Vérimag - Distributed and Complex System Group

Universitatea Politehnica București

Detecting Software Vulnerabilities Static Taint Analysis

Dumitru CEARĂ

Supervisors

Marie-Laure POTET, *Ph.D, ENSIMAG, Grenoble INP*
Laurent MOUNIER, *Ph.D, Université Joseph Fourier*
Nicolae ȚĂPUȘ, *Ph.D, Universitatea Politehnica București*

September 2009

Contents

1 Introduction **7**
- 1.1 Overview . . . 7
- 1.2 Vulcain . . . 8
- 1.3 Static Analysis . . . 9
- 1.4 The context of the internship . . . 9

2 Taint Analysis **10**
- 2.1 State of the art - Taint Analysis . . . 10
  - 2.1.1 Dynamic Taint Analysis . . . 11
  - 2.1.2 Static Taint Analysis . . . 11
- 2.2 Taint analysis as a type system . . . 12
  - 2.2.1 Language Description . . . 13
  - 2.2.2 Definitions . . . 15
  - 2.2.3 Type Inference Rules . . . 16
  - 2.2.4 Taint Analysis Type System Soundness . . . 19
    - 2.2.4.1 Command sequence . . . 20
    - 2.2.4.2 Assign command . . . 20
    - 2.2.4.3 If command . . . 21
    - 2.2.4.4 Loop command . . . 23
    - 2.2.4.5 Function calls . . . 23
  - 2.2.5 Extending the type system . . . 24
- 2.3 Taint-based directed test generation . . . 25
  - 2.3.1 Execution path metrics . . . 26

3 Frama-C Platform **28**
- 3.1 CIL - Front-End for C . . . 28
  - 3.1.1 CIL Overview . . . 28
  - 3.1.2 CIL Abstract Syntax Tree . . . 30
- 3.2 Callgraphs . . . 32
  - 3.2.1 What is a callgraph? . . . 32
  - 3.2.2 CIL Callgraph Implementation . . . 33
  - 3.2.3 Example . . . 33
- 3.3 Frama-C Overview . . . 35
- 3.4 Memory Model . . . 35

3.5 Existing Plug-Ins . . . 37
3.5.1 Value Analysis Plug-In . . . 37
3.5.2 Dependency Analysis Plug-in . . . 37
3.5.3 Imperative Inputs/Outputs Plug-In . . . 38
3.5.4 Operational Inputs Plug-In . . . 38
3.5.5 Slicing Plug-In . . . 38
3.5.6 Semantic Callgraph Plug-In . . . 38
3.5.7 Jessie Plug-In . . . 38
3.6 Architecture . . . 39
3.6.1 Extended CIL Layer . . . 40
3.6.2 Kernel Layer . . . 40
3.6.3 Plug-Ins Layer . . . 40

4 STAC **42**
4.1 Functionalities . . . 42
4.1.1 Taint Analysis . . . 42
4.1.2 Metrics computation . . . 44
4.2 Architecture . . . 45
4.2.1 Configuration Module . . . 46
4.2.2 Taint Analysis Engine . . . 47
4.2.2.1 Taint Environment . . . 48
4.2.2.2 Taint Typing . . . 49
4.2.2.3 Instruction Processing . . . 49
4.2.2.4 Taint Analysis As Dataflow Analysis . . . 51
4.2.3 Results Module . . . 52
4.2.4 Slicing Interface . . . 53
4.2.5 Generic Metrics Module . . . 53
4.2.6 Metrics Implementation . . . 55
4.2.6.1 Min/Max Read Metrics . . . 55
4.2.6.2 Min/Max Taint Metrics . . . 55

5 Experimental Results **57**

6 Comparison With Related Work **60**
6.1 Taint Analysis . . . 60
6.1.1 Static: Parfait, Pixy . . . 61
6.1.2 Taint Analysis and test generation: BuzzFuzz . . . 62
6.2 Non Interference . . . 62

7 Conclusions **64**
7.1 Status Quo . . . 64
7.2 Future Work . . . 65

A Appendix **67**
A.1 STAC User Manual 67
A.2 Implementation Interfaces 68
A.2.1 Configuration Module 68
A.2.2 Taint Analysis Engine 69
A.2.3 Results Module 70
A.2.4 Generic Metrics Module 71
A.3 STAC SAMATE results 71

Acknowledgements

I would like to thank all the people without whom I would have never been able to achieve the current results during my internship period at the VÉRIMAG laboratory.

First, I would like to express my gratitude to my project supervisors, for their generous and immediate assistance, not only regarding the teaching activities, but also the moral support:

Marie-Laure Potet, professor at Ensimag, Grenoble INP and senior researcher at VÉRIMAG and **Laurent Mounier**, professor at Université Joseph Fourier and senior researcher at VÉRIMAG for the continuous help provided in developing both the theoretical notions for my project and the implementation. I would also like to thank them for guiding me into discovering the subtleties and elegance of formalising software verification techniques.

Nicolae ȚĂPUȘ, professor and senior researcher, part of the Computer Science Department at Politehnica University of Bucharest, for being my teacher in Computing Systems and providing me a strong knowledge base in order be able to build such an interesting project.

Last, but not least, I would like to thank my colleagues from the VÉRIMAG laboratory who encouraged me permanently and ensured a perfect working atmosphere in the office.

Abstract

The programming errors (known as bugs) obtained directly or indirectly, through incorrectly combining different elements, can lead to unexpected behaviour of the built programs. These situations are called software vulnerabilities. A security vulnerability is a vulnerability that can be exploited by an attacker in order to gain control over the system (usually remotely). However, in order for an attacker to exploit a vulnerability, the software bug must be controlled by user input.

In this paper we introduce a static analysis for performing taint analysis. This analysis is used to determine the parts of the program dependent on user input and can be used as a starting point in any bug finding tool. We will provide a theoretical basis for our analysis, by building a taint analysis type system and proving that it is sound, and also a tool that implements the theoretical notions as a plug-in for the Frama-C platform. The paper will also show how to use the results of the taint analysis in order to perform automatic test generation for the analyzed programs.

Chapter 1

Introduction

1.1 Overview

Developing a large software system is a rather difficult task. One of the many problems that can occur is represented by the presence of security vulnerabilities. A security vulnerability is a software bug that can be exploited by a malicious user through the supplied input, in order to gain control over the system on which the software runs. These kinds of vulnerabilities can produce a lot of damage including losing confidential data and great amounts of money. A simple security vulnerability is represented by the buffer overflow which could allow the attacker to remotely execute code on the machine running the software [1]. There are many famous examples of exploits (programs that exploit the security vulnerabilities) like the Microsoft JPEG GDI+ vulnerability [2], the Microsoft SQL Slammer Worm [3] that have caused a lot of damage to the attacked applications.

To prevent these situations, software verification tools perform checks both statically and dynamically in order to find the security vulnerabilities. Usually performing this checks is very time consuming and because security vulnerabilities are the user input dependent bugs, an approach to minimize the cost of software verification is using taint analysis to determine the points in the analyzed programs that can be influenced by user input.

In this paper we will present our approach for performing static taint analysis. The rest of this paper is organized as follows. Chapter 2 presents the state of the art in performing taint analysis and the theoretical basis for our approach: creating a type system for a simplified programming language in order to model our approach, providing the proof that our taint analysis type system is a sound type system and also how to extend it in order to cover the major functionalities of a real programming language. Also in chapter 2 we will describe the theoretical notions for using the results provided by our taint analysis in order to automatically generate tests for the verified applications.

Chapter 3 will be an overview of CIL [16] and Frama-C [24] which will be used as a front end for our implementation. We will present the functionalities provided by CIL and Frama-C and an architectural view in order to understand how our implementation will be integrated in the platform.

Chapter 4 will present the functionalities provided by STAC (our taint analysis implementation) and also a more accurate view on the architectural design of STAC and on the techniques and algorithms used.

The following chapters will show the experimental results obtained with our implementation and compare our work with related ones. We will also present the status of our implementation and the future work that needs to be done in order to obtain a fully functional and complete security verification tool.

1.2 Vulcain

The Vulcain project [31] is part of the MSTIC program from the Université Joseph Fourier. Software bugs can produce incorrect behaviours of the software systems also called vulnerabilities. If one such vulnerability can be used to intentionally change the behaviour of a software system, it becomes a system failure. The code that exploits a system failure for effectively influencing the system is an exploit. We can differentiate some steps in preventing a security attack as follows:

- discovering the vulnerability.
- computing the necessary steps in order for the vulnerability to be activated.
- developing a program in order to exploit the previously discovered vulnerability.

A great number of vulnerabilities in software systems used on a regular basis are discovered and exposed by different organizations in charge of security alerts (for instance CERT). Usually, after a vulnerability is discovered, patches for the vulnerable application are developed in order to protect it against possible attacks. This can be a rather difficult task both for the patch developers and also for the users that have to update their applications. The goal of the Vulcain project is to offer automatic techniques to help detecting security vulnerabilities in software systems in a classical environment. This is done by checking software applications for vulnerabilities and describing their activation context. Different techniques are used (both dynamic and static) for analyzing the software applications like black box testing, white box testing and also different static analyses.

1.3 Static Analysis

Let us take a look at the existing techniques and tools used for finding software vulnerabilities. The obvious advantage of using static analysis is the fact that it provides better code coverage unlike when using dynamic analysis. On the other hand there are some disadvantages like the fact that usually static analysis cannot be as precise as a dynamic one because it cannot access the runtime information for the analyzed program.

