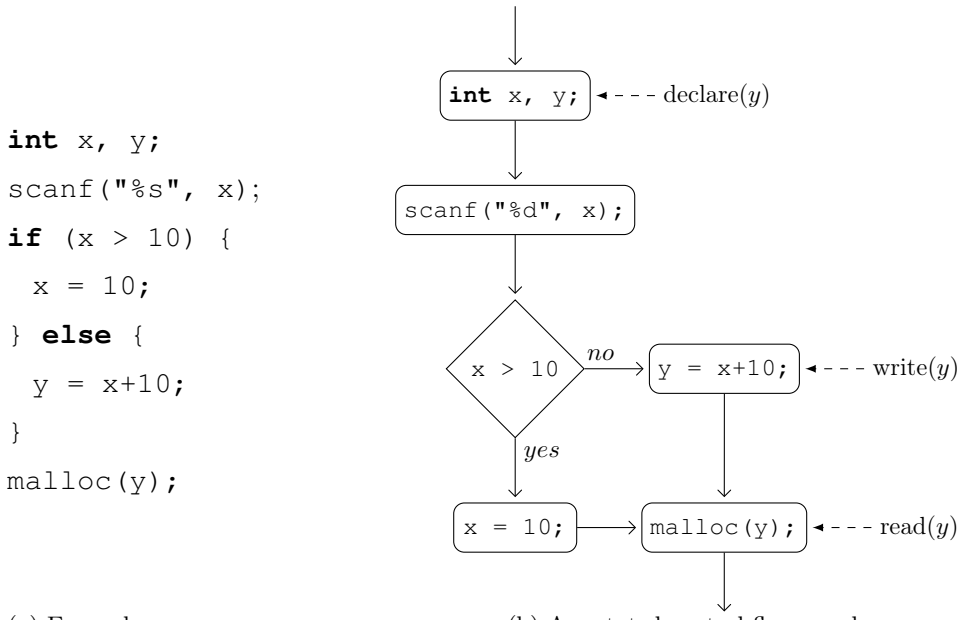
4.2.3. Example Program

I will demonstrate this approach with the simple example program shown in Figure 4.6b. An important property for a program is whether its variables are always defined before they are used. To check for this, I syntactically identify the statements in the program that declare variables, statements that write variables, and statements that read variables. For instance, for the variable `y` of Figure 4.6a I automatically label nodes with `declare(y)`, `write(y)`, and `read(y)`, as shown in Figure 4.6b.

Goanna provides a library of predefined patterns for these and other properties. The patterns are expressed in a tree query language and are evaluated at compile time based on the *abstract syntax tree* which represents the parsed code.

The following CTL formula specifies that after a declaration of `y` the variable is always written before it is read:

$$\mathbf{AG}\,(\texttt{declare(y)} \Rightarrow \mathbf{A}\,(\neg\,\texttt{read(y)}\;\mathbf{W}\;\texttt{write(y)}))$$



(a) Example program (b) Annotated control flow graph

Figure 4.6.: Example program and its annotated CFG for the write-before-read property.

This CTL formula translates to "on all paths and in all states holds that when the state declares y, then for all paths holds that y is not read until written or it is never read". A counter-example for this property is a path that declares y and reads it before it is defined. The example violates the specification because there is a path on which y is used uninitialized.

The combination of syntactical patterns which label the control flow graph and CTL formulae to specify corresponding properties has the advantage that the models are compact. Both CTL and the tree query language are very expressive and can be used to describe a broad variety of properties. Furthermore, model checking is able to automatically generate a counter-example with a trace through the program if a property is violated. This trace helps to explain to the user why a check failed.

4.2.4. Limitations

Model checking is insufficient to reason about data and its flow through a program. This applies to static analysis problems like aliasing, buffer sizes, and—most important for this work—tainted input. A standard approach would encode these properties into the finite state model with variables that model pointers, buffers, and the taint status of data. This would lead to a very large state space and as a result a slower analysis. With the current approach the state space grows in practice about linearly with the number of lines. Introducing a richer semantic model would negate this advantage.

I use a data flow analysis to approximate the data flow behavior of the program. This produces small models which are better suited for model checking. After the data flow analysis the model checking step generates paths which represent vulnerabilities and subjects these paths to a closer analysis with a rich semantic state model. This includes the false positive elimination which detects non-viable paths.

5. Architecture

This chapter presents the architecture of my approach. It describes how the taint analysis fits into the Goanna framework and the interactions between the different components. The taint analysis consists of three main steps:

1. Preprocessing
2. Data Flow Analysis
3. Model Checking

First the source code is preprocessed to create several artifacts which are used by later steps. Afterwards the data flow analysis propagates the taint information in the program. At last, the model checking step generates a counter-example trace which is reported to the user. The different parts and artifacts of my approach and the dependencies between them are depicted in Figure 5.1.

5.1. Preprocessing

In the first step the *source code* is processed to obtain the *abstract syntax tree*. From this the *control flow graph* is built. A *range analysis* takes the control flow graph as input and annotates the variables with their possible ranges.

Source Code The unmodified *source code* is the input for the analysis. Goanna doesn't require the programmer to add annotations to the code, new types for variables, or any other modifications to the original program. Goanna is called instead of the compiler. It performs its static analysis and checks for programming errors. Afterwards, the compiler is invoked and the program is compiled.

Abstract Syntax Tree The *abstract syntax tree* represents the syntactic structure of the source code. Each node stands for a construct occurring in the code. It is different from the parse tree as it does not represent every detail from the real syntax of the program. Therefore it is called the *abstract* syntax tree.

Goanna generates the abstract syntax tree from the source code and offers the ability to retrieve parts of it with a standard tree query language.

Control Flow Graph The *control flow graph* represents all paths which might be followed during a specific execution of the program. Nodes represent statements and edges the possible transitions of the flow of control.

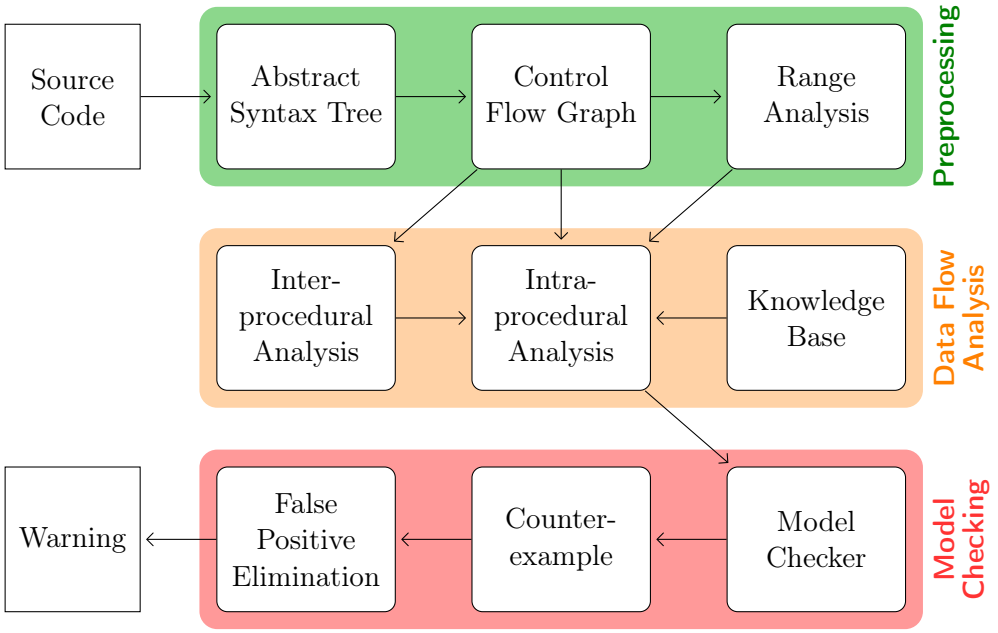


Figure 5.1.: Architecture of the taint analysis embedded in the Goanna model checker.

Range Analysis Goanna employs a *range analysis* which calculates the value ranges for integer variables. This information is already utilized to find buffer overflows. My approach depends on it to figure out if a vulnerability is exploitable.

5.2. Data Flow Analysis

The data flow analysis is the heart of my taint analysis. It tracks the influence of potentially malicious user input in the analyzed program. The data flow analysis itself consists of an *intra-* and an *inter-procedural* part. A *knowledge base* containing information about vulnerabilities and user input functions provides necessary information for both analyses.

Intra-procedural Analysis The *intra-procedural analysis* evaluates one function at a time. It determines sources and sinks. Afterwards, the taint information is propagated and possible vulnerabilities are reported. The intra-procedural analysis is presented in Chapter 6.

Inter-procedural Analysis The *inter-procedural analysis* extracts data flow information from the functions of the program. It determines the functions that return tainted data from user input functions or pass one of their parameters to a vulnerable function. The inter-procedural analysis is further described in Chapter 7.

Knowledge Base The *knowledge base* contains information about user input functions, vulnerable functions, and functions which transfer taint. It contains nearly 250 entries which classify functions of the C standard library. The knowledge base is included in Appendix B.

5.3. Model Checking

The results from the data flow analysis are used as the input for the model checker. The vulnerabilities are described as CTL formula. The model checker evaluates them and generates *counter-example traces*. A counter-example trace is a path through the program which violates a CTL formula. The following *false positive elimination* step determines counter-examples which rely on infeasible paths. All remaining counter-examples are presented as a *warning* to the programmer.

Model Checker The *model checker* verifies if an annotated model of a program fulfills temporal relationships. Goanna uses the labeled control flow graph as its model of the program and the temporal properties are expressed as CTL formula.

Counter-example Goanna automatically generates a *counter-example* when a CTL formula is violated. The *counter-example* is represented as a path in the program. For my taint analysis this is a trace from a user input source to a vulnerable function.

False Positive Elimination The *false positive elimination* removes infeasible paths. A counter-example may be infeasible with respect to the conditions of if statements. For instance when the conditions are not independent from one another the counter-example path could include two branches which are not possible in combination.

Warning All remaining counter-examples are presented as a *warning* to the user. They include the path which led to the vulnerability and a description of the problem.

6. Intra-procedural Analysis

This chapter presents the multi-staged intra-procedural taint analysis. I will first define the data flow analysis which propagates the taint through the program. Afterwards, I will describe the syntactic model checking stage which operates on the result of the data flow analysis to find security vulnerabilities. At last, the number of spurious warnings is reduced with a false positive elimination.

Chapter 7 extends the intra-procedural analysis with an inter-procedural framework to cover vulnerabilities caused by the interaction of multiple functions. The analysis presented here examines every function in isolation and is therefore limited in the set of vulnerabilities it can find. I will start by introducing a program which will serve as a representative example throughout this chapter.

6.1. Running Example

I will use the following program to demonstrate my approach. The program contains several programming errors and is subject to more than one vulnerability for the better illustration of the taint analysis.

```
1  char name[100], query[50];
2  int result;
3  do {
4    scanf("%s", name);
5    unsigned int size = strlen(name);
6    if (size < 50)
7      size = 50;
8    memcpy(query, name, size);
9    result = db_lookup(query);
10   if (!result)
11     printf(name);
12 } while (!result);
```

Listing 6.1: Running example for the intra-procedural analysis.

This program uses `scanf` in line 4 to ask the user for a string which is placed into the 100 character sized array `name`. Next, the actual length of the input string is calculated and checked to be of a valid size. Apparently, the programmer of the example did a simple typing mistake and used the wrong comparison operator in line 6. He intended to ensure that `size`

is between 0 and 50 but instead enforces that `size` is at least 50—I will come back to this later. In line 8 `memcpy()` copies `size` bytes of the input string from `name` to the character array `query` which can store 50 bytes. Afterwards a database is queried with the function `db_lookup()`. If this returns no result, `name` is printed and the user is again asked for input until finally the query is successful.

The source code contains two severe security vulnerabilities which are caused by unvalidated user input. The user provides the value of the variable `name`. Since `size` is derived from `name` it is also under the user's control. These two user-controlled variables are used as arguments to the vulnerable functions `memcpy()` and `printf()`.

This combination of user input and unchecked use leads to the following vulnerabilities:

- A *format string bug* in line 11, which an attacker can exploit by providing dangerous format tokens in the input string. This enables the attacker to read memory values and potentially even execute their own code with the privileges of the vulnerable program.
- A *buffer overflow* in line 8 if the user input is longer than the size of the `query` buffer. The overflow will overwrite the adjacent memory and is described in more detail in Section 6.3.2.

This specific program is characteristic for a larger class of problems, namely those programs where user or third party input enters the program and is later used without proper checking. In the following sections I will present the details of my multi-level approach to deal with these types of security issues. The approach combines two main analysis techniques which in turn consist of several steps:

1. Data Flow Analysis
   a) Finding tainted sources
   b) Propagating taint information
   c) Locating vulnerabilities
2. Model Checking
   a) Generating the model using results from 1.
   b) Defining vulnerabilities as CTL properties
   c) Presenting counter-examples

In the following I will present each of the steps in detail.

6.2. Data Flow Analysis

The first stage of the analysis determines where external input enters the program, where it can potentially flow, and which vulnerable statements might potentially be reached. This requires three steps: finding user input, propagating this taint information along the control flow, and locating vulnerable functions.

Note, that a data flow analysis alone cannot determine a path from the tainted source to the corresponding vulnerable sink. This will be addressed in Section 6.3 where I apply model checking to the results of the data flow analysis.

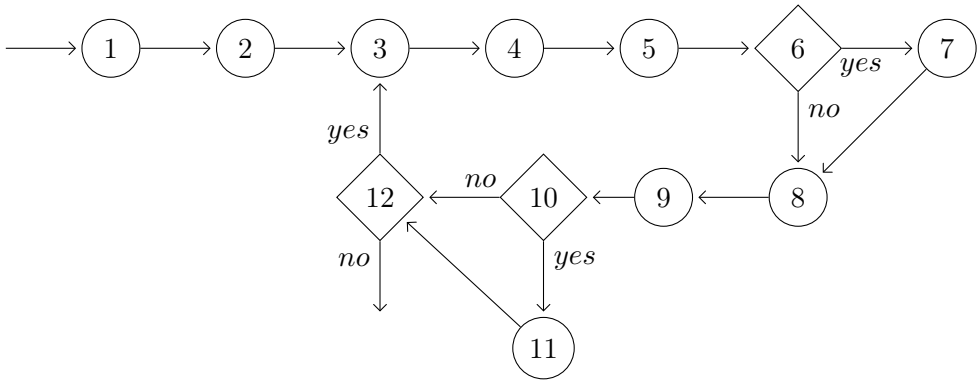


Figure 6.1.: CFG with node numbers that correspond to the line numbers in Listing 6.1.

6.2.1. Finding Tainted Sources

Program functions that are known to return user input are called *tainted sources*. I identified more than 50 of those functions from the C standard library (see Appendix B.1 for a complete list). The user can extend this configuration list to add missing third party functions. Once we know which functions return potentially malicious data, we can identify and track the variables that are likely to be under the attacker's control.

As an example for a tainted source and how I represent it in the configuration list, consider the `scanf()` function used in line 4 of Listing 6.1 which prompts the user and returns input from `stdin`:

```
4 scanf("%s", name);
```

The function `scanf()` parses the input into tokens according to the format specifiers contained in the first parameter. The resulting tokens are stored into the locations provided by the additional parameters. In the example the format string contains only the specifier `%s` which causes `scanf()` to read subsequent characters until it encounters a newline character. The input is stored in the character array `name`.

After a call to `scanf()` all parameters starting with the second as well as its return value are considered to be tainted. These semantics of `scanf()` are represented in the configuration list of user input functions by storing the name of the function, the function parameters that get tainted, and whether the return value gets tainted as well:

$$\begin{aligned} \mathit{scanf} &\in \text{InputFunctions} \\ \text{taintedParameters}(\mathit{scanf}) &= \{2, \ldots, n\} \\ \text{returnTainted}(\mathit{scanf}) &= \text{true} \end{aligned}$$

The entries of this configuration list are retrieved by my data flow analysis to generate the taint information which is propagated along the control flow graph.

6.2.2. Propagating Taints

At this point of the analysis we know which variables are directly controlled by the user. However, this is insufficient for any non-trivial security analysis which can easily be observed in lines 4 and 5 of the example program:

```
4 scanf("%s", name);
5 unsigned int size = strlen(name);
```

The variable `size` depends on the value of `name` which is supplied by the user. As a result the user also has control over `size`. Already these two lines demonstrates the need of a data flow analysis which propagates the taint information across assignments.

I use a classic data flow analysis to determine which variables may be influenced by user input. The information propagate by the taint analysis are the sources of user input. Tainted sources are represented by the name of the variable and the unique identifier of the originating user input function—for simplicity I will just use the corresponding line number in this discussion.

I use this information later to generate a path starting at that input function and ending at a vulnerable function. In the example program user input is assigned to the variable `name` in line 4. This tainted source is therefore represented as (`name`, 4). The variable `name` is subsequently used in an assignment to `size`. The generated taint information is represented as (`size`, 4) because the user input still originates from the `scanf()` function in line 4.

Defining the Flow Equations As described in Section 4.1 the data flow equation system for a forward may analysis is set up by defining for each node k the set of incoming information, equation (4.1), and the set of outgoing information, equation (4.2). This requires in particular to define for each node k the rules when new facts are generated (GEN_k) and old facts are deleted (KILL_k).