Bug checking software has been introduced many years ago and usually verification tools analyze the programs from many different points of view. A bug checking software performs a set of checks that are usually not performed by the compiler. These checks can be both syntactic and semantic checks. Because of the nature of static analysis, aproximations are performed which may lead to a number of false positives (reported vulnerabilities that are not really vulnerabilities). So one of the goals of bug checking tools is to minimize the number of false positives. Tools for finding vulnerabilities have been developed for a wide series of computer languages. One of the first tools was Lint which aimed at detecting bugs in C programs and ensuring additional typing rules for the analyzed programs. One of the main drawbacks of Lint was represented by the fact that the performed checks were mostly syntactic thus resulting in a very high rate of false positives. Nowadays the number of false positives provided by bug checking tools is minimized but still a percentage of the reported problems are false positives.

Static analysis can discover a large number of types of problems: deadlocks and race conditions, buffer overflows, command injection, cross-site scripting, format string vulnerabilities, SQL injection, secure information flow vulnerabilities, access control problems. A wide range of techniques is used in order to detect these kinds of problems: intra and inter procedural analyses, dataflow analysis, model checking.

1.4 The context of the internship

Our work has been developed during an internship between March and June 2009 at the Vérimag laboratory (the Distributed and Complex System group) which is one of the laboratories participating in the Vulcain project (along with the LIG laboratory).

Our work focuses on static analysis techniques that can be used in order to detect security vulnerabilities and their activation context. Specifically, we have developed a theoretical sound approach for performing taint analysis in the form of a specialized type system. Also we have also provided some ways of performing automatic test generation for the analyzed programs. An implementation for the theoretical notions is also available and will be described later in this paper.

Chapter 2

Taint Analysis

Building bug finding software can be done using several approaches. One of these approaches is using **Taint Analysis**. Taint analysis is also known as **User-Input Dependency Checking** [5] and appeared as a built-in feature in some computer languages, for instance *Perl*. The main idea behind taint analysis is that any variable that can be modified (directly or indirectly) by the user can become a security vulnerability (the variable becomes *tainted*). Through different operations the *taint* can be passed from variable to variable and when a *tainted* variable is used to execute dangerous commands a security breach may occur.

In this chapter we will present a short overview of some of the existing tools for performing taint analysis, our approach of reducing taint analysis to a type system and also a few methods of using the results provided by the taint analysis.

2.1 State of the art - Taint Analysis

According to the **Seven Pernicious Kingdoms** taxonomy [4] bugs can be classified into seven major groups. One of the groups defined here is the **Input validation and representation** class which covers bugs caused by metacharacters, alternate encodings, numeric representations, missing input validations which may have as results very important problems like *buffer overflows*, *cross-site scripting attacks*, *SQL injection*.

An alternate taxonomy is the one given by **OWASP** [21] where we can find the **Input Validation Vulnerability** category which consists of several types of vulnerabilities: *buffer overflow*, *format string vulnerabilities*, *improper data validation*, *string termination error*, *missing XML validation*.

Usually the technique used for detecting these kinds of bugs is taint analysis in its two possible forms: dynamic or static.

2.1.1 Dynamic Taint Analysis

The approach used in dynamic taint analysis is to label the data originating from untrusted sources (generally speaking this means user-dependent input) as *tainted*. The analysis keeps track of all the tainted data in the memory and when such data is used in a *dangerous* situation, a possible bug is detected. This approach offers the capabilities to detect most of the input validation vulnerabilities with a very low false positive rate. However there are some disadvantages when using dynamic taint analysis. The execution of the program is slower because of the necessary additional checks and the problems are detected only for the executions path that have been executed until now (not for all executable paths) which can lead to false negatives.

Some of the available tools for dynamic taint analysis are:

BitBlaze [22] a *binary analysis platform* which combines static analysis techniques with dynamic analysis techniques, mixing concrete and symbolic execution, system emulation and binary instrumentation. One of the dynamic techniques implemented by BitBlaze is taint analysis used for detecting overwrite attacks.

BuzzFuzz [6] an *automated white box fuzzing* tool which, unlike standard fuzzing tools, uses dynamic taint tracing to automatically locate regions of original input files that influence values used at key program attack points. New input files are generated by fuzzing the identified *taint regions*. Because it uses taint analysis to automatically discover and exploit information about the input file format, it is especially appropriate for testing programs that have complex input file formats.

TaintCheck [7] a dynamic taint analysis tool which uses binary runtime rewriting thus not needing the original source code or special compilation for the analyzed program. TaintCheck can be used to detect overwrite attacks which are largely used by most of the existing exploits.

Dytan [8] a generic customizable dynamic taint analysis framework which can run taint analysis on `x86` executables.

2.1.2 Static Taint Analysis

Static taint analysis is the technique used for detecting the overapproximation of the set of instructions that are influenced by user input. This set of tainted instructions is computed statically only by analyzing the sources of the program. The main advantage for static taint analysis is that it takes into account all the possible execution paths of the program. On the other hand the analysis may not be so accurate as the one performed dynamically because the static analyzer does not have access to the additional runtime information of the program. Some of the available tools for static taint analysis are:

Parfait [9] is a static multi-layered program analysis framework currently developed by **Sun Microsystems Laboratories**. It uses static taint analysis in its preprocessing stages. The approach used by Parfait is to reduce the taint analysis to a graph reachability problem [5].

CQual [10] is a tool that uses type qualifier annotations to determine if the user supplied data is used correctly. Partial relationships can be defined between these type qualifiers, thus adding the support for polymorphic type qualifiers. The advantage brought by CQual is that it doesn't need all the code to be annotated, but it uses a type inference engine to determine the corresponding types for all the symbols in the program (the engine is based on the use of type variables).

SPlint [23] is a tool that uses annotations in the form of stylized comments in order to provide the context for functions, variables, parameters and types. In effect it can be considered a rule based checker. It can be used for detecting a wide range of vulnerabilities, including buffer overflow and string format.

Safer [11] is a tool that combines taint analysis with control dependency analysis in order to detect control structures whose execution can be triggered by untrusted user input. It has been used to discover **DoS** (denial of service) attacks by detecting the user dependent loops in the execution of the program.

Pixy [12] is a tool that applies static taint analysis in order to detect SQL injection, cross-site scripting or command injection bugs in PHP scripts.

We have decided to take a different approach to our analysis and create a type system in order to perform static taint analysis. The main reason behind that is that type systems can be seen as logical systems in which we can reason about a wide variety of program properties. There are many advantages for using this kind of approach. On one hand, it gives us a formal specification that separates the taint analysis from the way it is implemented (the algorithms that will be presented in chapter 3). On the other hand, this approach allows us to prove easily the soundness of our analysis by proving that the chosen type system is safe. Similar approaches have already been performed, for instance in formalising the problem of ensuring secure information flow in programs also known as *non-interference* [14].

2.2 Taint analysis as a type system

In computer science a **type system** can be seen as a method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute [13]. A type system associates type information with each program symbol. By examining the flow for this type information, a type system attempts to prove that a program is type safe (this means that no *type errors* can occur). Each

type system determines what a *type erorr* is or, in a more general way, a type system guarantees that operations expecting a certain kind of type are not used with types for which that operation makes no sense.

Type systems can be classified in different ways according to different criteria:

Static Typing type systems have typing information associated to variables and not values. The typing information is computed and the type checking is performed at compile-time. Static typing can be seen as a way of verifying software properties. As we will present in the following sections, this is why we have decided to use a static typing type system in order to implement our taint analysis.

Dynamic Typing type systems perform most of the type checking operations during run-time. Opposed to static typing, in dynamic typing the type information is associated with values not variables. Dynamic typing can be more flexible than static typing because of the additional information supplied at run-time. The main advantage of dynamic typing is that special optimizations can be performed more easily with the additional run-time information. However the additional typing checks can result in run-time errors and can slow the execution of the program.

Specialized type systems are usually used for verifiying specific conditions and largely used in static program analysis. This is also the case for our taint analysis type system.

Explicit type declaration type systems require that all the symbols in a program must have explicitly associated type informations in the form of *type declarations*. This is the case for *C* or *Java* type systems.

Type inference type systems use special rules, *inference rules* in order to compute the type informations for the underlying program. An example of type system that uses type inference is the type system from OCaml where the underlying types of the symbols in the program are based on the contexts in which the symbols are used. For instance, for a function $f(x, y)$ which adds x and y the compiler can infer that x and y must be numbers (if the addition is only defined for numbers). The type system that we will suggest for taint analysis uses type inference, thus eliminating the need for taint type declarations.

2.2.1 Language Description

In order to build our type system we decided to use at first a simplified language and afterward extend it to cover all the functionalities in a real language (in this case C). For that we consider a subset of the C language as described below. For now we make the assumptions that no global variables exist in the program and

that the functions do not have side effects (the functions do not modify the values of the pointer parameters). Usually the source of tainted data is represented by the calls to library functions that require user input (for instance in C, calls to `scanf`, `fgets`). To simplify our type system we make the additional assumption that there is only one function that can return tainted data and name it *read*. Later we will show how to build an extension to our type system such that all these situations are covered. The following grammar describes our simplified language:

Functions : each function has a *signature* and a *body*. The *signature* consists of the name of the function and the list of formal parameters. The *body* of the function consists of the list of phrases which are executed when the function is called.

$$\begin{array}{lcl} func & : & signature\ p^+ \\ signature & : & id\ x^+ \end{array}$$

Phrases : a phrase can consist of a command, a call to another function or a call to the *read* function.

$$p \ : \ e \mid c \mid call\_func \mid call\_read$$

Expressions : an expression can represent accessing an identifier (which may represent a local variable or a formal parameter), accessing some kind of literal or of applying some operator to two expressions.

$$e \ : \ x \mid n \mid e\ op\ e$$

Function calls : a function call is represented by the *call* keyword followed by the function name and parameter bindings. The returned value is assigned to a lvalue expression.

$$call\_func \ : \ e \leftarrow call\ id\ (id \leftarrow e)^+$$

Commands : a command can consist of an assignment, a sequence of two commands, a conditional $if - then - else$ statement or a loop statement.

$$c \ : \ x \leftarrow e \mid c; c' \mid if\ e\ then\ c\ else\ c' \mid while\ e\ do\ c$$

The variable x represents the identifiers used for formal parameters and local variables. The variable n represents the literals present in the program and *op* stands for all the binary operators available in the language. The language description also makes the assumptions that all the functions return a value.

2.2.2 Definitions

We begin describing our taint analysis type systems by making some definitions. Each type system must have a *type domain* consisting of all the values that can be assigned as types to the variables of the program. Let $TD = \{T, U\}$ the type domain for our type system. The values T and U are associated to the *tainted* respectively *untainted* labels used when describing taint analysis. We also define a partial order on the type domain, " $\leq$ ", in the following way: $T \leq U$.

Our type system will have to handle programs that involve the presence of expressions so we have to define a binary operator $\oplus : TD \times TD \rightarrow TD$ as described below:

$$x \oplus y = \begin{cases} T, & \text{if } x = T \vee y = T \\ U, & \text{if } x = y = U \end{cases}$$

The $\oplus$ operator will be used to compute taint types for variables that depend on tuples of other variables. For instance for the following assignment, if the taint types for y, z, t are t_1, t_2, t_3: $x \leftarrow y + z + t$, the taint type for x will be computed as $t_1 \oplus t_2 \oplus t_3$.

Because of the way we have defined the $\oplus$ operator we can conclude that $\oplus$ has the following properties:

- Commutativity
  $\forall x, y \in TD \Rightarrow x \oplus y = y \oplus x$

- Associativity
  $\forall x, y, z \in TD \Rightarrow x \oplus y \oplus z = x \oplus (y \oplus z) = (x \oplus y) \oplus z$

For each function in the analyzed program we define three sets $Locals$ (the set of local variables for the function), $Formals$ (the set of formal parameters for the function) and $Vars = Locals \cup Formals$. In order to perform taint analysis we have to associate to each phrase from the function a taint environment. This is done by defining a mapping (Γ) between symbols and taint types in the following way:

$$\Gamma : Vars \rightarrow TD$$

Because our approach is an inter-procedural one, we have to keep track of the changes that are made to the taint types of the variables when a function is called. A function can be called from different contexts in the program (with different parameters) and its effects depend on the supplied parameters. For that we have

decided to build an environment for each function that can be reused in different calling contexts. In order for our environments to be reusable we have to define type variables G (called G for generic) with respect to a function environment Γ as the tuple of variables $(x_1, x_2, \ldots, x_n)$ on which the type variable depends. This denotes $G(x_1, x_2, \ldots, x_n) = \Gamma(x_1) \oplus \Gamma(x_2) \oplus \ldots \oplus \Gamma(x_n)$. Based on this definition and on the $\oplus$ operator definition and properties we add the folowing rules:

- $T \oplus x = x \oplus T = T$
- $U \oplus x = x \oplus U = x$

In order to simplify the type inference rules, we extend the $\oplus$ operator to Γ environments in this way:

$$\Gamma = \Gamma_1 \oplus \Gamma_2 \Leftrightarrow (\forall x \in Vars \Rightarrow \Gamma(x) = \Gamma_1(x) \oplus \Gamma_2(x))$$

Let $Funcs$ be the set of functions in the program. For each function, we associate an environment Γ in the following way: for each formal parameter x we associate the type variable $G(x)$. We also make the convention that for each function a new variable, ret, is created that will hold the type for the return value of the function and will never be read inside the function. The return value is a combination of the type variables corresponding to the formal parameters and values from TD. Let $\Gamma_{func} : Funcs \rightarrow (Vars \rightarrow TD)$, a mapping between functions and their associated environment. Initially, Γ_{func} contains the mappings for the library functions. The mappings for the user defined functions will be added when the type inference rules will be applied.

2.2.3 Type Inference Rules

As we stated earlier we associate to each phrase in the program a taint environment. Each phrase will change its associated environment and provide a new one for its successors. Before describing the inference rules we will take some time to describe the notations that will be used.

An inference rule is presented as a set of judgements that must be executed in order for the rule to be applied. Where are going to use two different judgement types:

commands : for a command c, the judgement $\Gamma \vdash c : \Gamma'$ means that after running the command c for the given environment Γ the resulting environment is Γ'.

expressions : for an expression e, the judgement $\Gamma \vdash e : \tau$ means that with respect to Γ the expression e has the taint type τ, where $\tau \in TD$.

We have to remind the reader that our language definition describes a language in which expressions do not have any side effects. For an easier understanding of the type system we also have to define the *Assigned* operator. When applying the *Assigned* operator to a command, it returns the set of variables (locals or formals), computed syntactically, that are assigned when executing the given command.

Now we can present the inference rules for our taint analysis type system:

Function definitions for each function definition a new environment is created and associated to the function. This environment will be updated every time a `return` phrase is analyzed. When the environment is created mappings are added for all the formal parameters to type variables and for all the local variables to U. The newly created environment is added to Γ_{func}.

$$\frac{\begin{array}{c}\Gamma = \{id_1 \rightarrow G(id_1), id_2 \rightarrow G(id_2), \ldots, id_n \rightarrow G(id_n), ret \rightarrow G(ret)\} \\ \Gamma \vdash body : \Gamma'\end{array}}{\Gamma_{func} \vdash f\ id_1, id_2, \ldots, id_n\ body : \Gamma_{func} \cup \{f \rightarrow \Gamma'\}}$$

Literals each literal value (string literal or numeric literal) is considered untainted.

$$\Gamma \vdash n : U$$

Variables each variable access with respect to a given Γ environment will have as taint type the associated type in Γ.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Expressions the type of an expression with respect to a given Γ environment will be obtained after applying the $\oplus$ operator between all the types of the subexpressions in which it consists.

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 \\ \Gamma \vdash e' : \tau_2\end{array}}{\Gamma \vdash e\ op\ e' : \tau_1 \oplus \tau_2}$$

Command sequence when executing a sequence of commands the environment is modified according to the two commands in the sequence.

$$\frac{\begin{array}{c}\Gamma \vdash c : \Gamma' \\ \Gamma' \vdash c' : \Gamma''\end{array}}{\Gamma \vdash c; c' : \Gamma''}$$

Assignments when an assignment is executed the mapping for the assigned variable in Γ is changed according to the taint type of the right hand side expression. In order to understand the inference rule for assignments we have to add a new notation $\Gamma_{[x:\tau]}$ which denotes the environment Γ in which we have changed the mapping for x to the type τ.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash x \leftarrow e : \Gamma_{[x:\tau]}}$$

If statements for each if statement the environment obtained after applying the inference rule will have to keep trace of all the assignments made on both branches but also on the taint type for the expression of the condition. This is required because of the presence of *implicit dependencies* [14]. Let's consider a simple example. Let's suppose that `x` is either 0 or 1 and consider:

```
if x = 1 then y := 1 else y := 0
```

Even if there is no explicit dependency between `y` and `x`, an implicit dependency exists because the value of `y` is dependent on the value of `x`. In order to prevent this situation the taint type for the condition is combined with the taint types in the environments from both branches of the if statement.

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \\ \Gamma \vdash c_1 : \Gamma_1 \\ \Gamma \vdash c_2 : \Gamma_2\end{array}}{\Gamma \vdash if\ e\ then\ c_1\ else\ c_2 : \Gamma_{[x:\tau \oplus \Gamma_1(x) \oplus \Gamma_2(x)]}, \forall x \in Assigned(c_1) \cup Assigned(c_2)}$$

Loop statements are dealt with in the following manner: the Γ environment for a loop statement is considered as an invariant for the loop. We add an environment subtyping rule in the following way: Γ' is a subtype of Γ ($\Gamma' \leq \Gamma$) if Γ' is obtained from Γ by replacing a variable type by its subtype. Based on that we can add the following rule stating that any environment obtained after running a command can be replaced by a subtype of it:

$$\frac{\Gamma \vdash c : \Gamma', \Gamma'' \leq \Gamma'}{\Gamma \vdash c : \Gamma''}$$

So the problem of infering the environment for a loop statement is reduced to finding an invariant environment for the command c and for which all the assigned variables in c are subtypes ($\leq$) of the condition expression type τ. This can be solved by finding the fixed point of the equation in Γ:

$$\Gamma \vdash c : \Gamma \wedge \Gamma \vdash e : \tau \wedge (\forall x \in Assigned(c) : \Gamma(x) \leq \tau).$$

and the type inference rule for loops becomes:

$$\frac{\begin{array}{c}\Gamma \vdash c\colon \Gamma \\ \Gamma \vdash e\colon \tau \\ \forall x \in \mathit{Assigned}(c)\colon \Gamma(x) \leq \tau\end{array}}{\Gamma \vdash \mathit{while}\ e\ \mathit{do}\ c\colon \Gamma}$$

Function calls the rule for function calls takes the previously computed result from Γ_{func} for the callee g and instantiates all the type variables in the return value to their current values in Γ.

$$\frac{\begin{array}{c}\Gamma \vdash e_1\colon \tau_1 \\ \vdots \\ \Gamma \vdash e_n\colon \tau_n \\ \Gamma_{func} \vdash g\colon \Gamma_g \\ \Gamma_g \vdash ret\colon \tau\end{array}}{\Gamma \vdash x \leftarrow \mathit{call}\ g(id_1 \leftarrow e_1, \ldots, id_n \leftarrow e_n)\colon \Gamma_{[x:\tau|_{G(id_1)\leftarrow\tau_1,\ldots,G(id_n)\leftarrow\tau_n}]}}$$

The notation $G(id_i) \leftarrow \tau_i$ represents the instantiation of the type variable $G(id_i)$ with the type τ_i.