Taint information is generated when a variable is either tainted by an input function as described in the previous section or when a tainted variable is used in an assignment, i.e., the tainted data is passed on as we saw in the example. This means:

$$\mathrm{GEN}_k = \mathrm{GEN}_k^{assign} \cup \mathrm{GEN}_k^{input}$$

A user input function can taint a variable either by returning tainted data or by assigning the external input to one of its output parameters:

$$\mathrm{GEN}_k^{input} = \begin{cases} \{(x,k) \mid f \in \mathrm{InputFunctions} \\ \qquad \wedge\, \mathrm{returnTainted}(f)\}, & \text{if } k \text{ is an assignment } x \coloneqq f(\ldots) \\ \{(x_i,k) \mid f \in \mathrm{InputFunctions} \\ \qquad \wedge\, i \in \mathrm{taintedParameters}(f)\}, & \text{if } k \text{ is a function call } f(\ldots,x_i,\ldots) \\ \varnothing, & \text{otherwise} \end{cases}$$

Moreover, I define that in an assignment the assigned variable becomes tainted if a tainted variable is used on the right-hand side. Using the notion $use(t)$ to denote the set of all variables occurring in expression t I define:

k	IN_k	OUT_k
1	—	—
2	—	—
3	—	—
4	(name, 4), (size, 4)	(name, 4), (size, 4)
5	(name, 4), (size, 4)	(name, 4), (size, 4)
6	(name, 4), (size, 4)	(name, 4), (size, 4)
7	(name, 4), (size, 4)	(name, 4)
8	(name, 4), (size, 4)	(name, 4), (size, 4)
9	(name, 4), (size, 4)	(name, 4), (size, 4)
10	(name, 4), (size, 4)	(name, 4), (size, 4)
11	(name, 4), (size, 4)	(name, 4), (size, 4)

Figure 6.2.: Fixed point solution for the taint analysis of Listing 6.1.

$$\mathrm{GEN}_k^{assign} = \begin{cases} \{(x, k') \mid \exists x' \in \mathrm{use}(t) \wedge (x', k') \in \mathrm{IN}_k\} & \text{if } k \text{ is an assignment } x \coloneqq t \\ \varnothing, & \text{otherwise} \end{cases}$$

Having covered the cases that generate taint information, I will next define when variables are cleaned from their taint status. A variable is "killed" when it is redefined by an assignment. Other statements do not kill taint information. Note, however, that by definition of equation (4.2) a variable which is killed by an assignment may afterwards be generated by it again. This happens if the right-hand side contains tainted variables or a function call that taints its return value. I define the set of killed variables at statement k as:

$$\mathrm{KILL}_k = \begin{cases} \{(x, k') \mid \forall k' \in \text{statements}\} & \text{if } k \text{ is an assignment } x \coloneqq t \\ \varnothing, & \text{otherwise} \end{cases}$$

Consider node 8 of the CFG depicted in Figure 6.1 which corresponds to line 8 of the example. This node has incoming edges from node 6 and 7 so their outgoing information is combined to obtain the incoming taint information for node 8:

$$\mathrm{IN}_8 = \bigcup_{i \in \mathrm{pred}(8)} \mathrm{OUT}_i = \mathrm{OUT}_6 \cup \mathrm{OUT}_7$$

As another example, line 5 is an assignment which redefines `size` and by this kills all previous taint information regarding `size`. The tainted variable `name` is used on the right-hand side of the assignment. This taints `size` again and hence (size, 4) is contained in the set GEN_5:

$$\begin{aligned} \mathrm{OUT}_5 &= \mathrm{GEN}_5 \cup (\mathrm{IN}_5 \setminus \mathrm{KILL}_5) \\ &= \{(\text{size}, 4)\} \cup (\mathrm{IN}_5 \setminus \{(\text{size}, *)\}) \end{aligned}$$

Solving the Equation System The set of flow equations over all nodes is iteratively solved until the least fixed point is reached. Since the lattice of facts is the product of a finite number of locations and variable names and since all the flow functions are monotonic, it is guaranteed that a fixed point will be reached.

I use an implementation of the work list algorithm explained in Section 4.1.4 to solve those equations. The resulting fixed point solution for the example is depicted in Figure 6.2. Both `name` and `size` are tainted by input originating from line 4. The taint status of `size` is killed by an assignment in line 7.

The result of the data flow analysis reveals only which variables *may* be tainted since it performs an over-approximation by using the set union as its *meet* operator. I will use model checking in the second stage to eliminate warnings produced by infeasible paths. This will keep the result more precise and reduce the number of false positives.

6.2.3. Locating Vulnerabilities

The next step is to identify if any user input is used as a parameter to a vulnerable function. Similar to the list of user input functions the analysis also maintains a collection of vulnerable functions. This includes format string functions, memory copy and allocation functions, string manipulation functions, as well as the array access operator. Each entry specifies a possible *vulnerability*.

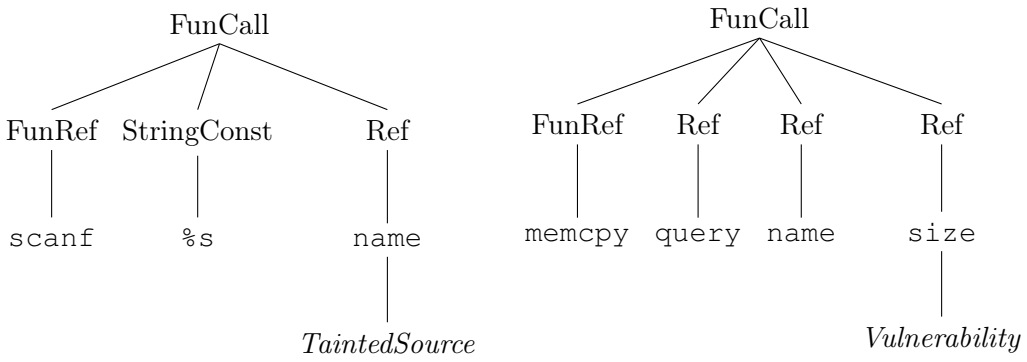
At the moment I consider about 130 different functions from the C standard library (see Appendix B.2 for a complete list). The configuration list contains the name and vulnerable parameters for each function. The semantics of the `memcpy()` function for instance are represented in the configuration list as follows:

$$\begin{aligned} memcpy &\in \text{VulnerableFunctions} \\ \text{vulnerableParameters}(memcpy) &= \{3\} \\ \text{rangeParameter}(memcpy) &= 1 \end{aligned}$$

This means, the function `memcpy()` with signature `memcpy(destination, source, size)` is subject to a vulnerability when user input is passed as the third parameter and not checked to be in the bounds of the first parameter. `memcpy()` causes a buffer overflow when the `size` parameter is larger than the size of the `destination` buffer.

Handing Over to Model Checking In this first stage the taint analysis computed the potential tainted sources and vulnerable sinks as well as generating and killing locations. The following model checking process will use some of this information to validate the actual paths through the program. To do so, the parse tree of the program is annotated with the tainted sources, vulnerable sinks, and the locations which kill taint information.

Figure 6.3a depicts the tainted source annotation which is added to the *Ref* node in the abstract syntax tree which corresponds to the `name` parameter of the `scanf()` function. Figure 6.3b shows the annotation which is added to the *Ref* node in the abstract syntax tree which corresponds to the `size` parameter of `memcpy()` to mark it as a vulnerable sink. Statements which kill taint information are annotated in a similar fashion.



(a) Annotated function call to `scanf()`. (b) Annotated function call to `memcpy()`.

Figure 6.3.: Annotations added to the abstract syntax tree.

6.3. Model Checking

The data flow analysis computes an over-approximation of the statements that can be affected by user input. I use model checking to reduce the number of false positives. A valid path leads from a *tainted source* to a *vulnerability* without encountering a *taint kill* on the way. Such a path represents an exploitable vulnerability. The user is warned with the resulting trace through the source code.

6.3.1. Generating the Model

In the previous analysis step the AST has been annotated with tainted sources, kills and vulnerabilities. An implicit property of the data flow analysis is that after a tainted source for variable `x` and before a taint kill which cleans `x`, `x` is tainted at every node in between. This in particular means, if the analysis can find a path from a source to a vulnerability without any corresponding kill in between, this will be a path along which the variable is tainted at all nodes. I consider this to be a *valid* path.

Translating the existing information into a Kripke structure for model checking, i.e., for finding a path as above, is straightforward: The CFG is directly translated into the transition structure of the Kripke model and the CFG nodes are labeled with their corresponding labels from the AST. The resulting labeled CFG of the example program is shown in Figure 6.4. We see that node 4 is a tainted source, node 8 and node 11 are vulnerabilities, and that the assignment at node 7 kills the taint for `size`.

6.3.2. Defining Vulnerabilities as CTL Properties

As mentioned above, we like to find a path from a source to a sink without an intermediate kill. Moreover, the model checker should generate the path for us as a counter-example. This means, we check for the negation of the above, i.e., that there is no path from a source to a matching sink without some intermediate kill. Should this be violated, the model checker will automatically generate the path we originally were looking for. This is formalized as:

$$\mathbf{AG}\,(\mathit{TaintedSource} \Rightarrow \mathbf{AX}\,(\mathbf{A}\,(\neg\,\mathit{Vulnerability})\,\mathbf{W}\,\mathit{TaintKill}))$$

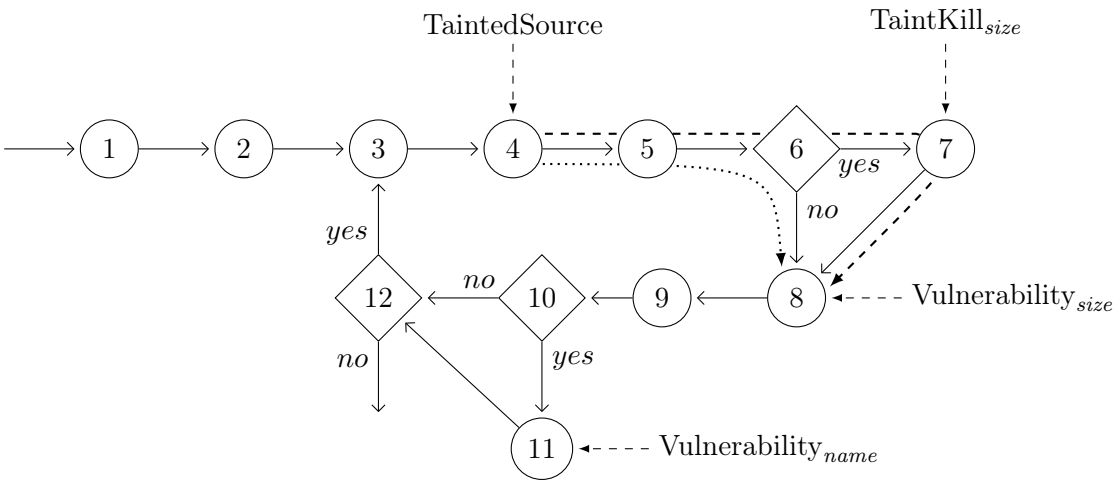


Figure 6.4.: CFG with a valid (dotted) and a spurious (dashed) path to Vulnerability$_{size}$.

The CTL specifies that on all paths every node is either not labeled with *TaintedSource* or for all following nodes holds that there is never a node labeled *Vulnerability* until a node is labeled *TaintKill* or there is never a node labeled *Vulnerability*. Of course, the actual CTL formula is slightly more complex since the *TaintKill* has to correspond to the variables reaching the *Vulnerability* and also the *TaintedSource* has to match the *Vulnerability*. As a result we will have one CTL formula for every occurrence of a vulnerable sink.

6.3.3. Presenting Counter-Examples

When the model checker determines that the CTL formula is not valid for the given Kripke structure it will report a counter-example. Because of the high level of abstraction where branching is interpreted as non-deterministic choice, this path might still be spurious. I will address this in the next section.

As for the example program two paths are reported to the user: The first path starts at the `scanf()` function in line 4 and ends with the use of `name` within the `printf()` function in line 11. The second path also starts at the `scanf` function, takes the false branch—by this skipping line 7—and ends with the use of `size` as the third argument for the `memcpy()` function in line 8. The output generated by Goanna for the second path is as follows:

```
vuln.c:8: User-controlled variable 'size' used as size parameter
1:   main -
2:   main -
3:   main -
4: * main - reference to variable 'name'
5:   main -
6: * main - take the False branch
8: * main - reference to variable 'size'
```

This path is depicted by the dotted arrow in Figure 6.4. The other path represented by the dashed arrow takes the true branch instead. It leads from the tainted source over a taint kill to the vulnerability. The model checker did not report the dashed path because it does not violate the CTL formula which required the path to always contain a relevant taint kill (remember that a vulnerability has to violate the CTL formula).

Both the dotted path and any path to node 11 represent severe vulnerabilities that enable an attacker to compromise the system. The first issue is a format string bug that can be exploited by providing dangerous format tokens to the input string as described in Section 3.2.5. The other vulnerability is a potential buffer overflow. If the user input in line 4 is longer than the size of the `query` buffer it will overflow and `memcpy()` will overwrite the adjacent memory. Buffer overflows have a long history and are ranked as the third most dangerous vulnerability in the current CWE/SANS list [SM11].

6.4. Improvements

The Goanna comes with two techniques to reduce the number of spurious warnings which increases the precision of my taint analysis. The first is an additional interval abstract interpretation that is performed before the other analysis steps. The other improvement is the integration into an existing abstraction refinement framework. I will briefly outline both approaches in the following two sections.

6.4.1. Value Range Validation

As a first step in the improved analysis Goanna runs an interval abstraction interpretation [Cou81]. As a result the integer variables are annotated with their potential value ranges for every relevant node in the CFG. I make use of this information to see if certain vulnerabilities are in fact false positives, because the developer appropriately checked the tainted data before using it.

The developer of my original example simply used the wrong comparison operator in their check of the `size` variable in line 6. Consider the following modification to the example which replaces the comparison operator < with >:

```
6 if (size > 50)
7   size = 50;
```

By correcting the original mistake `size` is always less than or equal to 50. The interval abstract interpretation will be able to also derive this fact. Hence, while the user still has control over the actual value of `size`, there is nonetheless no longer a security issue in the `memcpy()` call because `size` is checked to be smaller than the size of `query`. A buffer overflow is successfully prevented by user input validation.

My analysis utilizes the value range analysis to avoid certain false positives. In this specific example it would know that the lower and upper bound of `size` are valid to access the destination buffer. The warning corresponding to the dotted arrow in the CFG of the running example disappears after this modification of the source code.

6.4.2. Abstraction Refinement

The model of abstraction for a standard data flow analysis is fixed by the definition of its transfer functions. There is no straightforward way to include additional information about loop bounds and conditionals into this model. In the area of model checking program refinement is extensively used to balance between a precise semantic model and a fast analysis.

Goanna provides a refinement for my analysis which can automatically identify spurious vulnerabilities which correspond to impossible execution paths. An analysis which is based solely on the CFG would create counter-examples including such infeasible paths. One possible approach to prevent this is *counter-example guided abstraction refinement* (CEGAR), as used in [HJMS03, CKSY05]. The Goanna team presented a different approach in [FHS10] which computes a *precise least solution* of an interval equation system. This is computationally faster, at the expense of some precision.

The idea is to subject counter-examples to an interval abstract interpretation and check for the feasibility of the path. If the path is infeasible the model is refined with observer automata reflecting the minimal cause for it. The analysis is re-run until a bug disappears or no more infeasible counter-examples occur. The approach is implemented in the Goanna tool which is the foundation of my taint analysis.

Independent of the exact abstraction refinement approach used, program refinement works very well with model checking and is something that is not straightforward to achieve with the classical data flow framework.

7. Inter-procedural Analysis

The inter-procedural analysis extends the taint analysis presented in the previous chapter to vulnerabilities which span several functions. It builds upon a summary-based framework provided by Goanna. The analysis determines which functions pass their parameters to a vulnerable function and are itself vulnerable functions as a result. The other part of the analysis detects functions which return external input and are therefore considered to be user input functions in turn. The resulting function summaries are stored in a database and used to augment the taint analysis.

7.1. Running Example

I will illustrate the approach with an example program that prints the contents of a file. It spans three files and consists of five functions. The source code contains at least two serious security vulnerabilities. The call graph for the program is depicted in Figure 7.1.

The file *main.c* contains the `main()` function which is the starting point for the example. It passes the input returned from `getFile()` in line 7 to `printFile()` in line 10:

```
1 #include <stdio.h>
2 #define LOG_INFO 6
3
4 int main(int argc, char *argv[]) {
5   char *file;
6   printf("Please enter filename:\n");
7   getFile(&file);
8   syslog(LOG_INFO, "Printing the lines of the file ");
9   syslog(LOG_INFO, file);
10  printFile(file);
11 }
```

Listing 7.1: Running example for the inter-procedural analysis, file *main.c*.

The filename returned by `getFile()` is used as a format string for the vulnerable function `syslog()` in line 9. This is the first of two security flaws.

The function `getFile()` is defined in *input.c*. It calls the function `getInput()` in line 12 which returns external input obtained via the C library function `scanf()` in line 5. The input is assigned to the `in` variable which is subsequently assigned to `loop1` and `loop2` after two loop iterations. Finally, `loop2` is assigned to the out parameter `out` in line 17.

```
1 #include <stdio.h>
2 char storage[50];
3
4 char *getInput() {
5   scanf("%50s", storage);
6   return storage;
7 }
8
9 void getFile(char **out) {
10   char *in, *loop1, *loop2;
11   int iterations = 2;
12   in = getInput();
13   do {
14     loop2 = loop1;
15     loop1 = in;
16   } while (iterations-- > 0);
17   *out = loop2;
18 }
```

Listing 7.2: Running example for the inter-procedural analysis, file *input.c*.