2.2.4 Taint Analysis Type System Soundness

With respect to taint analysis, proving that a type system is safe can be done by proving the following property: *If the value of a tainted variable changes, then no untainted variable will change its value.* The property can be translated to:

$$\begin{array}{c}Ex(C, m_{0_T}, m_{0_U}) = (m_{1_T}, m_{1_U}) \wedge Ex(C, m'_{0_T}, m_{0_U}) = (m'_{1_T}, m'_{1_U}) \\ \Rightarrow m_{1_U} = m'_{1_U}\end{array}$$

where the function $Ex\colon Comm \times M \times M \rightarrow M \times M$ represents the execution of a command from the set of commands, $Comm$, on the memory. The memory is partitioned in two parts: the tainted memory and the untainted memory. Each part of the memory is represented as a mapping $M = Vars \rightarrow Vals$ from variables to values.

We will prove that each inference rule holds this property in order to show that our type system is sound. We will analyze only the rules that change the statement environments. We also have to create some additional definitions that will help us in creating our proof:

- $Val\colon exp \times M \times M \rightarrow Vals$ a function that returns the value of a given expression with respect to the values in the two memory partitions.
- $Exp\colon Comm \times Vars \rightarrow exp$ a function that returns the expression that assigns a given variable in the command passed as parameter.

2.2.4.1 Command sequence

In order to prove our property, we make the assumption that it holds for all types of commands and we try to prove that it will also hold for a sequence of commands. In this case, we can express our property in the following way:

P_1: $Ex(C_1, m_{0_T}, m_{0_U}) = (m_{1_T}, m_{1_U}) \wedge Ex(C_1, m'_{0_T}, m_{0_U}) = (m'_{1_T}, m'_{1_U})$
$\Rightarrow m_{1_U} = m'_{1_U}$

P_2: $Ex(C_2, m_{1_T}, m_{1_U}) = (m_{2_T}, m_{2_U}) \wedge Ex(C_2, m'_{1_T}, m_{1_U}) = (m'_{2_T}, m'_{2_U})$
$\Rightarrow m_{2_U} = m'_{2_U}$

and the conclusion:

C: $Ex(C_2, Ex(C_1, m_{0_T}, m_{0_U})) = (m_{2_T}, m_{2_U})$
$\wedge Ex(C_2, Ex(C_1, m'_{0_T}, m_{0_U})) = (m'_{2_T}, m'_{2_U})$
$\Rightarrow m_{2_U} = m'_{2_U}$

In order to prove the conclusion we rewrite its left part in the following way (by keeping in mind P_1):

$Ex(C_2, m_{1_T}, m_{1_U}) = (m_{2_T}, m_{2_U}) \wedge Ex(C_2, m'_{1_T}, m_{1_U}) = (m'_{2_T}, m'_{2_U})$ and now, according to P_2 we can conclude that $m_{2_U} = m'_{2_U}$.

In order for our proof to be complete we have to prove that our property holds for all the types of commands.

2.2.4.2 Assign command

For the assign command, our property can be written as shown here:

$Ex(x \leftarrow e, m_{T_0}, m_{U_0}) = (m_{T_1}, m_{U_1}) \wedge Ex(x \leftarrow e, m'_{T_0}, m_{U_0}) = (m'_{T_1}, m'_{U_1})$
$\Rightarrow m_{U_1} = m'_{U_1}$

With respect to the taint type of the left hand side variable and of the right hand side expression before the assignment is executed, there are four possible cases in our proof:

- **(1)** lvalue and right hand side expression tainted. We can compute the two outputs for Ex in the following way:

  $Ex(x \leftarrow e, m_{T_0}, m_{U_0}) = (m_{T_0} \cup \{(x, Val(e, m_{T_0}, m_{U_0}))\}, m_{U_0}) \Rightarrow m_{U_1} = m_{U_0}$

  and $Ex(x \leftarrow e, m'_{T_0}, m_{U_0}) = (m'_{T_0} \cup \{(x, Val(e, m'_{T_0}, m_{U_0}))\}, m_{U_0}) \Rightarrow m'_{U_1} = m_{U_0}$, thus concluding that $m_{U_1} = m'_{U_1}$.

- **(2)** lvalue untainted and right hand side expression tainted. We can compute the two outputs for Ex:

$Ex(x \leftarrow e, m_{T_0}, m_{U_0}) = (m_{T_0} \cup \{(x, Val(e, m_{T_0}, m_{U_0}))\}, m_{U_0} \setminus \{(x, \_)\}) \Rightarrow m_{U_1} = m_{U_0} \setminus \{(x, \_)\}$

and $Ex(x \leftarrow e, m'_{T_0}, m_{U_0}) = (m'_{T_0} \cup \{(x, Val(e, m'_{T_0}, m_{U_0}))\}, m_{U_0} \setminus \{(x, \_)\}) \Rightarrow m'_{U_1} = m_{U_0} \setminus \{(x, \_)\}$, thus concluding that $m_{U_1} = m'_{U_1}$.

- **(3)** lvalue and right hand side expression untainted. The two outputs for Ex are:

  $Ex(x \leftarrow e, m_{T_0}, m_{U_0}) = (m_{T_0}, m_{U_0} \cup \{(x, Val(e, m_{T_0}, m_{U_0}))\}) \Rightarrow m_{U_1} = m_{U_0} \cup \{(x, Val(e, m_{T_0}, m_{U_0}))\}$

  $Ex(x \leftarrow e, m'_{T_0}, m_{U_0}) = (m'_{T_0}, m_{U_0} \cup \{(x, Val(e, m'_{T_0}, m_{U_0}))\}) \Rightarrow m'_{U_1} = m_{U_0} \cup \{(x, Val(e, m'_{T_0}, m_{U_0}))\}$.

  So, proving our property is reduced at proving that: $Val(e, m_{T_0}, m_{U_0}) = Val(e, m'_{T_0}, m_{U_0})$. This can be done by contradiction. We make the assumption that the two values are different which means that τ, the taint type of e, is dependent on a tainted variable because only the tainted memory is changed: $\tau = \ldots \oplus T \oplus \ldots$, which contradicts our hypothesis that states $\tau = U$.

- **(4)** lvalue tainted and right hand side expression untainted. The outputs for Ex are:

  $Ex(x \leftarrow e, m_{T_0}, m_{U_0}) = (m_{T_0} \setminus \{(x, \_)\}, m_{U_0} \cup \{(x, Val(e, m_{T_0}, m_{U_0}))\}) \Rightarrow m_{U_1} = m_{U_0} \cup \{(x, Val(e, m_{T_0}, m_{U_0}))\}$

  $Ex(x \leftarrow e, m'_{T_0}, m_{U_0}) = (m'_{T_0} \setminus \{(x, \_)\}, m_{U_0} \cup \{(x, Val(e, m'_{T_0}, m_{U_0}))\}) \Rightarrow m'_{U_1} = m_{U_0} \cup \{(x, Val(e, m'_{T_0}, m_{U_0}))\}$.

  So, proving our property is reduced at proving that: $Val(e, m_{T_0}, m_{U_0}) = Val(e, m'_{T_0}, m_{U_0})$. The proof is done in a similar manner as for **(3)**.

2.2.4.3 If command

For the if command, the property can be written:

$$\begin{gathered} Ex(if\ e\ then\ c_1\ else\ c_2, m_{T_0}, m_{U_0}) = (m_{T_1}, m_{U_1}) \\ \wedge Ex(if\ e\ then\ c_1\ else\ c_2, m'_{T_0}, m_{U_0}) = (m'_{T_1}, m'_{U_1}) \\ \Rightarrow m_{U_1} = m'_{U_1} \end{gathered}$$

Before proving our property we will define a new operator $ValExpr : Comm \times Vars \times M \times M \rightarrow Vals$ where $ValExpr(c, v, m_1, m_2) = Val(Exp(c, v), m_1, m_2)$ which will compute the value for the expression that assigns a variable in a given command with respect to the two memory partitions.

There are two cases in our proof, according to the taint type of the condition expression:

- **(1)** expression tainted. In this case we have four subcases with respect to the value of the expression e when the memories (m_{T_0}, m_{U_0}) and (m'_{T_0}, m_{U_0}) are used:
  - **(1.1)** $Val(e, m_{T_0}, m_{U_0} = TRUE \wedge Val(e, m'_{T_0}, m_{U_0} = TRUE)$
  - **(1.2)** $Val(e, m_{T_0}, m_{U_0} = TRUE \wedge Val(e, m'_{T_0}, m_{U_0} = FALSE)$
  - **(1.3)** $Val(e, m_{T_0}, m_{U_0} = FALSE \wedge Val(e, m'_{T_0}, m_{U_0} = TRUE)$
  - **(1.4)** $Val(e, m_{T_0}, m_{U_0} = FALSE \wedge Val(e, m'_{T_0}, m_{U_0} = FALSE)$

  We will prove only subcase **(1.1)** because the proof for the others is similar. The two outputs for Ex in **(1.1)** are:

  $Ex(if\ e\ then\ c_1\ else\ c_2, m_{T_0}, m_{U_0}) = (m_{T_0} \cup \{(w_i, ValExp(c_1, w_i, m_{T_0}, m_{U_0}))\},$
  $m_{U_0} \setminus \{(w_i, \_)\}$ where $w_i \in Assigned(c_1) \cup Assigned(c_2)$
  $\Rightarrow m_{U_1} = m_{U_0} \setminus \{(w_i, \_)\}$

  $Ex(if\ e\ then\ c_1\ else\ c_2, m'_{T_0}, m_{U_0}) = (m'_{T_0} \cup \{(w_i, ValExp(c_1, w_i, m'_{T_0}, m_{U_0}))\},$
  $m_{U_0} \setminus \{(w_i, \_)\}$ where $w_i \in Assigned(c_1) \cup Assigned(c_2)$
  $\Rightarrow m'_{U_1} = m_{U_0} \setminus \{(w_i, \_)\}$

  thus concluding that $m_{U_1} = m'_{U_1}$.

- **(2)** expression untainted. If the expression is untainted there are only two subcases because the value of the expression is not dependent on the values in the tainted part of the memory m_{T_0} and m'_{T_0}:
  - **(2.1)** $Val(e, m_{T_0}, m_{U_0}) = Val(e, m'_{T_0}, m_{U_0}) = TRUE$
  - **(2.2)** $Val(e, m_{T_0}, m_{U_0}) = Val(e, m'_{T_0}, m_{U_0}) = FALSE$

  We will prove only subcase **(2.1)** because the proof for **(2.2)** can be done in a similar manner. In **(2.1)**, we obtain the following outputs for Ex:

  $Ex(if\ e\ then\ c_1\ else\ c_2, m_{T_0}, m_{U_0}) =$
  $(m_{T_0} \cup \{(w_{T_i}, ValExp(c_1, w_{T_i}, m_{T_0}, m_{U_0}))\} \setminus \{(w_{U_i}, \_)\},$
  $m_{U_0} \cup \{(w_{U_i}, ValExp(c_1, w_{U_i}, m_{T_0}, m_{U_0}))\} \setminus \{(w_{T_i}, \_)\})$
  where $w_{T_i}, w_{U_i} \in Assigned(c_1)$
  $\Rightarrow m_{U_1} = m_{U_0} \cup \{(w_{U_i}, ValExp(c_1, w_{U_i}, m_{T_0}, m_{U_0}))\} \setminus \{(w_{T_i}, \_)\})$

$$
\begin{aligned}
&Ex(if\ e\ then\ c_1\ else\ c_2, m'_{T_0}, m_{U_0}) = \\
&\quad (m'_{T_0} \cup \{(w_{T_i}, ValExp(c_1, w_{T_i}, m'_{T_0}, m_{U_0}))\} \setminus \{(w_{U_i}, \_)\}, \\
&\quad m_{U_0} \cup \{(w_{U_i}, ValExp(c_1, w_{U_i}, m'_{T_0}, m_{U_0}))\} \setminus \{(w_{T_i}, \_)\}) \\
&\text{where } w_{T_i}, w_{U_i} \in Assigned(c_1) \\
&\Rightarrow m_{U_1} = m_{U_0} \cup \{(w_{U_i}, ValExp(c_1, w_{U_i}, m'_{T_0}, m_{U_0}))\} \setminus \{(w_{T_i}, \_)\})
\end{aligned}
$$

So, proving the property is reduced at proving:

$$ValExp(c_1, w_{U_i}, m_{T_0}, m_{U_0}) = ValExp(c_1, w_{U_i}, m'_{T_0}, m_{U_0})$$

This is done by contradiction. We make the assumption that the two values are different which means that τ, the taint type of $Exp(c_1, w_{U_i})$, is dependent on a tainted variable because only the tainted memory is changed: $\tau = \ldots \oplus T \oplus \ldots$, which contradicts the hypothesis that states $\tau = U$.

2.2.4.4 Loop command

The execution of a loop command can be seen as a sequence of n if statements as shown next:

$$
\begin{aligned}
&while\ e\ do\ c\ = \\
&\qquad\qquad n \quad \begin{cases} if\ e\ then\ c\ else\ null \\ \vdots \\ if\ e\ then\ c\ else\ null \end{cases}
\end{aligned}
$$

There are two possibilities:

- **(1)** n is a finite number in which case the loop can be seen as a finite sequence of if commands. We proved earlier that our property holds for if statements and for sequences of commands (if it holds for the types of the commands in the sequence), so in this case by applying the two previous conclusions, the property holds.
- **(2)** n is a infinite number. In this case the property holds because in a real execution the statement never ends, thus we can say that the whole memory is tainted after the loop.

2.2.4.5 Function calls

The proof for function calls can be done in a similar manner to the one for assignments. Each function call can be seen as an assignment and the type instance computed for the return value mapping in the environment of the callee represents the assigned type expression.

By proving that all the type inference rules for statements in our language hold the property stated at the beginning of this section, we have shown that our type system is sound.

2.2.5 Extending the type system

The language chosen for exemplifying our type system is a subset of the C language. In order to cover all the main functionalities of C we have to extend our language and type system in order for them to support:

- **pointers as parameters** there is a very simple solution for adding support for pointers passed as parameters. We can easily consider functions that receive pointers as parameters as functions which return multiple values. For each formal parameter of a function which represents a pointer we add a new *return value* to the function. Whenever the function is called, in the calling environment for the statement, the mappings for the actual return value is changed, but also the ones for the pointer parameters. For instance for the following function and its calling context:

```
int foo(char* str) {str = read(); return 5;}
[...]
char* s = "foo";
int x = foo(s);
```

because passing a pointer as a parameter in C can be seen as a *call-by-reference*, when the `str` is tainted (when the call to `read` is made) inside `foo`, the actual parameter for the `foo` function becomes tainted. In order to avoid that we extend our language by adding the *pointer* attribute to function parameters (in order to track pointers as parameters) and the *ptr* operator to extract the value from the address pointed to by a pointer:

$$\begin{array}{lcl} func & : & signature\ p^+ \\ signature & : & id\ (pointer?\ x)^+ \end{array}$$

$$e \ : \ x \ | \ n \ | \ e\ op\ e \ | \ ptr\ e$$

and make the following translation:

$$\begin{array}{l} foo\ pointer\ str \\ \{ \\ ptr\ str \leftarrow 0 \\ return\ (5, str) \\ \} \\ [...] \\ s \leftarrow "foo" \\ (x, s) \leftarrow call\ foo(str \leftarrow s) \end{array}$$

which is an equivalent representation and also tracks `foo`'s side effects with respect to its pointer parameters.

- **global variables** the solution to adding support for global variables is based on the solution for pointers as parameters. At first all the global variables in the program are added to all the function definitions as pointer parameters. And, if the pointers as parameters problem is solved, the functions' side effects with respect to global variables problem is also solved. Let's look at a simple example:

```
int global;
int foo() {global = 20; return 5;}
[...]
int x = foo();
```

can be translated into:

$foo\ p\_global$
{
$p\_global \leftarrow 20$
$return\ (5, p\_global)$
}
[...]
$(x, global) \leftarrow call\ foo()$

thus adding support for global variables.

Even with the extensions supplied here, our type system still does not cover all the functionalities of C. We still have not found a solution for the problem of aliasing which can be done in different ways in C (by using pointers in the program, by using variable addresses). The proof offered for the soundness of our type system only deals with the subset of C presented above.

2.3 Taint-based directed test generation

One of the main goals of our approach is to find bugs in the programs that we analyze. An important aspect of software verification is software testing. This usually can be a rather difficult thing to do especially when the analyzed programs require important amounts of user supplied data. This is why it would be of great help to automatically generate tests for the analyzed program. But, in most of the cases, this is not an easy thing to do because of the infinite number of possible execution paths in a program, resulting in a very complicated test case scenario. We offer a possible solution for generating smaller test cases by using the results offered to us by the taint analysis previously discussed.

Static taint analysis may produce false positives (leading to false vulnerabilities), and a way to remove/confirm the results obtained is to perform a dynamic analysis

afterward. This dynamic analysis can be made by replaying execution sequences leading to the potential vulnerabilities, which is a kind of test execution (where the test objective is to activate a vulnerabilty). These test execution could also be improved by introducing some fuzzing mechanisms as shown in [6]. Because of the infinite number of possible execution paths in a program, we need some selection criteria, and we choose to define them as metrics on the execution paths. These metrics could be defined with several objectives: in order to ease the test execution, to increase the chance of activating the vulnerability. In this section we will present our notion of execution path metrics and how to link the metrics computation to the taint analysis in order to generate better test cases.

2.3.1 Execution path metrics

Because every program can contain conditional statements (for instance *if-then-else* statements in C) or loop statements, different executions of the same program can lead to different behaviours for the executed program. For this we define the notion of *execution path* which consists of a set of ordered program statements which represent one possible behaviour of the program with respect to the supplied conditions for the conditional and loop statements.

In order for us to perform an accurate testing of our programs, in the ideal case, all the execution paths for a given program should be tested. But in reality this can be a very difficult (maybe impossible) task as we said earlier because of the infinite number of possible execution paths. In order to improve our testing process we would like to test only the paths that have a *high risk of vulnerability*. If we want to achieve that, we have to somehow differentiate between different execution paths. This is why we have chosen to associate to each execution path a specific cost based on some custom heuristic. Now we can choose the paths with the highest cost according to our heuristic and generate tests that will force the program to execute the specified path.

After selecting a path an approach towards test generation could be to find the conditions that enable the selected path and combine them in a conjunction which will represent the *activability condition*. We have to determine the possible values for the variables in order to activate the path.

But, then again, we still have the problem of choosing the best heuristic for our testing process. Our approach is to use the taint analysis results in order to select the path with the highest probability of producing a bug and in the same time obtain smaller test-cases. One example of a simple way of using the taint analysis results for our metrics computation could be to determine the path that requires the minimum amount of user input and gets to a critical section in the program (for instance a system function call that requires that its parameters are untainted).