The function `printFile()` is included in the file *print.c*. The parameter `file` is concatenated to the variable `command` in line 9. This variable is used as an argument to `runCmd()` in line 10. The function `runCmd()` is a wrapper for the C standard library function `system()` which executes its argument as a system command.

```
1 #include <string.h>
2
3 void runCmd(char *cmd) {
4   system(cmd);
5 }
6
7 void printFile(char *file) {
8   char command[100] = "cat ";
9   strncat(command, file, 50);
10   runCmd(command);
11 }
```

Listing 7.3: Running example for the inter-procedural analysis, file *print.c*.

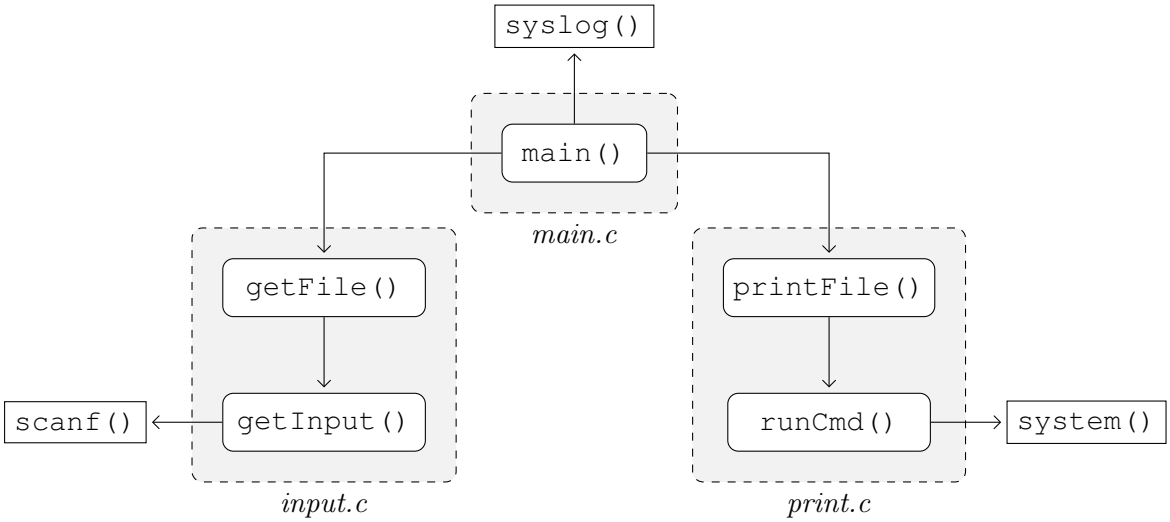


Figure 7.1.: Call graph for the running example depicted in Listings 7.1, 7.2, and 7.3.

The user input which is returned by `scanf()` is in turn returned by `getInput()` and `getFile()`. It is then used as a format string in a call to `syslog()`. Afterwards it is handed over to `printFile()` which again passes it to `runCmd()` where it is finally executed with `system()`. The example program is subject to two severe security flaws:

- A *format string vulnerability* in line 9 of the file *main.c*. The user input returned by `getFile()` is used as a format string in a call to `syslog()`. An attacker can exploit this vulnerability to read and write the main memory by providing dangerous format tokens in the input (see Section 3.2.5).
- A *command injection* in line 10 of the same file because external input is passed to `printFile()` which is at some point executed as a command with `system()`. An attacker can use special shell meta-characters to execute arbitrary commands with the privileges of the affected program (see Section 3.2.2).

The inter-procedural analysis is divided into two steps:

1. Determine Sources and Sinks
   a) Source Analysis
   b) Sink Analysis
2. Extended Taint Analysis
   a) Propagating taints
   b) Locating Vulnerabilities
   c) Presenting counter-examples

First, we will determine which functions in the program are tainted sources and vulnerable sinks. The functions are annotated accordingly. Second, the taint analysis described in the previous chapter is augmented with these annotations.

Function	Return Tainted	Tainted Parameters
`getInput()`	*yes*	$\varnothing$
`getFile()`	*no*	$\{1\}$

Figure 7.2.: Function summaries for tainted sources.

7.2. Finding Sources and Sinks

The analysis consists of two data flow analyses which determine the tainted source and vulnerable sink functions of the analyzed program. Sinks are functions which pass their parameters to vulnerable sinks and sources are functions which return input of tainted sources. Both the source and the sink analysis are performed according to a topological order of the call graph (breaking loops if needed) and repeated until no new annotations are added.

7.2.1. Source Analysis

Functions do not always call user input functions directly. Often the external input is passed through several intermediary functions. The source analysis aims at finding all functions which return user input—directly or indirectly. It is defined as a data flow analysis which is very similar to the taint analysis presented in Chapter 6. The result of the analysis are annotations to the return value and out parameters of a function which denote if they return tainted data.

The source analysis first locates tainted sources in the current function which are either predefined user input functions from the C standard library or other functions of the program which were annotated by a previous run of the source analysis. Afterwards the taint is propagated in the function with a forward may data flow analysis until a fixed point is reached. At last, all possible return values and out parameters are tested for their taint status. Annotations are created accordingly. The source analysis is run for each function of the program until a fixed point is reached.

The definition of the set of generated taint information for node k is extended with GEN_k^{DB}:

$$\mathrm{GEN}_k = \mathrm{GEN}_k^{assign} \cup \mathrm{GEN}_k^{input} \cup \mathrm{GEN}_k^{DB}$$

The generated information from user input functions and assignments is the same as in the intra-procedural analysis. I will only define the taint information generated by transitive tainted sources:

$$\mathrm{GEN}_k^{DB} = \begin{cases} \{(x,k) \mid f \in \mathrm{DB} \\ \qquad \wedge\, \mathrm{returnTainted}(f)\}, & \text{if } k \text{ is an assignment } x \coloneqq f(\ldots) \\ \{(x_i,k) \mid f \in \mathrm{DB} \\ \qquad \wedge\, i \in \mathrm{taintedParameters}(f)\}, & \text{if } k \text{ is a function call } f(\ldots,x_i,\ldots) \\ \varnothing, & \text{otherwise} \end{cases}$$

Figure 7.2 depicts the annotations added for transitive tainted sources in for the example. The function `getInput()` returns input from `scanf()` and is invoke by `getFile()`. The tainted data returned by `getInput()` reaches the first parameter of `getFile()`.

Function	Vulnerable Parameters	Vulnerability Type
`runCmd()`	{1}	Command Injection
`printFile()`	{1}	Command Injection

Figure 7.3.: Function summaries for vulnerable sinks.

7.2.2. Sink Analysis

External input is not always used directly in a vulnerable function. Often the user input is passed through several intermediary functions until it is finally used, which then results in a vulnerability. The sink analysis aims at detecting all functions which pass their parameters to a vulnerable function—either directly or indirectly.

The sink analysis first locates vulnerable sinks and the variables which are used as their arguments. Then a forward may data flow analysis propagates the parameters of the current function along the control flow graph. Afterwards, annotations are added to those parameters which reach an argument of a vulnerable function.

I will define the data flow analysis with GEN_k and KILL_k sets for each statement k. GEN_k is split into the information generated by parameter declarations $\mathrm{GEN}_k^{parameter}$ and the information propagated by assignments GEN_k^{assign}:

$$\mathrm{GEN}_k = \mathrm{GEN}_k^{assign} \cup \mathrm{GEN}_k^{parameter}$$

A parameter declaration k for parameter p generates (p, k):

$$\mathrm{GEN}_k^{parameter} = \begin{cases} \{(p,k) \mid p \in \text{parameters}\}, & \text{if } k \text{ is the declaration of } p \\ \varnothing, & \text{otherwise} \end{cases}$$

Assignments simply propagate the influence of a parameter. If a variable is used on the right-hand side of an assignment which is influenced by a parameter then the variable on the left-hand side is influenced as well:

$$\mathrm{GEN}_k^{assign} = \begin{cases} \{(x,k') \mid \exists x' \in \mathrm{use}(t) \wedge (x',k') \in \mathrm{IN}_k\} & \text{if } k \text{ is an assignment } x \coloneqq t \\ \varnothing, & \text{otherwise} \end{cases}$$

The influence of paremeters is killed when a variable is redefined:

$$\mathrm{KILL}_k = \begin{cases} \{(x,k') \mid \forall k' \in \text{statements}\} & \text{if } k \text{ is an assignment } x \coloneqq t \\ \varnothing, & \text{otherwise} \end{cases}$$

The sink analysis terminates after no new annotations are added to the database. Figure 7.3 shows the annotations for the running example. The function `runCmd()` passes its first parameter to `system()` which is vulnerable for command injection attacks. The function `printFile()` calls `runCmd()` with an argument that is influenced by its first parameter. Both functions are annotated as vulnerable functions.

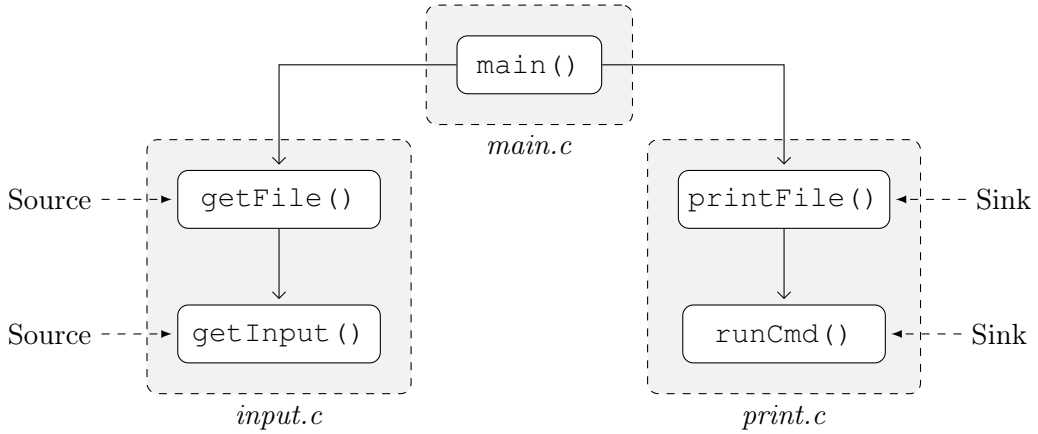


Figure 7.4.: Call graph for the running example annotated with sources and sinks.

7.3. Extended Taint Analysis

Figure 7.4 shows the call graph of the running example with the annotations for source and sink functions. The intra-procedural taint analysis is augmented with this information. It does not only consider predefined functions from the C standard library as tainted sources but also functions of the program which are annotated accordingly. The same is true for vulnerable functions.

7.3.1. Propagating Taints

The taint propagation step of the taint analysis is extended with GEN_k^{DB}. This set generates taint information for functions of the program which are annotated as tainted sources.

$$\mathrm{GEN}_k = \mathrm{GEN}_k^{assign} \cup \mathrm{GEN}_k^{input} \cup \mathrm{GEN}_k^{DB}$$

The definition of set KILL_k does not change:

$$\mathrm{KILL}_k = \begin{cases} \{(x, k') \mid \forall k' \in \mathit{statements}\} & \text{if } k \text{ is an assignment } x \coloneqq t \\ \varnothing, & \text{otherwise} \end{cases}$$

The taint propagation step propagates taint not only from `scanf()` in the `getInput()` function but also the taint returned by `getInput()` in the `getFile()` function and the taint returned by `getFile()` in the `main()` function.

7.3.2. Locating Vulnerabilities

The annotations for vulnerable functions are considered when vulnerable sinks are located after the taint propagation. Additional to the call to `system()` in the `runCmd()` function, the taint analysis considers the call to `runCmd()` in the `printFile()` function and the call to `printFile()` in the `main()` function.

7.3.3. Presenting Counter-Examples

Goanna warns about the two previously mentioned security vulnerabilities in the running example. Both flaws are caused by user input returned by `scanf()` which is in the one case used as a format string for `syslog()` and in the other case as an argument to `system()`.

The warnings contain a trace in the `main()` function because this is the function where user input is returned from a source function and passed to a vulnerable sink.

```
Goanna - analyzing file main.c
Number of functions: 1
main.c:9: warning: Goanna[SEC-format-string] User-controlled
   variable 'file' used as format string
        5:   main -
        6:   main -
        7: * main - the address of 'file'
        8:   main -
        9: * main - reference to variable 'file'
main.c:10: warning: Goanna[SEC-command-injection] User-controlled
   variable 'file' executed as system command
        5:   main -
        6:   main -
        7: * main - the address of 'file'
        8:   main -
        9:   main -
        10: * main - reference to variable 'file'
Total runtime : 0.03 seconds
```

8. Evaluation

This chapter evaluates my approach with respect to its detection rate, runtime performance, and the ability to find real world bugs. The first section presents preliminary results for the Sate IV benchmark which measures the accuracy of the analysis. I evaluated the runtime performance of my analysis with four large open source projects in the next section. The code base of the four projects ranges from 150,000 to 1.8 million lines of code. The third section describes the ability of the analysis to find real vulnerabilities with known security flaws selected from the Common Vulnerabilities and Exposures database.

The results for the detection rate of the analysis are not very expressive on their own. Therefore I tested other tools with selected test cases of the Sate IV benchmark. This allows me to compare my approach with other methods in the last section of this chapter.

8.1. Sate IV Benchmark

Sate IV is the fourth installment of a yearly evaluation conducted by the U.S. National Institute of Standards and Technology (NIST). It is part of the Samate program (see Section 3.1.6) which focuses on the application of static analysis to the security domain. Sate IV was developed to evaluate static analysis tools which target security vulnerabilities.

The benchmark is based on a large testbed of automatically generated test cases which target specific weaknesses. Some of the test cases are purely intra-procedural while others are designed to span several functions. I selected those weaknesses which are relevant to my taint analysis and will present the results in terms of false positives and false negatives. A comparison with other tools in Section 8.4 puts the results of my approach in context.

8.1.1. Test Case Design

A Sate IV test case combines a user input function and a vulnerable function in the presence of a data flow variant. User input functions and vulnerable functions vary for the different weaknesses but the data flow variants stay the same. Every possible combination of these three elements is generated and as a result some of the weaknesses have over a thousand associated test cases. I evaluated my approach with the test cases that belong to the following twelve CWE entries (some of which are described in Section 3.2):

- External Control of System or Configuration Setting (CWE 15)
- Path Traversal (CWE 22)
- Absolute Path Traversal (CWE 36)
- OS Command Injection (CWE 78)

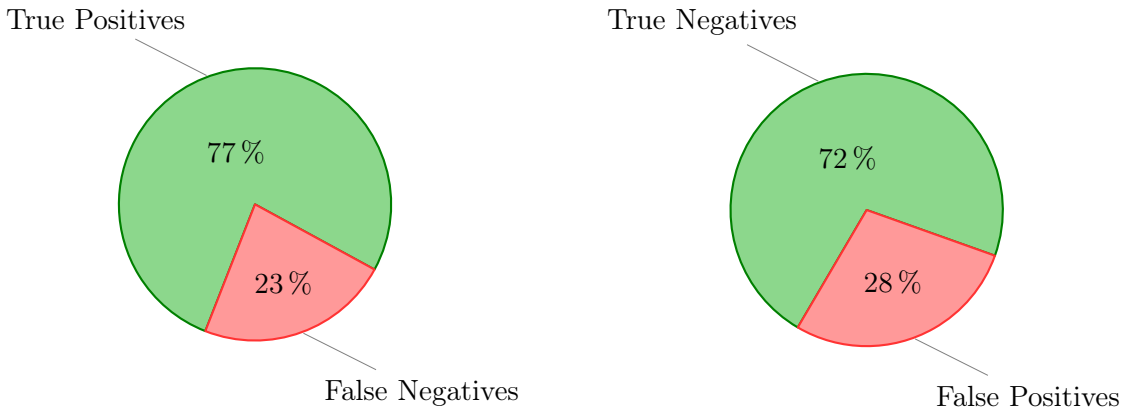
- Process Control (CWE 114)
- Improper Validation of Array Index (CWE 129)
- Uncontrolled Format String (CWE 134)
- Unexpected Sign Extension (CWE 194)
- Signed to Unsigned (CWE 195)
- Uncontrolled Search Path Element (CWE 427)
- Integer Overflow to Buffer Overflow (CWE 680)
- Uncontrolled Memory Allocation (CWE 789)

The selected weaknesses are covered by a total of 12,801 different test cases. The number of test cases varies significantly between different weaknesses. This ranges from only 19 test cases for CWE 15 to 3,700 test cases associated with CWE 78.

Example The following test case belongs to CWE 680 *Integer Overflow to Buffer Overflow*. It is intra-procedural with a `bad()` and a `good()` function. The `data` variable is tainted by `fgets()` and used in the vulnerable function `malloc()`. The loop adds some complexity.

```
1 void bad() {
2   int data = -1;
3   for (int i = 0; i < 1; i++) {
4     char input_buf[512] = "";
5     fgets(input_buf, 512, stdin);
6     data = atoi(input_buf);
7   }
8   malloc(data * sizeof(int));
9 }
10
11 void good() {
12   int data = 20;
13   for (int i = 0; i < 0; i++) {
14     char input_buf[512] = "";
15     fgets(input_buf, 512, stdin);
16     data = atoi(input_buf);
17   }
18   malloc(data * sizeof(int));
19 }
```

Listing 8.1: Modified and shortened test case which belongs to the SATE IV benchmark.



(a) Warnings issued for 77 % of the *bad* functions.

(b) Warnings issued for 28 % of the *good* functions.

Figure 8.1.: Visualization of the warnings for `bad()` and `good()` functions respectively.

Large user input could cause an integer overflow in the calculation in line 8. As a result the amount of memory allocated would be too small. This could lead to a buffer overflow later on in the execution of the program. The allocation in line 18 is not vulnerable because the variable `data` is always 20. The for loop is never executed due to its loop condition.

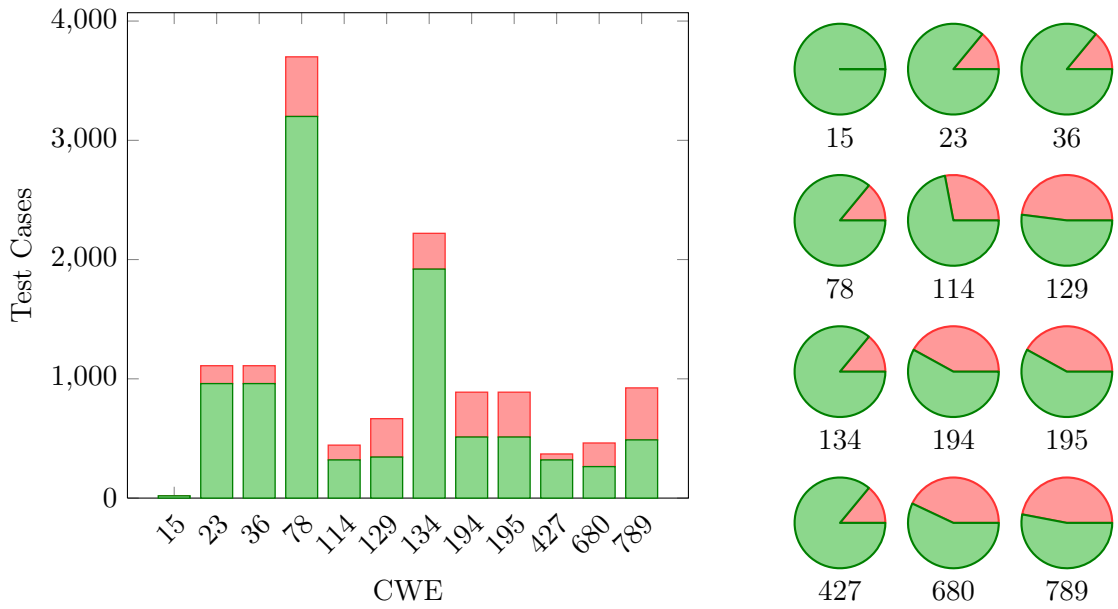
All test cases share the structure of this example: They contain a vulnerable `bad()` function and one or more `good()` functions which are not exploitable. The `good()` functions are safe because some of the associated code is enclosed in a dead code section—in the example the for loop which is never executed. This is not an adequate test for false positives because it only depends on the tool's ability to decide if code is executable.

The SATE IV benchmark considers 68 different flow variants. The subset of weaknesses selected for my evaluation includes test cases which are based on 40 variants. They are listed in Appendix C.1. The first 23 variants describe intra-procedural data flow whereas the other 17 variants include the interaction of at least two functions.

8.1.2. Preliminary Results

I evaluated my approach with 12,801 test cases which cover weaknesses that are caused by unvalidated user input. The test cases consist of one function that is subject to the corresponding weakness and one or more others that are not. I examined how often the analysis warned about vulnerabilities in `bad()` functions. This gives the number of *true positives*. The remaining test cases where the analysis did not warn about a `bad()` function gives the number of *false negatives*. True positives and false negatives are depicted in Figure 8.1a. My approach found 77 % of all vulnerabilities.

The other interesting key indicator is the number of *false positives*. For this I counted how often the tool incorrectly issued a warning for `good()` functions. The number of *true negatives* corresponds to the remaining test cases where the analysis did not report a false positive. False positives and true negatives are visualized in Figure 8.1b. My approach issued false positives for 28 % of all test cases.



(a) Absolute number of correct warnings for each weakness type.

(b) Relative detection rate.

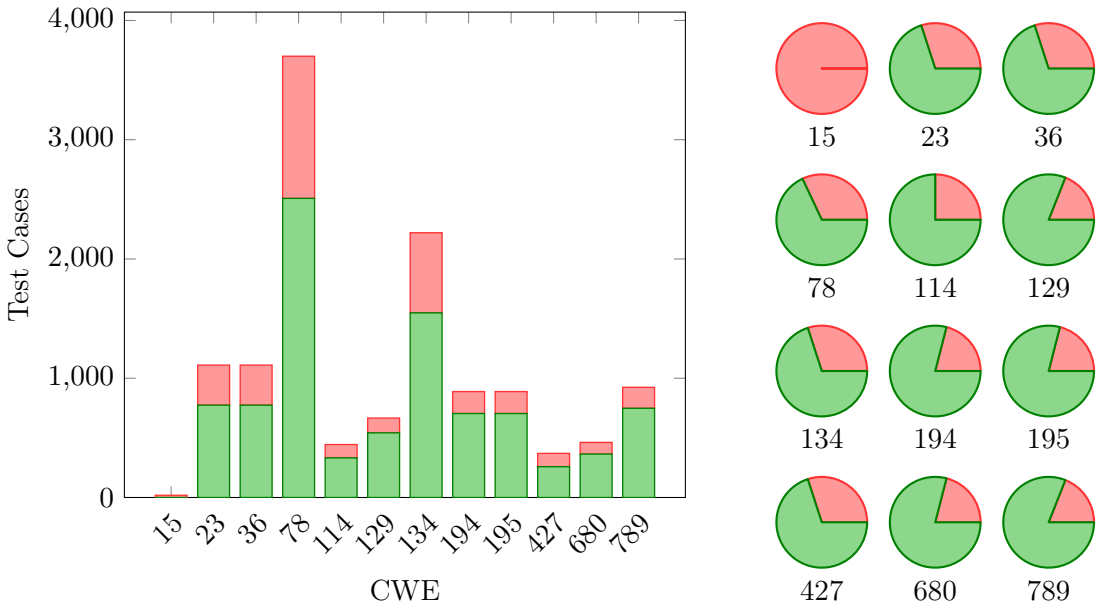
Figure 8.2.: True positives and false negatives for every selected CWE entry.

False Negatives My approach missed vulnerabilities in several test cases belonging to 11 of the 12 weaknesses. These false negatives occur mostly in test cases where constant values are used to trigger problems. The analysis is only able to detect flaws which are caused by external input. Without these test cases the detection rate increases from 77 % to 86 %. The remaining 14 % of the false negatives are caused by the following data flow variants:

- Pointers and C++ references (variants 32, 33)
- Function pointers (variants 44, 65)
- Global variables (variants 45, 68)

The analysis does not model pointer aliasing and is therefore not able to detect vulnerabilities in flow variants 32 and 33. Function pointers are the reason why the tool cannot handle the inter-procedural flow variants 44 and 65. The call graph does not include function calls which are based on function pointers. My taint analysis is not able to detect flaws in the presence of flow variants 45 and 68 because it does not consider global variables. These are no fundamental limitations of the analysis but rather features that have not been implemented yet. Figure 8.2 depicts the absolute and relative number of false negatives per CWE entry.

False Positives I measure false positives by counting the number of warnings for `good()` functions. They contain the same code as the corresponding `bad()` function with the difference that a part of the code which is responsible for the vulnerability lies in a dead code section. For instance, in one of the data flow variants the tainted source or the vulnerable function are part of a for loop which never executes.



(a) Absolute number of false positives for each weakness type.

(b) Relative false positives rate.

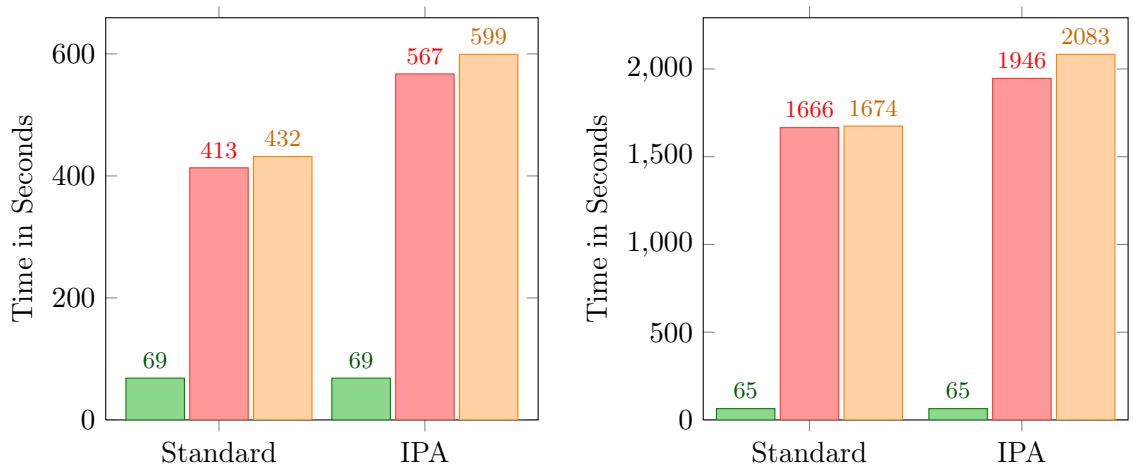
Figure 8.3.: True negatives and false positives for every selected CWE entry.

This reduces the evaluation of false positives to the ability of the tool to detect dead code regions. I think this is a rather unfortunate test case design. It would be more insightful when the `good()` functions instead focused on data flow variants which sometimes miss source or sink or where the data flow from the source does not reach the sink. This would test how accurate the tools model the data flow of the program.

My analysis reported false positives in 28 % of the test cases. The false positives occur for some of the test cases which include the following data flow variants:

- If statements (variants 3–11, 13, 14)
- Switch statement (variant 15)
- For loop (variant 17)

The first type of flow variants which cause false positives are if statements that surround either the tainted source or vulnerable sink with a condition that is never satisfied. Therefore the sink or source are never executed and the vulnerability is not exploitable. The false positive elimination (FPE) cannot decide whether conditions are unsatisfiable when they contain static or const variables or include function calls which return constant values. The second type of flow variants are switch statements that are used with a constant value. The source or sink are contained in a switch case which is never executed. The FPE step does not evaluate which switch cases are executable. The last type of flow variants are for loops with an unsatisfiable condition surrounding source or sink to disable the vulnerability. As with if statements the FPE analysis is not able to analyze certain loop conditions for satisfiability. Figure 8.3 depicts the absolute and relative number of false positives per CWE entry.



(a) Measurements in seconds for *Dovecot*.

(b) Measurements in seconds for *Vim*.

Figure 8.4.: Compilation with *gcc* compared to *Goanna* and *Goanna+DFA* .

8.2. Runtime Performance

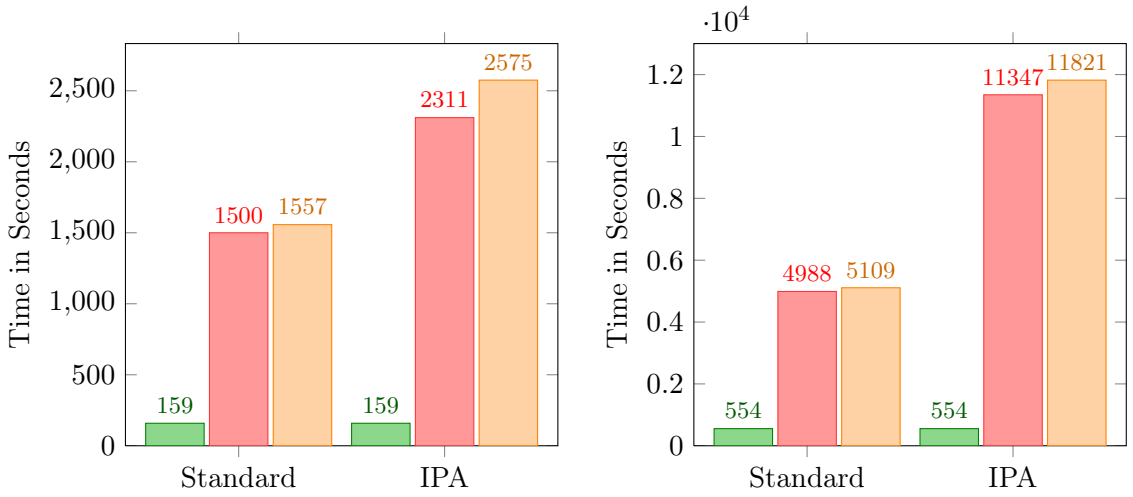
In this section I compare the analysis time for four large open source projects. I used the gcc compiler to obtain the time needed for compilation. Goanna is evaluated in its standard configuration. Goanna+DFA extends Goanna with the data flow framework and checks for security vulnerabilities. The measurements for Goanna and Goanna+DFA also include compilation and Goanna+DFA performs all standard Goanna checks as well.

I compare the intra-procedural analysis (Standard) with the expensive but more accurate inter-procedural analysis (IPA). The compile time is included in both cases as a baseline. All three tools were executed on a workstation with a quad core 2.3 GHz Intel Core i7 CPU, 4 GB of RAM, running Ubuntu 11.04. I used gcc in version 4.5.2 with standard options.

The values presented are averages from at least three measurements. The standard deviation is not depicted in the diagrams since it is too small to be visible. The inter-procedural analysis with Goanna+DFA evaluates on average 1,000 lines of code in 2.3 to 7 seconds. Goanna was configured to spend no more than 90 seconds on the analysis of a single file.

8.2.1. Dovecot 1.2.0

Dovecot is an open source mail server for UNIX-based systems developed by Timo Sirainen. It supports several operating systems including Linux, Solaris, FreeBSD, and Mac OS X. Version 1.2.0 contains 150,000 lines of C code. The intra-procedural analysis with Goanna takes 6.9 minutes which is 6 times longer than the compilation with gcc. Goanna+DFA finishes after 7.2 minutes and is therefore 6.4 times slower than compilation. The data flow framework is responsible for an increase of about 4.6 % in comparison to the standard version of Goanna. In the inter-procedural case Goanna takes 9.6 minutes which is 8.2 times longer than gcc and Goanna+DFA finishes after 10 minutes which is 8.7 times slower than compilation and 5.6 % more than Goanna. Figure 8.4a compares compilation and analysis.



(a) Measurements in seconds for *Ghostscript*.

(b) Measurements in seconds for *Wireshark*.

Figure 8.5.: Compilation with *gcc* compared to *Goanna* and *Goanna+DFA* .

8.2.2. Vim 7.3

Vim is a cross-platform text editor developed by Bram Moolenaar. Version 7.3 contains about 300,000 lines of C code. The compilation with gcc takes 65 seconds which is 25.6 times faster than the analysis with Goanna which finishes after 27.8 minutes. The analysis with security checks takes only slightly longer, namely 27.9 minutes. This is an increase of only 0.5 %. The standard inter-procedural analysis finishes after 32.4 minutes which is 29.9 times the compile time. The taint analysis takes 34.7 minutes which is 32 times slower than gcc and an overhead of 7 % compared to Goanna. The comparison is displayed in Figure 8.4b.

8.2.3. Ghostscript 9.02

Ghostscript is an open source raster image processor which is the basis of several PostScript and PDF viewers. Ghostscript 9.02 is built from 1.1 million lines of C code. The compilation with gcc which takes 2.7 minutes is 9.4 times faster than the standard analysis which finishes after 25 minutes. Compilation is also 9.8 times faster than the analysis with security checks enabled which takes 26 minutes. The data flow framework is responsible for an increase of 3.8 %. The standard inter-procedural analysis takes 38.5 minutes which is 14.5 times the compile time. This increases to 16.2 times for the data flow framework which finishes after 43 minutes. The analysis with security checks adds 11.4 % to the analysis time with standard Goanna. The comparison is depicted in Figure 8.5a.

8.2.4. Wireshark 1.2.0

Wireshark is an open source packet analyzer which is used for the analysis of network traffic. Wireshark 1.2.0 is based on 1.8 million lines of C code. The intra-procedural analysis with standard Goanna finishes after 83.1 minutes which is 9 times slower than compilation.

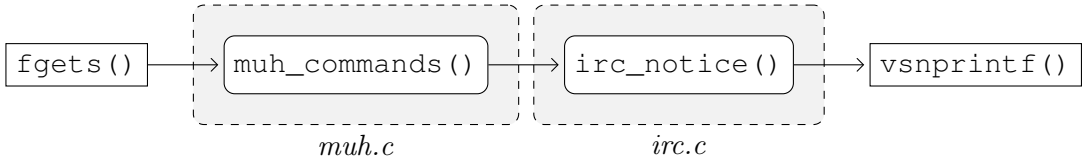


Figure 8.6.: Data flow causing the format string vulnerability in *muh 2.0.5c*.

Goanna with security checks takes 85.2 minutes which is 9.2 times the compile time and an overhead of 2.4 % compared to Goanna. The inter-procedural analysis is finished after 3.2 hours which is 20.5 times longer than gcc. This increases to 21.3 times when the data flow framework is enabled which takes 3.3 hours. The taint analysis takes 4.2 % more time than standard Goanna. Figure 8.5b compares the measurements for Wireshark.

8.3. Bugs in Real Software

Synthetic test cases are designed to cover specific types of vulnerabilities in the presence of various flow variants. This introduces some complexity to achieve a higher degree of realism. However, the resulting test cases are still not close to the complexity of software which was built and extended by different authors over several years. I selected vulnerable versions of an IRC bouncer, an FTP server, and a configuration management software to test my approach with real software. The three open source projects contain format string bugs.

8.3.1. muh 2.05d

muh is an IRC bouncer which acts as an intermediary between the chat client and the IRC server. Versions prior to and including 2.0.5d contain a dangerous format string vulnerability which is described in CVE-2000-0857.