Similar approaches have been implemented mostly for dynamic test generation using taint analysis information [15]: taint is tracked dynamically when the program is ran with a valid input and based on the taint values, critical sections (where vulnerabilities can occur) are determined. For each critical section, new inputs are generated with the parts that affect values in the critical section changed, and the program is run again to check for other errors.

Chapter 3

Frama-C Platform

In this chapter we will present Frama-C, platform we used as a front-end for our implementation. We will cover both the end user's view of the platform and the developer's view and also the architecture of Frama-C and we will have a closer look at CIL (**C** **I**ntermediate **L**anguage) [16] which makes the translation from C code to its abstract interpretation. We will also speak about callgraphs and their implementation in CIL.

3.1 CIL - Front-End for C

In order to be able to present Frama-C, first of all we have to take a closer look at CIL which is the front-end used by Frama-C for parsing C source files.

3.1.1 CIL Overview

CIL can be seen as a high-level representation of C which allows an easier analysis on the input programs. The main functionality of CIL is building an easy to use intermediate representation of the source file by performing transformations on the input file in order to obtain an **AST**(*abstract syntax tree*) which uses a few core constructs and a very clean semantics.

In essence, CIL is a highly-structured, clean subset of C. As we said earlier, a number of simplifications are done on the analyzed code. For instance:

- all looping constructs are reduced to a single form (a `while(1)` construct followed by a `if` instruction for the loop condition).
- for all the functions in the program an explicit return statement is added.
- function arguments with array types are transformed into pointers
- declarations for unused entities are removed.

- nested structure definitions are pulled apart. All structure definitions can be found by simply looking at the global definitions.
- all type definitions from inner scopes are moved to the global scope.
- prototypes are added for the functions that are called before being defined.
- local variables in inner scopes are moved to the function scope (with the appropriate renaming in order to avoid name collisions). This makes it easy to operate on all local variables in a function.
- one of the most important simplification is that expressions that contain side-effects are separated into statements.
- all types are computed and explicit casts are inserted for all implicit casts that a compiler must handle.

All these transformations structure the program in a manner more suitable to rapid analysis and transformation. For a better understanding, we will now present a small example on how loops are translated by CIL. For the following C code:

```
[...]
int i = 10;
int x = 2;
int n = 0;
while (i < 200) {
  x *= 2;
  ++ i;
}
for (i = 0; i < x; ++ i)
  n += x;
[...]
```

the translation made by CIL produces the following equivalent code:

```
[...]
int i = 10;
int x = 2;
int n = 0;
while (1) {
  if (!(i < 200))
    break;
  x *= 2;
  ++ i;
}
i = 0;
```

```
while (1) {
  if (!(i < x))
    break;
  n += x;
  ++ i;
}
[...]
```

so as one can see all loop statements have been changed into `while(1)` and the condition of the loop written as an `if-break` statement at the beginning of the loop.

The main advantage of running CIL on a source file is that it organizes the imperative features of C into expressions, instructions and statements based on the presence or absence of side-effects and control-flow. Every statement will be annotated with successor and predecessor information and will consist of one or more instructions (without control-flow effects), thus providing a program representation that can be used with routines that require an AST (for instance, type based analyses), as well with routines that require a control flow graph (for instance, dataflow analyses).

3.1.2 CIL Abstract Syntax Tree

The top level representation of a CIL source file is represented by the AST. Its main content is the list of global declarations and definitions. There are multiple kinds of **globals**:

- Typedefs: nodes in the AST which associate names to previously defined types.
- Compound Type Definitions: represent `struct` or `union` definitions with their associated fields.
- Compound Type Declarations: represent forward declarations of compound types.
- Enumeration Definitions: used for defining an enumeration and its fields.
- Enumeration Declarations: represent forward declarations of enumerations.
- Variable Definitions: consist of variable definitions and their initializer. The type of the variable is also attached to the node.
- Function Definitions: maintain the return type and the name of the function (as a variable), the list of local variables and their associated type, the list of formal parameters and their types and also the body of the function in the form of a list of statements.

CIL provides several functions that can be useful for manipulating globals like: iterating through globals, creating globals, modifying global definitions.

It is interesting to see how CIL manipulates the types used in the analyzed source. There are several kinds of **types**, each of them with different meanings and corresponding to a C type:

- Void: is mapped on the `void` type in C (all the functions that return `void` or all `void` variables will have this type associated).
- Int: is used to describe all the integral types existing in C. It has subtypes corresponding to signed/unsigned characters, booleans, signed/unsigned integers of different sizes.
- Float: is used to describe all the floating-point types from C. It has subtypes corresponding to `float`, `double` and `long double`.
- Ptr: corresponds to all pointer types. It also holds information about the type of the location to which the variable points.
- Array: indicates an array type by specifying its base type and length.
- Named: indicates the use of a renamed type (this can happen when using `typedef`s).
- Compound Types And Enumerations: indicate the use of structure or enumeration types

For better manipulation of types, CIL provides functionalities for: creating new types, comparing types, querying for type kinds, scanning types.

As we said earlier CIL organizes C statements into two major groups: CIL statements (which contain control-flow information) and CIL instructions (which correspond to C statements that do not affect the control flow of the program). Let's have a closer look at how CIL maintains this structure. We will first start with the CIL **statements**, that can be of several kinds:

- Return statements: correspond to (explicit or CIL added) `return` statements in the original source code. These are leaves in the control flow graph. These nodes can also hold the return expressions for the functions that return a non-void value.
- Goto statements: correspond to (explicit or CIL added) `goto` statements in the original source code. The destination statement is also stored in the AST node.
- Break and Continue statements: contain control-flow information. They break or continue to the end of the nearest enclosing loop.

- If statements: hold information about the conditional expression of the if statement and also control information in the form of the two successors (the "then" and "else" branches).
- Swith statements: are similar to the if statements, the only difference being in the fact that they can have more than two successors (according to the number of cases).
- Loop statements: correspond to all kinds of C loops in the original code. As stated earlier, all the loops are translated to `while(1)` loops.
- Instruction statements: hold CIL instructions that will not affect the control-flow.

There are two major types of CIL **instructions**:

- Assignments: which provide the variable being assigned (the lvalue) and the expression being assigned to it.
- Function Calls: which provide the expression that holds the function call and the list of expressions passed as actual parameters to the called function.

As one can see the abstract representation obtained after running CIL is a very compact one, using simple constructs which make it easy to run complex analyses on it. This is a great advantage especially given the fact that parsing C itself is usuallly not a trivial task.

3.2 Callgraphs

In this section we will describe the *callgraph* notion, how a callgraph can be used and the callgraph implementation from CIL. We will also provide a small example for better understanding of the notions.

3.2.1 What is a callgraph?

A **callgraph** (also known as a call multigraph) is a directed graph that represents calling relationships between subroutines in a computer program. Specifically, each node represents a function and each edge $f \rightarrow g$ indicates that function f calls function g. If the underlying program contains recursive functions (or mutually recursive functions) computing the callgraph will result in obtaining a directed graph with a cycle for each recursive function (or mutually recursive function group).

Callgraphs are a basic program analysis result that can be used as a basis for further analyses, like tracking the flow of values between procedures. Callgraphs are usually very useful when performing interprocedural optimizations or analyses.

A callgraph can be computed in two ways: dynamically or statically. A **dynamic callgraph** can be seen as a record of an execution of the program and, while being very accurate, it only describes one run of the program. On the other hand, a **static callgraph** is a call graph which represents every possible execution of the program. Computing the exact static callgraph is an undecidable problem so static callgraph algorithms are overapproximations.

With respect to the degree of precision of the callgraph there are two main groups: context-sensitive callgraphs (which have for each function a separate node for each call stack that the function can be activated with) and context-insensitive callgraphs (there is only one node for each function in the program).

3.2.2 CIL Callgraph Implementation

Besides the abstract program representation, CIL also provides different tools in order to simplify applying different analyses. CIL provides its own implementation for callgraph computation. CIL computes a static callgraph in a context-insensitive manner.

The results of the callgraph computation are provided as a mapping between function name and specialized callgraph nodes. As we said earlier, when computing context-insensitive callgraphs, each function has only one associated node. Each node has an unique identifier and custom information about the function it represents (the most important is a link to the variable naming the function) and also informations about its predecessors (the callers of the function) and successors (the callees of the function). The informations about successors and predecessors are stored as mappings between node identifiers and nodes.

CIL also supplies functionalities for manipulating callgraph nodes, extracting function information and used in conjunction with external libraries (like ocamlgraph [30]) allows the programmer to easily implement interprocedural analyses in an easy manner.

3.2.3 Example

For a better understanding on how context-insensitive callgraphs are built we supply a simple C example (that will also be reused in the following chapters):

```
int main()
{
  int x, b1, b2, y;
  scanf("%d", &x);
  b1 = even(x);
  b2 = odd(10);
  y = compute(x);
  return 0;
}
```

```
int even(int x)
{
  if (x == 0)
    return 1;
  else
    return odd(x - 1);
}
```

```
int odd(int x)
{
  if (x == 1)
    return 0;
  else
    return even(x - 1)
}
```

```
int compute(int x)
{
int sum, i;
  if (x == 2)
    sum = taint();
  else
    sum = 0;
  for(i = 0; i < x; ++ i)
    sum += i;
  return sum;
}
```

The example shows an implementation of `even` and `odd` functions by using mutual recursivity. Also the `compute` function is provided in order to see how the computation works when non recursive functions are used. A graphical representation of the callgraph is presented in figure 3.1. In graph theory a directed graph is called

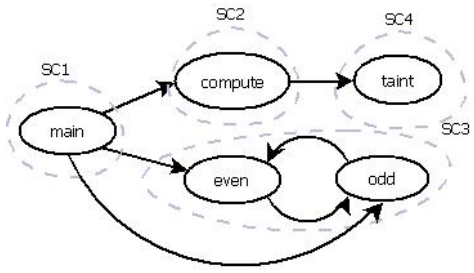


Figure 3.1: Simple callgraph example

strongly connected if there is a path from each vertex in the graph to every other vertex. The *strongly connected components* of a directed graph are its maximal strongly connected subgraphs. In the case of a callgraph each strongly connected component can correspond to two different situations:

- a single node when the function represented by the node is not a part of a mutually recursive functions group. The function can either be recursive or not recursive.