Vulnerability The vulnerability is caused by an interaction of two different functions which are part of two separate files. The function `muh_commands()` from the file *irc.c* reads user input with the `fgets()` function in line 842. This external input is passed as the third argument to `irc_notice()` in line 844.

```
820 void muh_commands(char *command, char *param)
821 {
841   s = (char *)malloc(1024);
842   while(fgets(s, 1023, messagelog)) {
843     if(s[strlen(s) - 1] == '\n') s[strlen(s) - 1] = 0;
844     irc_notice(&c_client, status.nickname, s);
845   }
909 }
```

Listing 8.2: Modified excerpt of the `muh_commands()` function from *muh.c*.

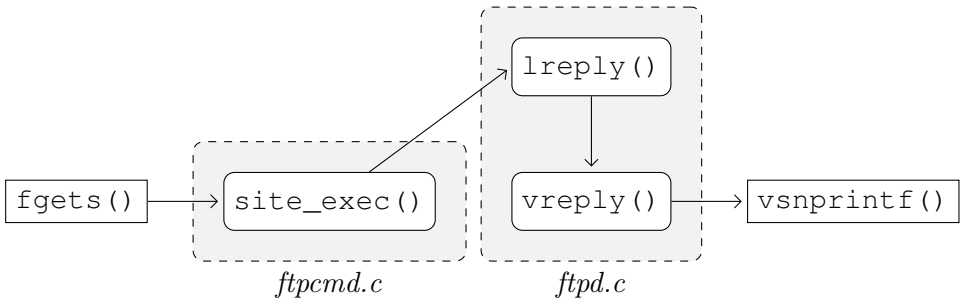


Figure 8.7.: Data flow causing the format string vulnerability in *wu-ftpd 2.6.0*.

The `irc_notice()` function which resides in the file *muh.c* passes its third parameter as a format string to the `vsnprintf()` function in line 263.

```
257 void irc_notice(con_t *con, char nickname[], char *format)
258 {
263     vsnprintf(buffer, BUFFERSIZE - 10, format, va);
267 }
```

Listing 8.3: Modified excerpt of the `irc_notice()` function from *irc.c*.

The `vsnprintf()` function is vulnerable to the format string vulnerability. The interaction between the functions which cause this vulnerability is summarized in Figure 8.6.

Warning The taint analysis warns about a format string vulnerability in line 844 of the *muh.c* file. The following is an excerpt of this warning:

```
muh.c:844: warning: Goanna[SEC-format-string] User-controlled
   variable 's' used as format string
842: * muh_commands - reference to variable 's'
842: * muh_commands - take the True branch
843: * muh_commands - take the False branch
844: * muh_commands - reference to variable 's'
```

The warning describes the following: Variable `s` is tainted in line 842 by the user input function `fgets()`. `s` is then passed to the vulnerable function `irc_notice()` in line 844.

8.3.2. wu-ftpd 2.6.0

A vulnerability was found in the FTP server software *wu-ftpd 2.6.0* in the year 2000 where it was present since 1993 [US-00]. It is a format string vulnerability that allows attackers to remotely gain root access to the server which executes this version of wu-ftpd. The vulnerability is described in CVE-2000-0573.

Vulnerability I have reduced the responsible source code to a large extent. This allows me to briefly describe the reasons for the vulnerability. The following three listings show the core of the problem and contain line numbers that refer to the original files. The *ftpcmd.c* file contains the `site_exec()` function. Line 1930 reads data from the network with the `fgets()` function. The retrieved data is stored in the `buf` variable and passed to the `lreply()` function in line 1935.

```
1865 void site_exec(char *cmd)
1866 {
1930   while (fgets(buf, sizeof buf, cmdf)) {
1935     lreply(200, buf);
1942   }
1949 }
```

Listing 8.4: Modified excerpt of the `site_exec()` function from *ftpcmd.c*.

The other relevant file *ftpd.c* contains the `lreply()` function. The `buf` variable is passed as the second argument. The `lreply()` function uses its parameter `fmt` as the third argument in a call to `vreply()`.

```
5343 void lreply(int n, char *fmt, ...)
5344 {
5353   vreply(USE_REPLY_LONG, n, fmt, ap);
5356 }
```

Listing 8.5: Modified excerpt of the `lreply()` function from *ftpd.c*.

The function `vreply()` uses its third parameter directly as the format string for the `vsnprintf()` function which is part of the C standard library. `vsnprintf()` is subject to the format string vulnerability which I described in Section 3.2.5.

```
5274 void vreply(long flags, int n, char *fmt, va_list ap)
5275 {
5290   vsnprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 :
         sizeof(buf), fmt, ap);
5306 }
```

Listing 8.6: Modified excerpt of the `vreply()` function from *ftpd.c*.

The format string vulnerability in wu-ftpd involves the interaction of three different functions which are located in two separate files. It is clear that only an inter-procedural analysis can find vulnerabilities of this kind. The data flow between the three functions which cause this security flaw is depicted in Figure 8.7.

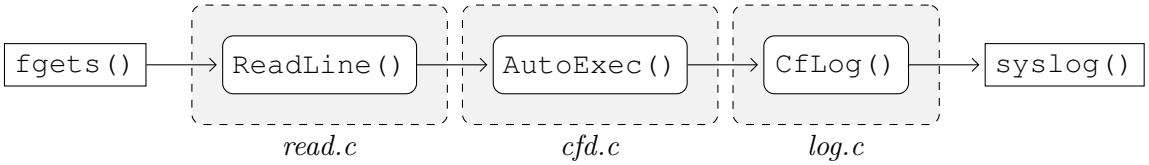


Figure 8.8.: Data flow causing the format string vulnerability in *CFEngine 1.5.x*.

Warning Goanna issues the following warning about a format string vulnerability which I have shortened for brevity of presentation:

```
ftpcmd.c:1935: warning: Goanna[SEC-format-string] User-controlled
   variable 'buf' used as format string
1930: * site_exec - reference to variable 'buf'
1930: * site_exec - take the True branch
1933: * site_exec - take the False branch
1935: * site_exec - reference to variable 'buf'
```

The warning describes the following: The variable `buf` is tainted by `fgets()` in line 1930 and afterwards used in line 1935 as a parameter for the vulnerable `lreply()` function.

8.3.3. CFEngine 1.5.x

CFEngine is an open source configuration management system developed by Mark Burgess. It is designed to manage the configuration of large computer systems which consist of heterogenous devices. The versions 1.5.x of CFEngine are subject to a format string vulnerability. The problem is summarized in CVE-2000-0947.

Vulnerability The vulnerability involves three different files. The `AutoExec()` function in the file *cfd.c* calls `ReadLine()` in line 1124 which taints the `line` variable. The `sprintf()` function copies the string from `line` to `logbuffer` in line 1145. Afterwards `logbuffer` is tainted, too. The newly tainted variable is used in a call to `CfLog()`.

```
1069 void AutoExec()
1071 {
1124   ReadLine(line,1,pp);
1145   sprintf(logbuffer,"%s\n",line);
1146   CfLog(cfinform,logbuffer,"");
1157 }
```

Listing 8.7: Modified excerpt of the `AutoExec()` function from *cfd.c*.

The `ReadLine()` function which resides in the *read.c* file reads the contents of a file with the `fgets()` function. The external input is returned through its first parameter.

```
43 ReadLine(char *buff,int size,FILE *fp)
49 {
53   if (fgets(buff, size, fp) == NULL)
54   {
57   }
70   return true;
71 }
```

Listing 8.8: Modified excerpt of the `ReadLine()` function from *read.c*.

The `CfLog()` function in the *log.c* file passes its second parameter as a format string to the logging function `syslog()`.

```
38 CfLog(enum cfoutputlevel level, char *string, char *errstr)
43 {
128   syslog(LOG_ERR,string,VFQNAME);
156 }
```

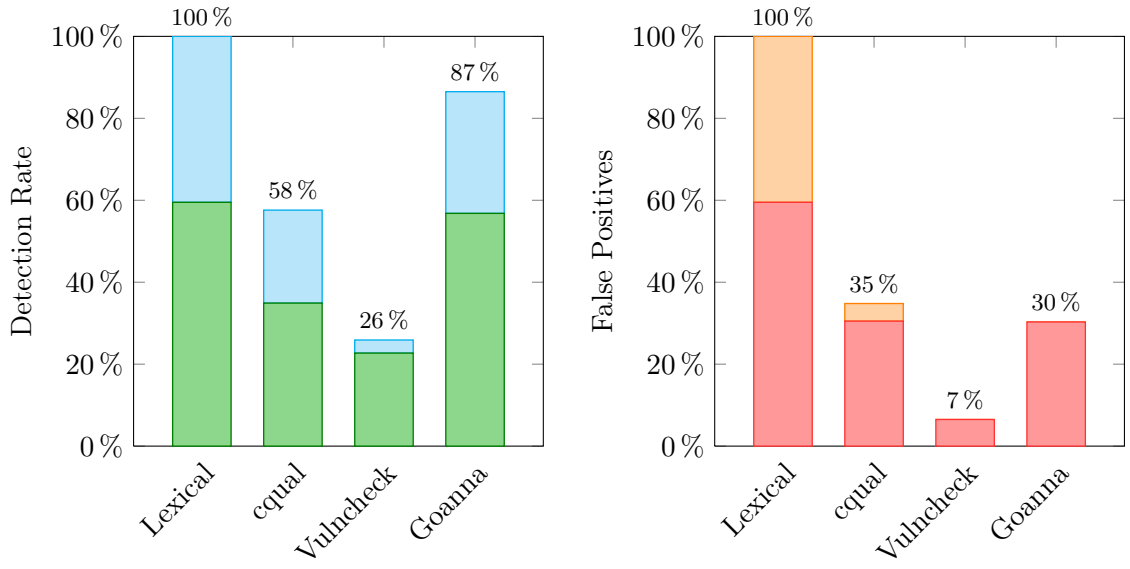
Listing 8.9: Modified excerpt of the `CfLog()` function from *log.c*.

The format string vulnerability in CFEngine is a typical example for a flaw which is difficult to find during testing. It is caused by a malicious input used in a logging function. This is a path of execution which does not occur during normal operation and would require a special test case. Static analysis tools consider every path of a program and hence can find this vulnerability without special treatment.

Warning My taint analysis warns about line 1146 of the *cfd.c* file. This is the line in which the `logbuffer` variable which was indirectly tainted by `ReadLine()` function is passed to the `CfLog()` function. This data flow is summarized in Figure 8.8.

```
cfd.c:1146: warning: Goanna[SEC-format-string] User-controlled
   variable 'logbuffer' used as format string
1124: * AutoExec - reference to variable 'line'
1126: * AutoExec - take the False branch
1134: * AutoExec - take the False branch
1143: * AutoExec - take the True branch
1146: * AutoExec - reference to variable 'logbuffer'
```

Vulnerabilities often span several functions and files as in this case. A complete trace from tainted source to vulnerable sink through the different functions would be helpful for the user to evaluate the warning. Unfortunately, this is currently not possible in Goanna. I will discuss future work which addresses this issue in Chapter 9.



(a) Detection rate for standard and IPA tests. (b) False positives for standard and IPA tests.

Figure 8.9.: Comparison of static analysis tools based on SATE IV format string test cases.

8.4. Tool Comparison

The benchmark results for my taint analysis gain their full meaning only when compared to the outcome of other approaches. I selected five out of the nine tools presented in Chapter 2 for comparison with Goanna: The lexical analysis tools ITS4, RATS and Flawfinder, the type-based approach cqual, and the data flow analysis Vulncheck.

The source code for these five tools is available online. They all cover format string vulnerabilities and have a configuration file of vulnerable functions. I decided to add missing format string and user input functions to all tools in this comparison. This shifts the focus from testing the completeness of their configuration lists to the evaluation of their actual reasoning capabilities. Every tool was evaluated with the format string test cases from the SATE IV benchmark. The results of my comparison—i.e., the detection rate and the number of false positives per tool—are depicted in Figure 8.9.

8.4.1. ITS4, RATS, Flawfinder

ITS4, *RATS* and *Flawfinder* are simple lexical analysis tools. They all miss some format string functions. Adding these to ITS4 resulted in an increase of both the detection and false positive rate from 58.3 % to 100 %. The other tools behaved in exactly the same way.

Lexical analysis tools do not consider user input or the data flow of a program but merely warn about the occurrence of a specific string in the source code. The three tools always report the corresponding false positive when they detect an actual vulnerability in a test case. The results show that they cannot distinguish between a vulnerability and a false positive.

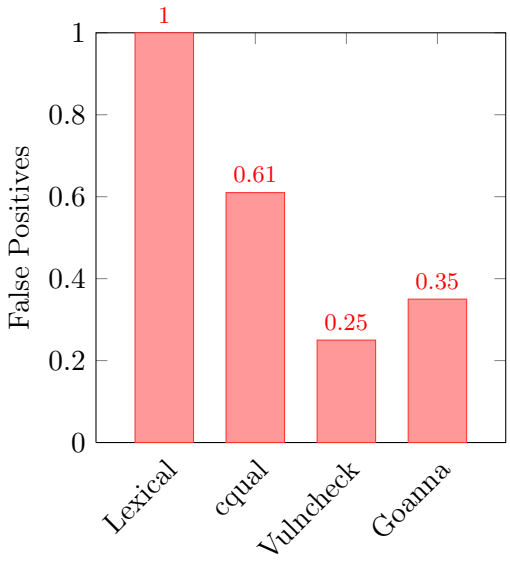


Figure 8.10.: False positives per warning.

8.4.2. cqual

The analysis of *cqual* is based on type annotations provided by the programmer. I decided not to add annotations to the source code which would require additional effort and expertise with the tool. Instead I used its standard configuration file. The file contains annotations for nearly all of the user input, taint transfer, and format string functions used in the SATE IV test cases. I added nine missing functions for the `wchar_t` data type.

The type-based approach of cqual does not achieve the perfect detection rate of the lexical analysis tools. However, the results demonstrate its ability to distinguish between actual vulnerabilities and false positives which I think is more important. The tool takes user input sources into account so that warnings are likely to be accurate.

The before mentioned design of the test cases which tests for false positives based on dead code sections is responsible for most of the false positives reported by cqual. Other false positives are caused by its flow-insensitive approach, an inherent property of type checking: A variable is either considered to be tainted everywhere in a function or nowhere. This can even lead to taint flowing backwards from a user input function to previous statements.

Tainted data passed as void pointer (variant 64) seems to remove the tainted type so that a vulnerability is not detected. cqual also has problems to find flaws when a union data structure is part of the data flow (variant 34). Figure 8.10 compares how many false positives a tool reported on average per warning. cqual reported 0.61 false positives per warning.

8.4.3. Vulncheck

Vulncheck is a compiler extension for gcc which adds warnings for security vulnerabilities. It employs a data flow analysis to track tainted variables and uses a range analysis provided by another extension of gcc to eliminate false positives.

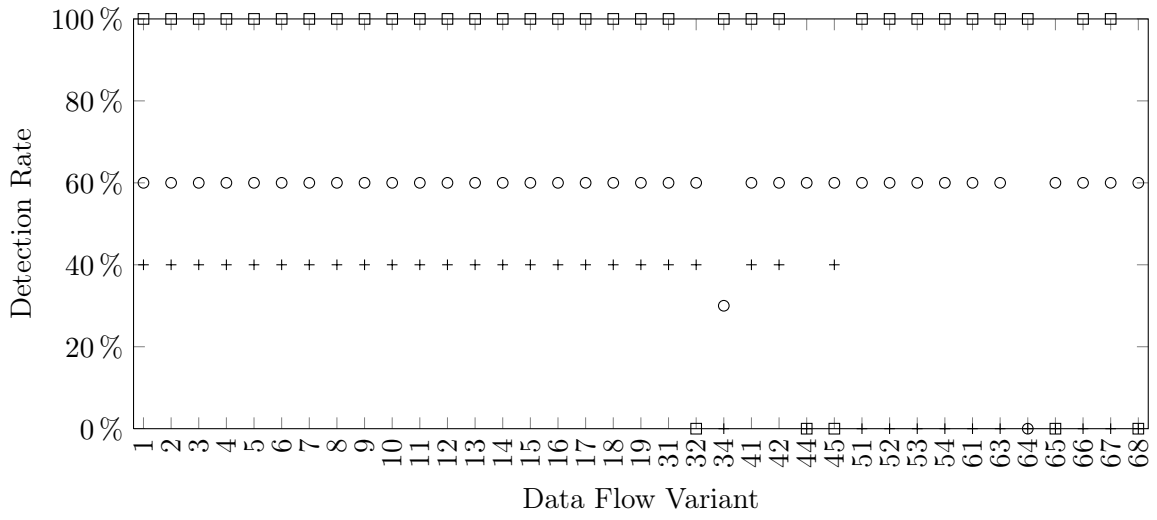


Figure 8.11.: Detection rate by flow variant for Goanna ▫, cqual ○ and Vulncheck +.

When I first tested Vulncheck it failed to report any warnings. The reason is pointer arithmetics used in function parameters like `fgets(data+data_len, ...)`. Vulncheck seems to be confused when an argument is not a single variable. I modified the source code to remove this barrier. Additionally, I added the network input function `recv()` to its list of tainting functions.

Vulncheck still has problems with functions that take a variable argument list. Similar to cqual it fails when the data flow contains a union data structure. Vulncheck only considers taint propagation through assignments and ignores string copy functions like `strcpy()`.

The approach is purely intra-procedural and is therefore not able find vulnerabilities which span several functions. The results suggest that it found some vulnerabilities in inter-procedural test cases. However, this is a side effect of the underlying gcc which inlined the code of some functions. Vulncheck reported on average 0.25 false positives per warning.

8.4.4. Goanna

Goanna achieved the highest detection rate in the comparison only topped by the lexical analysis tools. But in contrast to ITS4, RATS and Flawfinder it reports only 0.35 false positives per warning. The false positive elimination was able to reduce the number of false positives from 35 % to 30.3 %. Similar to cqual this high number of false positives is mainly caused by the design of the test cases.

Apart from this, my approach is not able to detect vulnerabilities in test cases that use pointers for taint propagation (variant 32) due to the lack of adequate pointer aliasing. Inter-procedural test cases which rely on function pointers (variants 44, 65) are also problematic since these are not represented in the internal call graph. Taint which is propagated by global variables is also not modeled by my approach (variants 45, 68).

Figure 8.11 compares the detection rate of Goanna, cqual and Vulncheck by data flow variant. Appendix C.1 describes the different variants.

9. Conclusion

This chapter presents the contributions of my approach which combines data flow analysis and model checking. Next, I will compare my analysis with others from the literature. At last, an outlook discusses possible improvements which could be addressed in future work.

9.1. Contributions

This thesis describes a fully inter-procedural analysis which is reasonably fast and achieves a balance between a high detection rate and few false positives. My combined approach of data flow analysis and model checking covers a broad scope of vulnerabilities.

Combined Approach In this work I described how model checking and data flow analysis can be combined to obtain a fast, precise, traceable, and flexible taint analysis. The data flow analysis provides an efficient way to propagate tainted data in a program and is better suited to deal with data-dependent properties than model checking. However, unlike standard approaches which rely on data flow analysis alone, my combined approach is able to report warnings which incorporate counter-example traces indicating the path leading to a security vulnerability. Moreover, by leveraging model checking techniques I am able to enhance the precision of the analysis in an efficient manner.

Inter-procedural Analysis Most of the security vulnerabilities that occur in software span several functions. They are often caused by the interaction of code written by different developers which rely on different assumptions. Goanna provides a framework for summary-based inter-procedural analyses which allowed me to lift my taint analysis to an inter-procedural level. As a result the analysis is able to detect vulnerabilities caused by user input which cross function boundaries.

Extendability My approach covers a broad variety of different vulnerability types: path traversals, command injections, invalid array accesses, uncontrolled format string problems, buffer overflows, and uncontrolled memory allocations. The configuration files allow the user to add user input, transfer, and vulnerable functions. The generic data flow framework can also be extended to cover other weaknesses which are caused by user input.

Runtime Performance The evaluation results presented in the previous chapter show that the inter-procedural analysis can evaluate 1,000 lines of code in 2.3 to 7 seconds on average. This differs with the complexity of the analyzed program. The data flow framework and the associated security checks are responsible for an overhead of only 4.2 to 11.4 % compared to the standard analysis of a program with Goanna.

Detection Rate The preliminary results for the SATE IV benchmark are promising with an average detection rate of 86 % for vulnerabilities which are caused by external input. The relatively high false positive rate of 28 % can be explained by the design of the test cases. Still Goanna reports on average less false positives per correct warning than comparable tools.

I also demonstrated the ability of my approach to find vulnerabilities in real software by studying format string bugs in three open source projects. My analysis leverages the results of a standard range analysis provided by Goanna to distinguish between exploitable buffer overflows and safe memory accesses.

9.2. Related Work

Several approaches which apply static analysis to the domain of security have been proposed in the literature. This section summarizes those that are similar to my approach. Chapter 2 covers these and others in greater detail.

Lexical Analysis ITS4 [VBKM00], Flawfinder [Whe11] and RATS [Sec11] are lexical analysis tools which match the tokenized source code with a list of dangerous functions. This simple approach is responsible for the high number of false positives reported by these tools. The three tools ignore the data flow of the program and are therefore not able to detect vulnerabilities caused by unvalidated external input.

Annotation-based Analysis Splint [LE01] employs an annotation-based analysis to detect security flaws in C programs. It requires the programmer to add annotations to the program. Splint does not perform an inter-procedural analysis but relies on specific annotations to check locally for vulnerabilities which span several functions. Splint considers the expressions used as actual arguments in its constraints and with this achieves context-sensitivity.

Type-based Analysis The type-based approach by cqual [STFW01] relies on the programmer to add type qualifiers to the program. It facilitates type inference and type checking to detect security flaws as type inconsistencies. cqual also provides a configuration list which provides annotations for functions of the C standard library.

The tool is able to find inter-procedural vulnerabilities by checking the type of parameters against actual arguments. A major drawback of the approach is its flow-insensitivity: a tainted variable is considered to be tainted anywhere in the program independent from the actual statement which tainted it.

Data Flow Analysis Vulncheck [Sot05] is an extension for the open source compiler gcc. It augments the compiler with a data flow analysis. This analysis is used to propagate taint information inside functions.

Vulncheck does not include an inter-procedural analysis and is therefore limited to simpler vulnerabilities which occur inside a single function. Similar to my approach it facilitates a value range analysis to avoid spurious warnings about buffer overflows which are not exploitable because the programmer validated the user input.

9.3. Future Work

The evaluation conducted in the previous chapter revealed several opportunities to improve my approach. This includes certain ways of taint propagation which I do not consider at the moment, a better handling of pointer aliases, and improved warnings for vulnerabilities which involve several functions.

Global Variables Taint can spread via global variables. When one function taints a global variable and another unrelated function passes this variable to a vulnerable function it is difficult to decide if the second function is executed before the first—which would be safe—or afterwards—which would result in a vulnerability. Apart from that it is easy to handle global variables. They could just be treated as additional parameters to every function in the same scope. The analysis would then only need to keep track of tainted global variables and warn about their uses in vulnerable functions.

Pointer Aliasing Another more significant improvement to my approach would be the incorporation of a flow-sensitive alias analysis in the data flow framework. Pointer alias information would improve the existing checks and enable the analysis to detect more sophisticated security vulnerabilities. The challenge is again to balance between the preciseness of the alias analysis and the associated computational cost. The client-driven pointer analysis presented by Guyer and Lin in [GL03] combines a data-flow-based pointer analysis and a client data flow analysis. My approach would benefit from a similar alias analysis.

Structures My analysis treats structures as single objects and does not distinguish individual fields. This is an over approximation and my algorithm would report a false positive when some field of a structure is tainted and another one is used in a vulnerable function. A better abstraction would maintain a taint status for each field of a structure. However, unions should still be handled as one single object because they provide different fields to access the same data. This is even tested in variant 34 of the Sate IV benchmark and detected by my analysis.

Taint Propagation The taint analysis proposed in this work detects functions which return user input obtained from another function and also functions which use one of their parameters in a vulnerable function. Furthermore, it considers several library functions which transfer taint from one parameter to another like `strcpy()`. My approach misses a similar handling of user defined functions. This would require a preceding inter-procedural analysis of the data flow behavior for all functions in a program.

Warnings Detailed warnings for vulnerabilities which span several functions would further improve my approach. At the moment a warning contains a path for the function where the taint is returned from one function and passed to a vulnerable function. A complete path through all participating functions from the source to the sink or at least an additional call stack would help the user to better understand the flaw. This would make it easier for the user to evaluate a warning and decide if it is in fact a vulnerability.

A. Terminology

This section clarifies the meaning of several important keywords which are frequently used in this work. The terminology presented here is based on the *IEEE Standard Glossary of Software Engineering Terminology* [Soc90].

Error Human action that produces an incorrect result. For instance a programmer who introduces a *bug* into a software product.

Fault also *bug*, *flaw* or *defect*. An incorrect step, process, or data definition in a computer program. A fault can lead to a *failure* when it is encountered during the execution of the affected program.

Failure The inability of a software system to perform its required functions with respect to its specification. A failure is discovered when the observed behavior differs from the expected.

Vulnerability also *security flaw*. A fault in the design, implementation, or operation of a software system that could be exploited by an *attacker* which would result in a security relevant *failure*.

Weakness A category which contains *vulnerabilities* of the same type. A program can for instance be subject to a specific `printf()` vulnerability which is an instance of the generic format string weakness. Often used as a synonym for *vulnerability*.

Validation Checking that a software system performs according to its specification and achieves the intended purpose. Generally includes an activity to ensure the absence of *faults* and *vulnerabilities*.

Integrity The assurance that a system will work as intended under all conditions. The integrity of a system includes that information is unmodified and possible modifications would be detected.

Exploit A small piece of software especially written to take advantage of a *vulnerability* in order to cause unintentional or unexpected behavior in other computer software. This violates the *integrity* of the system and may be associated with a gain for the *attacker*.

Attacker A person who *exploits* a *vulnerability* of a computer system. This breaks its *integrity* and may lead to damage for the system or its users.

Abbot *et al.* [ACD+76] explain several terms of software security by drawing an striking analogy to the physical concept of potential and kinetic energy:

"The mere existence of a *flaw* renders an installation *vulnerable*. This is analogous to the engineering concept of "unavailable" potential energy. When an individual (or group) becomes aware of a flaw, an active potential to violate installation *integrity* is achieved—analogous to "available" potential energy. With adequate motivation, skill, resources, and opportunity, this potential is transformed into kinetic energy, and an installation's integrity is penetrated. This penetration of integrity provides the individual with potential access to one or more classes of resources—items of value to an installation or its users. If the individual now chooses, this access may be *exploited* to produce a loss for the installation (such as a loss of information, service, or equipment) and/or a gain for the *attacker*."

B. Knowledge Base

The knowledge base of my taint analysis contains information about user input functions and vulnerable functions. It is used to determine which variables get tainted and where user input triggers a vulnerability.

B.1. User Input Functions

The following tables list input functions for nine different categories: Character Input Functions, Line Input Functions, Formatted Input Functions, Primitive Input Functions, Working Directory Functions, Symbolic Links Functions, Password Functions, Environment Variables Functions, and Network Input Functions. Each row corresponds to a function and provides its name, signature, which parameters are tainted by it, and whether the return value is tainted as well. A short description explains the functions purpose.

Character Input Functions Functions reading characters from external input.

Table B.1.: Character Input Functions

Name	Signature	Parameters	Return
fgetc	`int fgetc(FILE *stream);`	{}	*true*
	Reads the next character from stream and returns it.		
getc	`int getc(FILE *stream);`	{}	*true*
	Similar to the fgetc() function but maybe implemented as a macro.		
getchar	`int getchar(void);`	{}	*true*
	Equivalent to getc(stdin).		
fgetc_ unlocked	`int fgetc_unlocked(FILE *stream);`	{}	*true*
	Non-locking equivalent of the fgetc() function.		
getc_ unlocked	`int getc_unlocked(FILE *stream);`	{}	*true*
	Non-locking equivalent of the getc() function.		
getchar_ unlocked	`int getchar_unlocked(void);`	{}	*true*

Table B.1.: Character Input Functions (cont'd)

Name	Signature	Parameters	Return
	Non-locking equivalent of the getchar() function.		
fgetwc	`wint_t fgetwc(FILE *stream);`	{}	*true*
	Wide-character equivalent of the fgetc() function.		
getwc	`wint_t getwc(FILE *stream);`	{}	*true*
	Wide-character equivalent of the getc() function.		
getwchar	`wint_t getwchar(void);`	{}	*true*
	Wide-character equivalent of the getwchar() function		
fgetwc_ unlocked	`wint_t fgetwc_unlocked(FILE *stream);`	{}	*true*
	Wide-character equivalent of the fgetc_unlocked() function.		
getwc_ unlocked	`wint_t getwc_unlocked(FILE *stream);`	{}	*true*
	Wide-character equivalent of the getc_unlocked() function.		
getwchar_ unlocked	`wint_t getwchar_unlocked(void);`	{}	*true*
	Wide-character equivalent of the getchar_unlocked() function.		
getw	`int getw(FILE *stream);`	{}	*true*
	Reads a word from stream and returns it.		
_IO_getc	`int _IO_getc(_IO_FILE * __fp);`	{}	*true*
	Reads the next character from __fp and returns it.		
_gettc		{}	*true*

Line Input Functions Functions reading lines from external input.

Table B.2.: Line Input Functions

Name	Signature	Parameters	Return
fgets	`char *fgets(char *s, int size, FILE *stream);`	{1}	*true*

Table B.2.: Line Input Functions (cont'd)

Name	Signature	Parameters	Return
	Reads in at most one less than size characters from stream and stores them into the buffer pointed to by s and returns them.		
gets	`char *gets(char *s);`	{1}	*true*
	Reads a line from stdin into the buffer pointed to by s and returns it.		
fgets_ unlocked	`char *fgets_unlocked(char *s, int n, FILE *stream);`	{1}	*true*
	Non-locking equivalent of the fgets() function.		
fgetws	`wchar_t *fgetws(wchar_t *ws, int n, FILE *stream);`	{1}	*true*
	Wide-character equivalent of the fgets() function.		
fgetws_ unlocked	`wchar_t *fgetws_unlocked(wchar_t *ws, int n, FILE *stream);`	{1}	*true*
	Wide-character equivalent of the fgets_ unlocked() function.		
getline	`ssize_t getline(char **lineptr, size_t *n, FILE *stream);`	{1}	*false*
	*Reads an entire line from stream, storing the address of the buffer containing the text into *lineptr.*		
fgetln	`char *fgetln(FILE *stream, size_t *len);`	{}	*true*
	Reads an entire line from stream and returns it.		
fgetline	`char *fgetline(FILE *stream, size_ t *len);`	{}	*true*
	Reads an entire line from stream and returns it.		
getdelim	`ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *stream);`	{1}	*false*
	Similar to the getline() function but the delimiter can be specified.		
__ getdelim	`ssize_t __getdelim(char **lineptr, size_t *n, int delim, FILE *stream);`	{1}	*false*
	Macro equivalent of the getdelim() function.		

Formatted Input Functions Functions reading input according to a format.

Table B.3.: Formatted Input Functions

Name	Signature	Parameters	Return
scanf	`int scanf(const char *format, ...);`	$\{2 \ldots n\}$	$false$
	Reads input from the standard input stream and assigns it to a variable number of arguments.		
fscanf	`int fscanf(FILE *stream, const char *format, ...);`	$\{3 \ldots n\}$	$false$
	Reads input from the stream pointer stream and assigns it to a variable number of arguments.		
vscanf	`int vscanf(const char *format, va_list ap);`	$\{2\}$	$false$
	va_list equivalent of the scanf() function.		
vfscanf	`int vfscanf(FILE *stream, const char *format, va_list ap);`	$\{3\}$	$false$
	va_list equivalent of the fscanf() function.		
wscanf	`int wscanf(const wchar_t *restrict format, ...);`	$\{2 \ldots n\}$	$false$
	Wide-character equivalent of the scanf() function.		
fwscanf	`int fwscanf(FILE *restrict stream, const wchar_t *restrict format, ...);`	$\{3 \ldots n\}$	$false$
	Wide-character equivalent of the fscanf() function.		

Primitive Input Functions Functions reading bytes from external input.

Table B.4.: Primitive Input Functions

Name	Signature	Parameters	Return
read	`ssize_t read(int fd, void *buf, size_t count);`	$\{2\}$	$false$
	Reads up to count bytes from file descriptor fd into the buffer buf.		

Table B.4.: Primitive Input Functions (cont'd)

Name	Signature	Parameters	Return
pread	`ssize_t pread(int fd, void *buf, size_t count, off_t offset);`	{2}	*false*
	Reads up to count bytes from file descriptor fd at offset offset into the buffer starting at buf.		
pread64	`ssize_t pread64(int fildes, void *buf, size_t nbyte, off64_t offset);`	{2}	*false*
	64bit equivalent of the pread() function.		
readv	`ssize_t readv(int fd, const struct iovec *iov, int iovcnt);`	{2}	*false*
	Reads up to count bytes from file descriptor fd into the multiple buffers described by iov.		
preadv	`ssize_t preadv(int d, const struct iovec *iov, int iovcnt, off_t offset);`	{2}	*false*
	Reads up to count bytes from file descriptor fd at offset offset into the multiple buffers described by iov.		
fread	`size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`	{1}	*false*
	Reads nmemb elements of data, each size bytes long, from the stream pointed to by stream, storing them at the location given by ptr.		
fread_ unlocked	`size_t fread_unlocked(void *ptr, size_t size, size_t n, FILE *stream);`	{1}	*false*
	Non-locking equivalent of the fread() function.		
aio_read	`int aio_read(struct aiocb *aiocbp);`	{1}	*false*
	Requests an asynchronous 'n = read(fd, buf, count)' with fd, buf, count given by aiocbp->aio_fildes, aiocbp->aio_buf, aiocbp->aio_nbytes, respectively.		

Working Directory Functions Functions returning the current working directory.

Table B.5.: Working Directory Functions

Name	Signature	Parameters	Return
getcwd	`char *getcwd(char *buf, size_t size);`	{1}	*true*
	Copies an absolute pathname of the current working directory to the array pointed to by buf, which is of length size and returns it.		
getwd	`char *getwd(char *buf);`	{1}	*true*
	Copies an absolute pathname of the current working directory to the array pointed to by buf and returns it.		
get_ current_ dir_name	`char *get_current_dir_name(void);`	{}	*true*
	Returns an absolute pathname of the current working directory.		
g_get_ current_dir	`gchar* g_get_current_dir(void);`	{}	*true*
	Returns an absolute pathname of the current working directory.		

Symbolic Links Functions Functions returning the path of a symbolic link.

Table B.6.: Symbolic Links Functions

Name	Signature	Parameters	Return
readlink	`ssize_t readlink(const char *path, char *buf, size_t bufsiz);`	{2}	*false*
	Places the contents of the symbolic link path into the buffer buf, which has size bufsiz.		

Password Functions Functions reading passwords.

Table B.7.: Password Functions

Name	Signature	Parameters	Return
getpass	`char *getpass(const char * prompt);`	{}	*true*

Table B.7.: Password Functions (cont'd)

Name	Signature	Parameters	Return
	Outputs the string prompt and reads one line (the password) which is returned.		

Environment Variables Functions Functions returning the values of environment variables.

Table B.8.: Environment Variables Functions

Name	Signature	Parameters	Return
getenv	`char *getenv(const char *name);`	{}	*true*
	Searches the environment list for a variable that matches name and returns the value.		
wgetenv	`wchar_t *wgetenv(const wchar_t *varname);`	{}	*true*
	Wide-character equivalent of the getenv() function.		
_wgetenv	`wchar_t *_wgetenv(const wchar_t *varname);`	{}	*true*
	Wide-character equivalent of the getenv() function.		
curl_ getenv	`char *curl_getenv(const char *name);`	{}	*true*
	Portable wrapper of the getenv() function.		
g_getenv	`gchar* g_getenv(const gchar *variable);`	{}	*true*
	Searches the environment list for variable and returns the corresponding value.		
g_get_ home_dir	`gchar* g_get_home_dir(void);`	{}	*true*
	Returns the home directory of the user.		
g_get_ tmp_dir	`gchar* g_get_tmp_dir(void);`	{}	*true*
	Returns a directory to be used for temporary files.		

Network Input Functions Functions reading input from the network.

Table B.9.: Network Input Functions

Name	Signature	Parameters	Return
recv	`ssize_t recv(int s, void *buf, size_t len, int flags);` *Receive a message from a socket.*	{2}	*false*
recvfrom	`ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);` *Receive a message from a socket.*	{2}	*false*
recvmsg	`ssize_t recvmsg(int s, struct msghdr *msg, int flags);` *Receive a message from a socket.*	{2}	*false*

B.2. Vulnerable Functions

The following tables list vulnerable functions for nine different categories: Format String Vulnerabilities, Command Injection Vulnerabilities, String Manipulation Vulnerabilities, Memory Allocation Vulnerabilities, Memory Copy Vulnerabilities, Configuration Vulnerabilities, Path Traversal Vulnerabilities, Process Control Vulnerabilities, and Search Path Vulnerabilities. Each row corresponds to a function and provides its name, signature, and which parameters are considered vulnerable to user input. A short description explains the functions purpose.

Format String Vulnerabilities Unchecked user input is used as the format string parameter of certain formatting functions. An attacker can crash the program or even execute harmful code.

Table B.10.: Format String Vulnerabilities

Name	Signature	Parameters
printf	`int printf(const char *format, ...);` *Writes formatted data to the standard output using a variable number of arguments for expansion.*	{1}
fprintf	`int fprintf(FILE *stream, const char *format, ...);`	{2}

Table B.10.: Format String Vulnerabilities (cont'd)

Name	Signature	Parameters
	Writes formatted data to the specified stream using a variable number of arguments for expansion.	
sprintf	`int sprintf(char *str, const char *format, ...);`	{2}
	Writes formatted data to the specified character array using a variable number of arguments for expansion.	
snprintf	`int snprintf(char *str, size_t size, const char *format, ...);`	{3}
	Writes at most n characters to the specified character array using a variable number of arguments for expansion.	
vprintf	`int vprintf(const char *format, va_list ap);`	{1}
	va_list equivalent of the printf() function.	
vfprintf	`int vfprintf(FILE *stream, const char *format, va_list ap);`	{2}
	va_list equivalent of the fprintf() function.	
vsprintf	`int vsprintf(char *str, const char *format, va_list ap);`	{2}
	va_list equivalent of the sprintf() function	
vsnprintf	`int vsnprintf(char *str, size_t size, const char *format, va_list ap);`	{3}
	va_list equivalent of the snprintf() function.	
wprintf	`int wprintf(const wchar_t *format, ...);`	{1}
	Wide-character equivalent of the printf() function.	
fwprintf	`int fwprintf(FILE *stream, const wchar_t *format, ...);`	{2}
	Wide-character equivalent of the fprintf() function.	
swprintf	`int swprintf(wchar_t *wcs, size_t maxlen, const wchar_t *format, ...);`	{3}
	Wide-character equivalent of the snprintf() function.	

Table B.10.: Format String Vulnerabilities (cont'd)

Name	Signature	Parameters
snwprintf		{3}
	Non-existent wide-character equivalent of the snprintf() function.	
vwprintf	`int vwprintf(const wchar_t *format, va_list args);`	{1}
	Wide-character equivalent of the vprintf() function.	
vfwprintf	`int vfwprintf(FILE *stream, const wchar_t *format, va_list args);`	{2}
	Wide-character equivalent of the vfprintf() function.	
vswprintf	`int vswprintf(wchar_t *wcs, size_t maxlen, const wchar_t *format, va_list args);`	{3}
	Wide-character equivalent of the vsnprintf() function.	
syslog	`void syslog(int priority, const char *format, ...);`	{2}
	Writes formatted data to the system log using a variable number of arguments for expansion.	
vsyslog	`void vsyslog(int priority, const char *format, va_list ap);`	{2}
	Writes formatted data to the system log using a va_ list for expansion.	
obstack_printf		{2}
	Obstack equivalent of the printf() function.	
obstack_vprintf		{2}
	Obstack equivalent of the vprintf() function.	
asprintf	`int asprintf(char **strp, const char *fmt, ...);`	{2}
	Writes formatted data to an allocated character array using a variable number of arguments for expansion.	
vasprintf	`int vasprintf(char **strp, const char *fmt, va_list ap);`	{2}

Table B.10.: Format String Vulnerabilities (cont'd)

Name	Signature	Parameters
	Writes formatted data to an allocated character array using a va_list for expansion.	
setproctitle	`void setproctitle(const char *fmt ...)`	{1}
	Sets the process title that appears in the ps command.	

Command Injection Vulnerabilities Unchecked user input is executed as a command with the same privileges as the vulnerable application.

Table B.11.: Command Injection Vulnerabilities

Name	Signature	Parameters
system	`int system(const char *command);`	{1}
	Executes the argument as a command.	
popen	`FILE *popen(const char *command, const char *type);`	{1}
	Executes the argument as a command.	
execl	`int execl(const char *path, const char *arg, ...);`	$\{1 \ldots n\}$
	Initiates a new program in the same environment in which it is operating.	
execle	`int execle(const char *path, const char *arg, ..., char *const envp[]);`	$\{1 \ldots n\}$
	Executes a process in an environment assigned.	
execlp	`int execlp(const char *file, const char *arg, ...);`	$\{1 \ldots n\}$
	PATH environment equivalent of the execl() function.	
execlpe	`int execlpe(const char *file, const char *arg0, ..., char *const envp[]);`	$\{1 \ldots n\}$
	PATH environment equivalent of the execle() function.	
execv	`int execv(const char *path, char *const argv[]);`	{1,2}

Table B.11.: Command Injection Vulnerabilities (cont'd)

Name	Signature	Parameters
	Similar to execl except that the arguments are passed as null terminated array.	
execve	`int execve(const char *filename, char *const argv[], char *const envp[]);`	$\{1,2,3\}$
	Executes a program.	
execvp	`int execvp(const char *file, char *const argv[]);`	$\{1,2\}$
	PATH environment equivalent of the execv() function.	
execvpe	`int execvpe(const char *file, char *const argv[], char *const envp[]);`	$\{1,2,3\}$
	PATH environment equivalent of the execve() function.	
_spawnl	`int _spawnl(int _mode, const char *_path, const char *_argv0, ...);`	$\{2 \ldots n\}$
_spawnle	`int _spawnle(int _mode, const char *_path, const char *_argv0, ..., char *const _envp[]);`	$\{2 \ldots n\}$
_spawnlp	`int _spawnlp(int _mode, const char *_path, const char *_argv0, ...);`	$\{2 \ldots n\}$
_spawnlpe	`int _spawnlpe(int _mode, const char *_path, const char *_argv0, ..., char *const _envp[]);`	$\{2 \ldots n\}$
_spawnv	`int _spawnv(int _mode, const char *_path, char *const _argv[]);`	$\{2,3\}$
_spawnve	`int _spawnve(int _mode, const char *_path, char *const _argv[], char *const _envp[]);`	$\{2,3,4\}$

Table B.11.: Command Injection Vulnerabilities (cont'd)

Name	Signature	Parameters
_spawnvp	`int _spawnvp(int _mode, const char *_path, char *const _argv[]);`	{2,3}
_spawnvpe	`int _spawnvpe(int _mode, const char *_path, char *const _argv[], char *const _envp[]);`	{2,3,4}
_wsystem	`int _wsystem(const wchar_t *command);` *Wide-character equivalent of the system() function.*	{1}
wpopen	`FILE *_wpopen(const wchar_t *command, const wchar_t *mode);` *Wide-character equivalent of the popen() function.*	{1}
wexecl	`intptr_t _wexecl(const wchar_t *cmdname, const wchar_t *arg0, ...);` *Wide-character equivalent of the execl() function.*	{1...n}
wexeclp	`intptr_t _wexeclp(const wchar_t *cmdname, const wchar_t *arg0, ...);` *Wide-character equivalent of the execlp() function.*	{1...n}
wexecle	`intptr_t _wexecle(const wchar_t *cmdname, const wchar_t *arg0, ...);` *Wide-character equivalent of the execle() function.*	{1...n}
wexeclpe	`intptr_t _wexeclpe(const wchar_t *cmdname, const wchar_t *arg0, ..., const wchar_t *const *envp);` *Wide-character equivalent of the execlpe() function.*	{1...n}
wexecv	`intptr_t _wexecv(const wchar_t *cmdname, const wchar_t *const *argv);` *Wide-character equivalent of the execv() function.*	{1,2}

Table B.11.: Command Injection Vulnerabilities (cont'd)

Name	Signature	Parameters
wexecve	`intptr_t _wexecve(const wchar_t *cmdname, const wchar_t *const *argv, const wchar_t *const *envp);` *Wide-character equivalent of the execve() function.*	$\{1,2,3\}$
wexecvp	`intptr_t _wexecvp(const wchar_t *cmdname, const wchar_t *const *argv);` *Wide-character equivalent of the execvp() function.*	$\{1,2\}$
wexecvpe	`intptr_t _wexecvpe(const wchar_t *cmdname, const wchar_t *const *argv, const wchar_t *const *envp);` *Wide-character equivalent of the execvpe() function.*	$\{1,2,3\}$
_wspawnl	`int _wspawnl(int _mode, const wchar_t *_path, const wchar_t *_argv0, ...);` *Wide-character equivalent of the _wspawnl() function.*	$\{2 \ldots n\}$
_wspawnle	`int _wspawnle(int _mode, const wchar_t *_path, const wchar_t *_argv0, ..., wchar_t *const _envp[]);` *Wide-character equivalent of the _spawnle() function.*	$\{2 \ldots n\}$
_wspawnlp	`int _wspawnlp(int _mode, const wchar_t *_path, const wchar_t *_argv0, ...);` *Wide-character equivalent of the _spawnlp() function.*	$\{2 \ldots n\}$
_wspawnlpe	`int _wspawnlpe(int _mode, const wchar_t *_path, const wchar_t *_argv0, ..., wchar_t *const _envp[]);` *Wide-character equivalent of the _spawnlpe() function.*	$\{2 \ldots n\}$
_wspawnv	`int _wspawnv(int _mode, const wchar_t *_path, wchar_t *const _argv[]);` *Wide-character equivalent of the _spawnv() function.*	$\{2,3\}$
_wspawnve	`int _wspawnve(int _mode, const wchar_t *_path, wchar_t *const _argv[], wchar_t *const _envp[]);`	$\{2,3,4\}$

Table B.11.: Command Injection Vulnerabilities (cont'd)

Name	Signature	Parameters
	Wide-character equivalent of the _spawnve() function.	
_wspawnvp	`int _wspawnvp(int _mode, const wchar_t *_path, wchar_t *const _argv[]);`	{2, 3}
	Wide-character equivalent of the _spawnvp() function.	
_wspawnvpe	`int _wspawnvpe(int _mode, const wchar_t *_path, wchar_t *const _argv[], wchar_t *const _envp[]);`	{2, 3, 4}
	Wide-character equivalent of the _spawnvpe() function.	
fexecve	`int fexecve(int fd, char *const argv[], char *const envp[]);`	{1}
	File descriptor equivalent of the execve() function.	
WinExec	`UINT WINAPI WinExec(LPCSTR lpCmdLine, UINT uCmdShow);`	{1}
	Executes the argument as a command.	
ShellExecute	`HINSTANCE ShellExecute(HWND hwnd, LPCTSTR lpOperation, LPCTSTR lpFile, LPCTSTR lpParameters, LPCTSTR lpDirectory, INT nShowCmd);`	{1}
	Executes the third argument as a command.	
ShellExecuteA	`HINSTANCE ShellExecuteA(HWND hwnd, LPCTSTR lpOperation, LPCTSTR lpFile, LPCTSTR lpParameters, LPCTSTR lpDirectory, INT nShowCmd);`	{1}
	ANSI variant of the ShellExecute() function.	
ShellExecuteW	`HINSTANCE ShellExecuteW(HWND hwnd, LPCTSTR lpOperation, LPCTSTR lpFile, LPCTSTR lpParameters, LPCTSTR lpDirectory, INT nShowCmd);`	{1}
	Wide-character variant of the ShellExecute() function.	
ShellExecuteEx	`BOOL ShellExecuteEx(LPSHELLEXECUTEINFO lpExecInfo);`	{1}
	Executes the argument as a command.	

Table B.11.: Command Injection Vulnerabilities (cont'd)

Name	Signature	Parameters
ShellExecuteExA	`BOOL ShellExecuteExA(LPSHELLEXECUTEINFO lpExecInfo);` *ANSI variant of the ShellExecuteEx() function.*	{1}
ShellExecuteExW	`BOOL ShellExecuteExW(LPSHELLEXECUTEINFO lpExecInfo);` *Wide-character variant of the ShellExecuteEx() function.*	{1}

String Manipulation Vulnerabilities Unchecked user input is copied into the destination string buffer. An attacker could trigger a buffer overflow.

Table B.12.: String Manipulation Vulnerabilities

Name	Signature	Parameters
strcpy	`char *strcpy(char *dest, const char *src);` *Copies the src string the dest character array.*	{2}
stpcpy	`char *stpcpy(char *dest, const char *src);` *Copies the src string the dest character array.*	{2}
strcat	`char *strcat(char *dest, const char *src);` *Appends the src string to the dest character array.*	{2}
wcscpy	`wchar_t *wcscpy(wchar_t *dest, const wchar_t *src);` *Wide-character equivalent of the strcpy() function.*	{2}
wcpcpy	`wchar_t *wcpcpy(wchar_t *dest, const wchar_t *src);` *Wide-character equivalent of the stpcpy() function.*	{2}
wcscat	`wchar_t *wcscat(wchar_t *dest, const wchar_t *src);` *Wide-character equivalent of the strcat() function.*	{2}

Table B.12.: String Manipulation Vulnerabilities (cont'd)

Name	Signature	Parameters
strccpy	`char *strccpy(char *output, const char *input);`	{2}
	Copies the input string to the output character array.	
strcadd	`char *strcadd(char *output, const char *input);`	{2}
	Copies the input string to the output character array.	
strecpy	`char *strecpy(char *output, const char *input, const char *exceptions);`	{2}
	Copies the input string to the output character array.	
streadd	`char *streadd(char *output, const char *input, const char *exceptions);`	{2}
	Copies the input string to the output character array.	

Memory Allocation Vulnerabilities Unchecked user input is used to determine the size of memory to allocate. An attacker could exploit an integer overflow to trigger a buffer overflow.

Table B.13.: Memory Allocation Vulnerabilities

Name	Signature	Parameters
malloc	`void *malloc(size_t size);`	{1}
	Allocates size bytes and returns a pointer to the allocated memory.	
calloc	`void *calloc(size_t nmemb, size_t size);`	{1,2}
	Allocates an array of nmemb elements of size bytes each and returns a pointer to the allocated memory.	
realloc	`void *realloc(void *ptr, size_t size);`	{2}
	Changes the size of the memory block to size.	
valloc	`void *valloc(size_t size);`	{1}
	Allocates size bytes and returns a pointer to the allocated memory.	
memalign	`void *memalign(size_t boundary, size_t size);`	{2}

Table B.13.: Memory Allocation Vulnerabilities (cont'd)

Name	Signature	Parameters
	Allocates size bytes and returns a pointer to the allocated memory.	
posix_memalign	`int posix_memalign(void **memptr, size_t alignment, size_t size);`	{3}
	Allocates size bytes and places the address of the allocated memory in memptr.	
alloca	`void *alloca(size_t size);`	{1}
	Allocates size bytes of space in the stack frame of the caller.	
operator new[]	`void* operator new[] (std::size_t size) throw (std::bad_alloc);`	{1}
	Allocates size bytes of memory.	

Memory Copy Vulnerabilities Unchecked user input is used to determine the size of the memory to copy into the destination buffer. An attacker could trigger a buffer overflow.

Table B.14.: Memory Copy Vulnerabilities

Name	Signature	Parameters
memcpy	`void *memcpy(void *dest, const void *src, size_t n);`	{3}
	Copies n bytes from memory block src to memory block dest.	
mempcpy	`void *mempcpy(void *dest, const void *src, size_t n);`	{3}
	Copies n bytes from memory block src to memory block dest.	
memmove	`void *memmove(void *dest, const void *src, size_t n);`	{3}
	Copies n bytes from memory block src to memory block dest.	
memset	`void *memset(void *s, int c, size_t n);`	{3}
	Fills the first n bytes of the memory area pointed to by s with the constant byte c.	
memccpy	`void *memccpy(void *dest, const void *src, int c, size_t n);`	{4}

Table B.14.: Memory Copy Vulnerabilities (cont'd)

Name	Signature	Parameters
	Copies n bytes from memory block src to memory block dest, stopping when the character c is found.	
wmemcpy	`wchar_t *wmemcpy(wchar_t *dest, const wchar_t *src, size_t n);` *Wide-character equivalent of the memcpy() function.*	{3}
wmempcpy	`wchar_t *wmempcpy(wchar_t *dest, const wchar_t *src, size_t n);` *Wide-character equivalent of the mempcpy() function.*	{3}
wmemmove	`wchar_t *wmemmove(wchar_t *dest, const wchar_t *src, size_t n);` *Wide-character equivalent of the memmove() function.*	{3}
wmemset	`wchar_t *wmemset(wchar_t *wcs, wchar_t wc, size_t n);` *Wide-character equivalent of the memset() function.*	{3}
bcopy	`void bcopy(const void *src, void *dest, size_t n);` *Copies n bytes from memory block src to memory block dest.*	{3}
CopyMemory	`void CopyMemory(PVOID Destination, const VOID *Source, SIZE_T Length);` *Copies n bytes from memory block Source to memory block Destination.*	{3}
MoveMemory	`void MoveMemory(PVOID Destination, const VOID *Source, SIZE_T Length);` *Copies n bytes from memory block src to memory block dest.*	{3}
strlcat	`size_t strlcat (char *dst, const char* src, size_t siz);` *Appends the src string to the dest character array of size siz.*	{3}
strlcpy	`size_t strlcpy (char* dst, const char* src, size_t siz);` *Copies the src string to the dest character array of size siz.*	{3}

Table B.14.: Memory Copy Vulnerabilities (cont'd)

Name	Signature	Parameters
strncpy	`char *strncpy(char *dest, const char *src, size_t n);` *Copies at most n characters from src to dest.*	{3}
stpncpy	`char *stpncpy(char *dest, const char *src, size_t n);` *Copies at most n characters from src to dest.*	{3}
strncat	`char *strncat(char *dest, const char *src, size_t n);` *Appends at most n characters from src to dest.*	{3}
wcsncpy	`wchar_t *wcsncpy(wchar_t *dest, const wchar_t *src, size_t n);` *Wide-character equivalent of the strncpy() function.*	{3}
wcpncpy	`wchar_t *wcpncpy(wchar_t *dest, const wchar_t *src, size_t n);` *Wide-character equivalent of the stpncpy() function.*	{3}
wcsncat	`wchar_t *wcsncat(wchar_t *dest, const wchar_t *src, size_t n);` *Wide-character equivalent of the strncat() function.*	{3}
strxfrm	`size_t strxfrm(char *dest, const char *src, size_t n);` *Places the first n characters of the transformed string in dest.*	{3}
snprintf	`int snprintf(char *str, size_t size, const char *format, ...);` *Writes at most n characters to the specified character array using a variable number of arguments for expansion.*	{2}
vsnprintf	`int vsnprintf(char *str, size_t size, const char *format, va_list ap);` *va_ list equivalent of the snprintf() function.*	{2}
swprintf	`int swprintf(wchar_t *wcs, size_t maxlen, const wchar_t *format, ...);`	{2}

Table B.14.: Memory Copy Vulnerabilities (cont'd)

Name	Signature	Parameters
	Wide-character equivalent of the snprintf() function.	
vswprintf	`int vswprintf(wchar_t *wcs, size_t maxlen, const wchar_t *format, va_list args);`	{2}
	Wide-character equivalent of the vsnprintf() function.	

Configuration Vulnerabilities Unchecked user input is used to set configuration settings.

Table B.15.: Configuration Vulnerabilities

Name	Signature	Parameters
SetComputerName	`BOOL WINAPI SetComputerName(LPCTSTR lpComputerName);`	{1}
	Sets a new NetBIOS name for the local computer.	
SetComputerNameA	`BOOL WINAPI SetComputerNameA(LPCTSTR lpComputerName);`	{1}
	ANSI variant of the SetComputerName() function.	
SetComputerNameW	`BOOL WINAPI SetComputerNameW(LPCTSTR lpComputerName);`	{1}
	Wide-character variant of the SetComputerName() function.	
sethostid	`int sethostid(long hostid);`	{1}
	Sets a unique 32-bit identifier for the current machine.	

Path Traversal Vulnerabilities Unchecked user input is used to construct a pathname.

Table B.16.: Path Traversal Vulnerabilities

Name	Signature	Parameters
fopen	`FILE *fopen(const char *path, const char *mode);`	{1}
	Opens the file whose name is the string pointed to by path and associates a stream with it.	

Table B.16.: Path Traversal Vulnerabilities (cont'd)

Name	Signature	Parameters
fdopen	`FILE *fdopen(int fildes, const char *mode);` *Associates a stream with the existing file descriptor fildes.*	{1}
freopen	`FILE *freopen(const char *path, const char *mode, FILE *stream);` *Opens the file whose name is the string pointed to by path and associates the stream pointed to by stream with it.*	{1}
open	`int open(const char *pathname, int flags);` *Establishes the connection between a file and a file descriptor.*	{1}
wopen	*Wide-character equivalent of the open() function.*	{1}
CreateFile	`HANDLE WINAPI CreateFile(LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDisposition, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);` *Creates or opens a file or I/O device.*	{1}
CreateFileA	`HANDLE WINAPI CreateFileA(LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDisposition, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);` *Character equivalent of the CreateFile() function.*	{1}

Table B.16.: Path Traversal Vulnerabilities (cont'd)

Name	Signature	Parameters
CreateFileW	`HANDLE WINAPI CreateFileW(LPCTSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPSECURITY_ ATTRIBUTES lpSecurityAttributes, DWORD dwCreationDisposition, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);` *Wide-character equivalent of the CreateFile() function.*	{1}
remove	`int remove(const char * filename);` *Deletes the file whose name is specified in filename.*	{1}
rename	`int rename(const char *oldname, const char *newname);` *Changes the name of the file or directory specified by oldname to newname.*	{1}

Process Control Vulnerabilities Unchecked user input is used to determine which library is loaded.

Table B.17.: Process Control Vulnerabilities

Name	Signature	Parameters
LoadLibrary	`HMODULE WINAPI LoadLibrary(LPCTSTR lpFileName);` *Loads the specified module into the address space of the calling process.*	{1}
LoadLibraryA	`HMODULE WINAPI LoadLibraryA(LPCTSTR lpFileName);` *ANSI variant of the LoadLibrary() function.*	{1}
LoadLibraryW	`HMODULE WINAPI LoadLibraryW(LPCTSTR lpFileName);` *Wide-character variant of the LoadLibrary() function.*	{1}
LoadLibraryEx	`HMODULE WINAPI LoadLibraryEx(LPCTSTR lpFileName, HANDLE hFile, DWORD dwFlags);`	{1}

Table B.17.: Process Control Vulnerabilities (cont'd)

Name	Signature	Parameters
	Loads the specified module into the address space of the calling process.	
LoadLibraryExA	`HMODULE WINAPI LoadLibraryExA(LPCTSTR lpFileName, HANDLE hFile, DWORD dwFlags);` *ANSI variant of the LoadLibraryEx() function.*	{1}
LoadLibraryExW	`HMODULE WINAPI LoadLibraryExW(LPCTSTR lpFileName, HANDLE hFile, DWORD dwFlags);` *Wide-character variant of the LoadLibraryEx() function.*	{1}
AfxLoadLibrary	`HINSTANCE AFXAPI AfxLoadLibrary(LPCTSTR lpszModuleName);` *Maps a DLL module.*	{1}

Search Path Vulnerabilities Unchecked user input is used to construct the search path.

Table B.18.: Search Path Vulnerabilities

Name	Signature	Parameters
putenv	`int putenv(char *string);` *Adds or changes the value of environment variables.*	{1}
wputenv	`int wputenv(wchar *string);` *Wide-character equivalent of the putenv() function.*	{1}
setenv	`int setenv(const char *name, const char *value, int overwrite);` *Adds or changes the value of the environment variable with name name to value.*	{1}

C. Details for the Evaluation

This appendix contains additional information related to the evaluation in Chapter 8. The table lists all data flow variants of the SATE IV benchmark and names the associated data flow constructs.

C.1. Data Flow Variants

The SATE IV benchmark uses 68 different flow variants of which 40 are used in test cases of the weaknesses I selected for the evaluation. Variants 1-33 are intra-procedural whereas variants 41-68 involve at least two different functions.

Table C.1.: SATE IV Data Flow Variants

Variant	Description
1	Baseline
2	`if(1)` and `if(0)`
3	`if(5==5)` and `if(5!=5)`
4	`if(static_const_t)` and `if(static_const_f)`
5	`if(static_t)` and `if(static_f)`
6	`if(static_const_five==5)` and `if(static_const_five!=5)`
7	`if(static_five==5)` and `if(static_five!=5)`
8	`if(static_returns_t())` and `if(static_returns_f())`
9	`if(global_const_t)` and `if(global_const_f)`
10	`if(global_t)` and `if(global_f)`
11	`if(global_returns_t())` and `if(global_returns_f())`
12	`if(global_returns_t_or_f())`
13	`if(global_const_five==5)` and `if(global_const_five!=5)`
14	`if(global_five==5)` and `if(global_five!=5)`
15	`switch(6)`
16	`while(1)` and `while(0)`

Table C.1.: SATE IV Data Flow Variants (cont'd)

Variant	Description
17	For loops
18	Goto statements
19	Dead code after a return
31	Using a copy of data within the same function
32	Using two pointers to the same value within the same function
33	Use of a C++ reference to data within the same function
34	Use of a union containing two methods of accessing the same data (within the same function)
41	Data passed as an argument from one function to another in the same source file
42	Data returned from one function to another in the same source file
43	Data flows using a C++ reference from one function to another in the same source file
44	Data passed as an argument from one function to a function in the same source file called via a function pointer
45	Data passed as a static global variable from one function to another in the same source file
51	Data passed as an argument from one function to another in different source files
52	Data passed as an argument from one function to another to another in three different source files
53	Data passed as an argument from one function through two others to a fourth; all four functions are in different source files
54	Data passed as an argument from one function through three others to a fifth; all five functions are in different source files
61	Data returned from one function to another in different source files
62	Data flows using a C++ reference from one function to another in different source files
63	Pointer to data passed from one function to another in different source files
64	Void pointer to data passed from one function to another in different source files

Table C.1.: SATE IV Data Flow Variants (cont'd)

Variant	Description
65	Data passed as an argument from one function to a function in a different source file called via a function pointer
66	Data passed in an array from one function to another in different source files
67	Data passed in a struct from one function to another in different source files
68	Data passed as a global variable from one function to another in different source files

Bibliography

[ACD+76] R.P. Abbott, J. S. Chin, J. E. Donnelley, W. L. Konigsford, S. Takubo, and D. A. Webb. Security analysis and enhancements of computer operating systems. Final Report NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards, Washington, District of Columbia, United States, April 1976. 10–26 pp.

[BAMP81] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '81, pages 164–176. ACM, 1981.

[Bla05] Paul E. Black. Software assurance metrics and tool evaluation. *International Conference on Software Engineering Research and Practice (SERP)*, June 2005.

[Bla07] Paul E. Black. SAMATE and evaluating static analysis tools. *Ada User Journal*, 28(3):184–188, June 2007.

[Bla09] Paul E. Black. Static analyzers in software engineering. *CrossTalk*, 22(3):16–17, March 2009.

[BM09] David Basin and Ueli Maurer. Information security. Lecture notes, ETH Zurich, 2009.

[CKSY05] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *TACAS*, LNCS 3440, pages 570–574. Springer, 2005.

[CM04] Brian Chess and Gary McGraw. Static analysis for security. *IEEE Security and Privacy*, 2(6):76–79, November 2004.

[Cou81] Patrick Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Englewood Cliffs, New Jersey, United States, 1981.

[CSL08] Walter Chang, Brandon Streiff, and Calvin Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 39–50. ACM, October 2008.

[CW02] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, CCS '02, pages 235–244. ACM, 2002.

[EGHT94] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, pages 87–96. ACM, December 1994.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, pages 42–51, February 2002.

[FFA99] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 192–203. ACM, May 1999.

[FHJ+07] Ansgar Fehnker, Ralf Huuck, Patrick Jayet, Michel Lussenburg, and Felix Rauch. Model checking software at compile time. In *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 45–56. IEEE Computer Society, 2007.

[FHS10] Ansgar Fehnker, Ralf Huuck, and Sean Seefried. Counterexample guided path reduction for static program analysis. In *Concurrency, Compositionality, and Correctness*, volume 5930 of *LNCS*, pages 322–341. Springer, 2010.

[GL03] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS '03, pages 214–236. Springer, 2003.

[HCF05] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 303–311. IEEE Computer Society, 2005.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Proceedings of the 10th International Conference on Model Checking Software*, SPIN '03, pages 235–239. Springer-Verlag, 2003.

[HLV05] Michael Howard, David LeBlanc, and John Viega. *19 Deadly Sins of Software Security.* McGraw-Hill Osborne Media, July 2005.

[HLV09] Michael Howard, David LeBlanc, and John Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them.* McGraw-Hill Osborne Media, September 2009.

[JKK06] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security*, PLAS '06, pages 27–36. ACM, June 2006.

[Kau87] Christa Kaufmann. Univerum: Universal verification using model checking. *International Research Journal of Applied Life Sciences*, pages 9–29, September 1987.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206. ACM, 1973.

[Kra05] Kendra June Kratkiewicz. Evaluating static analysis tools for detecting buffer overflows in c code. Master's thesis, Harvard University, March 2005.

[Kri63] Saul Kripke. Semantical considerations on modal logic. In *Acta Philosophica Fennica*, volume 16, pages 83–94. Philosophical Society of Finland, 1963.

[LBMC94] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3): 211–254, September 1994.

[LE01] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, volume 10 of *SSYM '01*, pages 177–189. USENIX Association, 2001.

[LL03] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '03, pages 317–326. ACM, September 2003.

[LL05] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium*, volume 14. USENIX Association, 2005.

[MdR99] Todd C. Miller and Theo de Raadt. strlcpy and strlcat—consistent, safe, string copy and concatenation. In *Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference*. USENIX Association, June 1999.

[Mit11a] Mitre Corporation. Common vulnerabilities and exposures. Website, November 2011.
See: `http://cve.mitre.org/`.

[Mit11b] Mitre Corporation. Common weakness enumeration. Website, June 2011.
See: `http://cwe.mitre.org/`.

[Nat11a] National Institute of Standards and Technology. Common weakness enumeration. Website, June 2011.
See: `http://nvd.nist.gov/cwe.cfm`.

[Nat11b] National Institute of Standards and Technology. Standard reference dataset. Website, March 2011.
See: `http://samate.nist.gov/SRD/`.

[New00] Tim Newsham. Format string attacs. Online, September 2000.
See: `http://www.thenewsh.com/~newsham/format-string-attacks.pdf`.

[NIC11] NICTA. The Goanna Project. Website, November 2011.
See `http://www.nicta.com.au/research/projects/goanna/`.

[NNH05] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, second edition, 2005.

[SAB10] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 317–331. IEEE Computer Society, 2010.

[Sec11] Secure Software, Inc. Rough auditing tool for security (RATS). Website, November 2011.
See: `https://www.fortify.com/ssa-elements/threat-intelligence/rats.html`.

[SM11] SANS Institute and Mitre Corporation. CWE/SANS top 25 most dangerous software errors. Website, November 2011.
See: `http://www.sans.org/top25-software-errors/`.

[Soc90] IEEE Computer Society. IEEE standard glossary of software engineering terminology. IEEE Standard, 1990.
See: `http://standards.ieee.org/findstds/standard/610.12-1990.html`.

[Sot05] Alexander Ivanov Sotirov. Automatic vulnerability detection using static source code analysis. Master's thesis, University of Alabama, Tuscaloosa, Alabama, United States, 2005.

[STFW01] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, pages 201–216. USENIX Association, August 2001.

[TCM05] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. to be published in Proceedings of the NIST Workshop on Software Security Assurance Tools, Techniques, and Metrics (SSATTM), 2005.

[Tur37] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1937.

[US-00] US-CERT: United States Computer Emergency Readiness Team. CERT Advisory CA-2000-13: Two input validation problems in FTPD. Website, July 2000.
See `http://www.cert.org/advisories/CA-2000-13.html`.

[US-11] US-CERT: United States Computer Emergency Readiness Team. Microsoft Windows based applications may insecurely load dynamic libraries. Website, November 2011.
See: `http://www.kb.cert.org/vuls/id/707943`.

[VBKM00] John Viega, Jon-Thomas Bloch, Yoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *Proceedings of the 16th Annual Computer Security Applications Conference*, ACSAC '00, pages 257–267. IEEE Computer Society, December 2000.

[Vie05] John Viega. *The CLASP Application Security Process*. Secure Software, Inc., 2005.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of the 7th Networking and Distributed System Security Symposium*, pages 3–17. Internet Society (ISOC), February 2000.

[Whe11] David A. Wheeler. Flawfinder. Website, November 2011. See: `http://www.dwheeler.com/flawfinder/`.

[XCE03] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '03, pages 327–336. ACM, 2003.

[ZLL04] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '04, pages 97–106. ACM, 2004.