- multiple nodes in which case the functions represented by the nodes are mutually recursive.

As we can see in this simple example, the obtained callgraph consists of four strongly connected components: `SC1` and `SC2` for the non recursive functions `main` and `compute`, `SC3` consisting of two nodes corresponding to the mutual recursivity between the two functions `even` and `odd` and `SC4` for the `taint` function.

3.3 Frama-C Overview

Frama-C [24] is an extensible platform for source-code analysis of C programs. It can also be seen as a collaborative framework for developing static analyzers. The collaborative approach allows analyzers to use the results already computed by other analyzers in the framework. Frama-C is an extensible framework because of its analyzer implementation. Each analysis is implemented as a plug-in for the framework. Its modular approach makes it easy to use.

The goal of Frama-C, through the included static plug-ins, is to become closer to the heuristic bug (problems that may occur at runtime) finding tools but in the same time to maintain its correctness and not allow false negatives (that is not to miss any bugs that may occur). The analysis performed by Frama-C is a very flexible one because of the possible interaction with the user through functional specifications. With this approach Frama-C helps the user to ensure the correctness of the programs by detecting the existence of bugs and by proving that the program respects the provided functional specifications. Frama-C is written in OCaml [25] and uses as a front-end CIL [16], which is a high level representation of C programs and which provides means of analyzing and transforming C programs.

3.4 Memory Model

Frama-C comes with a wide variety of ready to use analyzers (implemented as plug-ins) but the most used plug-in is the *Value Analysis* plug-in. Most of the other analyzers use the results provided by the *Value Analysis* plug-in. So it is very important to know the abstractions used by the plug-in to implement the memory model for the analyzed program.

The **memory model** [26] uses the notion of *base address*. Each variable in the program (global or local) defines **one and only one** base address. For the following definitions:

```
int x;
int y[12][12][12];
int *z;
```

three base addresses are defined, for `x`, `y`, `z`. The sub-arrays composing `y` share the same base address. At a given time of execution we can refer to the base address corresponding to the memory location pointed to by `z`.

The most important assumption made by the memory model is the **bases separation** hypothesis: **It is possible to pass from one address to another through the addition of an offset, if and only if the two addresses share the same base address.**

An **address** is defined as a pair of a base address and an offset. The offset is an integer value used to index different parts of the *object* refered to by the base address. For instance the addresses of the different sub-arrays of `y` are expressed as different offsets with respect to the same base address (the base address of `y`).

This is an approximation made by the analysis which restrics the set of C programs that can be analyzed correctly. It is true that in C it is always possible to compute an offset such that adding it to a base address we can obtain an address corresponding to a different *object*. However, based on syntactic checks, the *Value Analysis* plug-in generates alarms when the analyzed code could be outside the analyzable subset of C programs. An example of base changing is the following:

```
int x, y;
int *p = &y;

void main (int c)
{
  if (c) {
    x = 2;
  } else {
    while (p != &x) p ++;
    *p = 3;
  }
}
```

An alarm is generated by the value analysis plug-in because of the pointer comparison performed in the condition for the `while` statement. In this case using the previously specified memory model will produce incorrect results because an analyzer will never be able to detect the base change performed on `p`. The variables `x` and `y` have different base addresses. Because of the assignment, the pointer `p` will have the same base address as `y`. According to the **bases separation** hypothesis whichever offset we add to the base address of `y` we can never obtain an address which will have as a base address the base addres of `x`. When running the code showed above on a real machine, it is possible for the value of `p`, when executing the `while` statement, to become the address of `x` and so to actually modify `x`.

3.5 Existing Plug-Ins

The Frama-C framework comes with a set of already built analyzers and their results can be used very easily by new plug-ins. Some of the most important are the following:

3.5.1 Value Analysis Plug-In

The Value Analysis plug-in is the most commonly used analysis by the other computations. It is automatically activated when an option that depends on *Value Analysis* is requested (this is actually what happens for every analysis required as a dependency). The analysis computes an over-approximation of the set of values possibly taken by a variable at a designated point for all possible execution paths. The possible values for the variable are represented as *variation domains*. The variation domains can be *set of integers* (as enumerations, intervals with or without periodicity information), *floating-point values or intervals*, *sets of addresses*. The main causes for approximations are introduced by calls to library functions, fusions of values with different alignments, reading locations that have not been written in the same operation. The analysis works in the following way:

- the global variables are initialized to their specified values (or 0 if none specified)
- the main function formal parameters are initialized to non-deterministic values according to their type, non-aliasing locations for pointers and fixed depth for chain-like structures
- these values are then propagated throughout the program; for the encountered loops in the program successive approximations are applied to memory locations (the technique is called *widening*). The loop analysis can be customized by using *loop unrolling* or by specifying bounds (*widen hints*).

Some of the limitations of the analysis are: missing the support for recursive function calls, it cannot be used to prove that a function terminates, the set of provided values is computed making the assumption that all the alarms emitted during the analysis have been verified by the user.

3.5.2 Dependency Analysis Plug-in

The Dependency Analysis plug-in computes the dependencies for modified values specifying for each modified value the input variables on which it depends. The analysis doesn't track non terminating areas as these sections will never influence the changes made afterwards.

3.5.3 Imperative Inputs/Outputs Plug-In

The Imperative Inputs plug-in computes for a specific function an over-approximation of the locations that may be read during its execution.

The Imperative Outputs plug-in computes for a specific function an over-approximation of the locations that may be written during its execution.

3.5.4 Operational Inputs Plug-In

The Operational Inputs plug-in determines the set of memory locations that are read before being written. It can be very useful when detecting which variables must be initialized in order to be able to correctly execute a specific function.

3.5.5 Slicing Plug-In

The Slicing plug-in transforms the program obtaining a smaller compilable part of the program with respect to different criteria:

- **function returns**: each return in the original program is made also in the slice and the returned value is the same.
- **function calls**: each call to a given function is also made in the computed slice and all the effects of the function are preserved.
- **lvalue write access**: each write access to a specific variable is preserved in the computed slice.
- **lvalue read access**: each read access to a specific variable is preserved in the computed slice.
- **pragmas**: specific instructions can be kept in the computed slice using code annotations (*pragmas*) in the original program.

3.5.6 Semantic Callgraph Plug-In

The Semantic Callgraph plug-in is based on the value analysis plug-in and computes a semantic callgraph of the program based on the possible values for the variables computed by the value analysis plug-in. It represents a finer approximation than a classical callgraph.

3.5.7 Jessie Plug-In

The Jessie plug-in uses preconditions, postconditions and assertions added to the analyzed program using ACSL [28] to perform deductive verifications. It uses Why

[29] to generate verification conditions that are afterward submitted to external theorem provers such as Simplify, Alt-Ergo, Z3, Yices, CVC3.

3.6 Architecture

The architecture of Frama-C [27] is structured on three levels as shown in figure 3.2:

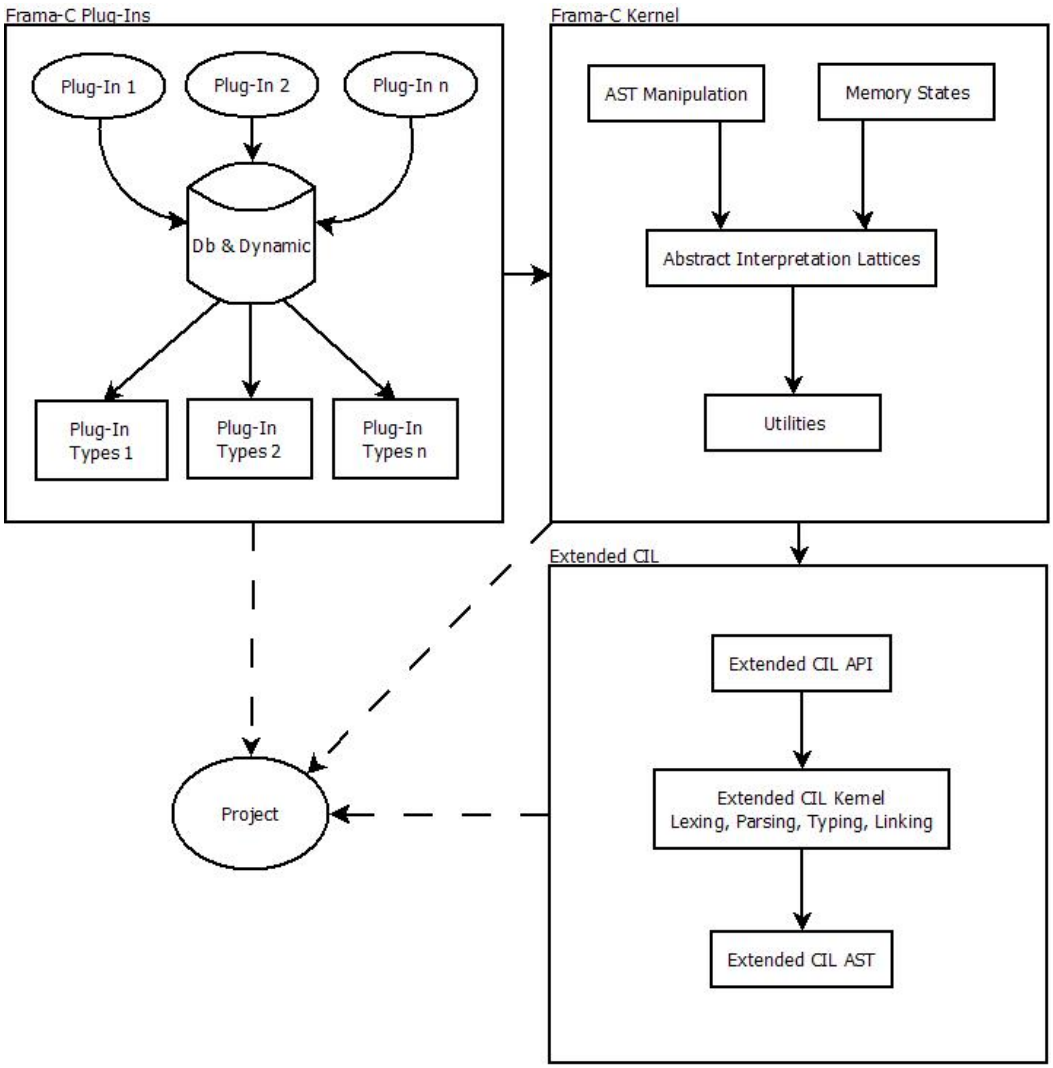


Figure 3.2: Developer View - Software Architecture

In the following sections we will present a short overview of each major layer from the architecture of Frama-C.

3.6.1 Extended CIL Layer

The Extended CIL Layer is built on top of CIL. As we said earlier in section 3.1, CIL consists of the high-level representation of the analyzed program (the *abstract syntax tree*) and a set of tools for analysis and transformations. The program representation and tools are available to the programmer through the API supplied by Frama-C.

The CIL [16] front-end is extended by Frama-C by adding types and operations to handle ACSL [28] programs.

3.6.2 Kernel Layer

The Kernel Layer is built on top of the Extended Cil Layer and groups together different kinds of modules:

Utilities is represented by the module `Extlib` and contains useful operations and data structures (not necessarily specific to program analysis) like specialized containers, sets and maps.

Lattices is represented by the module `Abstract_interp` and provides generic lattices useful for abstract interpretation.

Memory States is represented by the module `Locations`. This is actually the implementation of the Frama-C memory model. It also contains specialized operations and data structures for handling memory locations.

AST Manipulation is represented by the module `Visitor` and provides generic implementations vor visiting, copying or modifying in place the AST. As we saw earlier, the CIL layer offers a visitor class, `Cil.cilVisitor`, that performs a traversal on the AST and for each type of node found in the AST calls its specific method. So, performing transformations over an AST is reduced to inheriting the `Cil.cilVisitor` class and overwriting the specific methods. However the `Cil.cilVisitor` isn't aware of Frama-C and its current state. This is why there exists another visitor implemented by Frama-C, `Visitor.generic_frama_c_visitor`, which adds support for multiple projects management (transparent for the programmer). It takes an additional argument in its constructor (the project in which the AST should be put in). It also adds additional suport for copy-visitors by providing functions for changing the mapping from the original to copy nodes.

3.6.3 Plug-Ins Layer

The Plug-ins Layer contains two modules for handling plug-in registrations. A plug-in can be static (if the plug-in is statically compiled into Frama-C) or dynamic (if it is

loaded dynamically when Frama-C runs). This layer also handles all the interaction between the used plug-ins. Each static plug-in exports its functionalities using the `Db` module. All the dynamic plug-ins export their functionalities using the functions defined in the `Dynamic` module. Static plug-ins can also export custom types and operations to handle those custom types.

Chapter 4

STAC

In this chapter we will present our implementation of taint analysis, **STAC** (Static Taint Analysis for C), which is built as a plug-in for Frama-C. The chapter will consist of a description of the functionalities of STAC and a more detailed section about its architecture.

4.1 Functionalities

The main goal of **STAC** is to perform **interprocedural taint analysis** on C source files. The implementation of taint analysis in STAC follows the theoretical concepts presented in chapter 2. An overapproximation of the set of instructions influenced by user input is computed. The results obtained after performing taint analysis are used for computing metrics for execution paths most suitable to be tested in order to detect possible vulnerabilities. The results can also be used for computing slices of the original program with respect to possible vulnerable points in the program.

4.1.1 Taint Analysis

As this is the main functionality of the implementation, it offers multiple ways of obtaining taint analysis related information. All the other functionalities implemented by STAC are dependent on the taint analysis computation. STAC takes each function from the input file and computes a taint environment for each of the statements in the function.

One way of interacting with the results is by using the environments computed for the return statements of each function. Using the sample code presented in section 3.2.3 we get the following results (the results are shown as outputed by a pretty printer, but can also be used programatically by other modules in the implementation):

```
Env for function main:          Env for function even:
======================          ======================
x = Tainted                     x = Gamma(x)
b1 = Tainted                    tmp = Gamma(x)
b2 = Untainted                  even = Gamma(x)
y = Tainted
main = Untainted

Env for function odd:           Env for function compute:
======================          ======================
x = Gamma(x)                    x = Gamma(x)
tmp = Gamma(x)                  sum = Tainted
odd = Gamma(x)                  compute = Tainted
```

Let's take some time to interpret the results:

Each environment printed for a function represents the Γ environment computed for the return statement of the function. The values `Tainted`, `Untainted` correspond to the TD values T and U described in our type system in chapter 2 and the value `Gamma(x)` corresponds to a taint type variable $G(x)$. For each function a special mapping is added for the return value of the function. The mapping key has the name of the function and its value represents the taint type for the returned value.

Let's start with the `compute` function. The return taint type for this function is tainted because of the presence of the conditional statement. On one of the branches the `sum` variable is assigned a tainted value, thus tainting it from that point forward.
Another interesting situation appears for the mutual recursive functions `odd` and `even`. The environment for the `odd` function is computed using the values from the `even` environment, but in the same time the environment for the `even` function is computed using the values from the `odd` environment. Because of the way in which each of these functions uses its `x` parameter, the return value for both of the function will be a taint type variable dependent on the taint type for the formal parameter `x`. The effect of these environment values can be seen when analyzing the environment for the `main` function. When the `even` function is called with a tainted parameter, its result is tainted, but when the `odd` function is called with an untainted parameter its result becomes untainted.

Another way of interacting with the results is by using the environments computed for each statement in the analyzed program. For the same example as above we can observe how each environment changes after each statement is executed (the results are a simplified version of an output from a pretty printer). Let's look for instance at the `main` function:

```
   int main(void)
   {
     [...]
1:   scanf("%d", &x);
     /*x = Tainted*/
2:   b1 = even(x);
     /*b1 = Tainted*/
3:   b2 = odd(10);
4:   y = compute(x);
     /*y = Tainted*/
5:   __retres = 0;
     [...]
   }
```

As you we said in chapter 2 the initial taint types for the local variables are untainted. In the output above we can observe how the environments change for each statement. Let's start with statement `1`. Because `x`'s value is read using `scanf` in the environment for statement `1` the mapping for `x` is changed to tainted. When other functions are called (statements `2`, `3`, `4`) their computed environment is instantiated according to the taint types of the actual parameters. For instance, when `even` is called, at statement `2`, the computed environment for the return statement in `even` has the mapping `Gamma(x)` for the return value. This taint variable is then instantiated according to the actual parameter `x` which is tainted, thus the new taint type for `1` in the environment of statement `2` is tainted. The same approach is used for all other function calls (statements `3` and `4`).

4.1.2 Metrics computation

A possible use of taint analysis (as described in chapter 2) is test generation and as we said before we would like to link the taint analysis computation to the test generation functionality. In order to do that, our implementation provides a few simple types of metrics that use the results computed by the taint analysis. STAC offers the ability to compute the execution paths with the minimum/maximum number of read operations but also the paths that may have the minimum/maximum number of tainted variables. Let's take a closer look at an output obtained for the same example used in section 3.2.3 in order to show STAC's path metrics computing functionality:

```
int compute(int x )
{ [...]
  /*BEST_PATH*/
  if (x == 2) {sum = taint();}
  else {/*BEST_PATH*/sum = 0;}
  /*BEST_PATH*/
```

```
  i = 0;
  /*BEST_PATH*/
  while (i < x) {/*BEST_PATH*/sum += i;
    /*BEST_PATH*/i ++;}
  [...]
}
```

This output corresponds to applying a metric that computes the path with the minimum number of tainted variables. In the output, the statements preceded by the comment `/*BEST_PATH*/` form the execution path with the highest metrics cost. As we can see there exists at least one point in the analyzed program where two different execution paths are created. We can consider it for instance the `if` statement from the `compute` function. In this case the "true" branch will be contained in one execution path and the "false" branch will be contained in another execution path (depending on the condition `x==2`). The `taint` function is a generic function added for testing purposes that always returns a tainted value. It is clear to us that from the two possible paths for this function, the one with the minimum number of reads will always contain the "false" branch of the `if`.

4.2 Architecture

In this section we will present the architecture of STAC, with a detailed description of how each module is implemented. An even more detailed view of the interfaces for STAC modules can be found in appendix A.2. As shown in figure 4.1, STAC is built as a multi-layer hierarchy of modules.

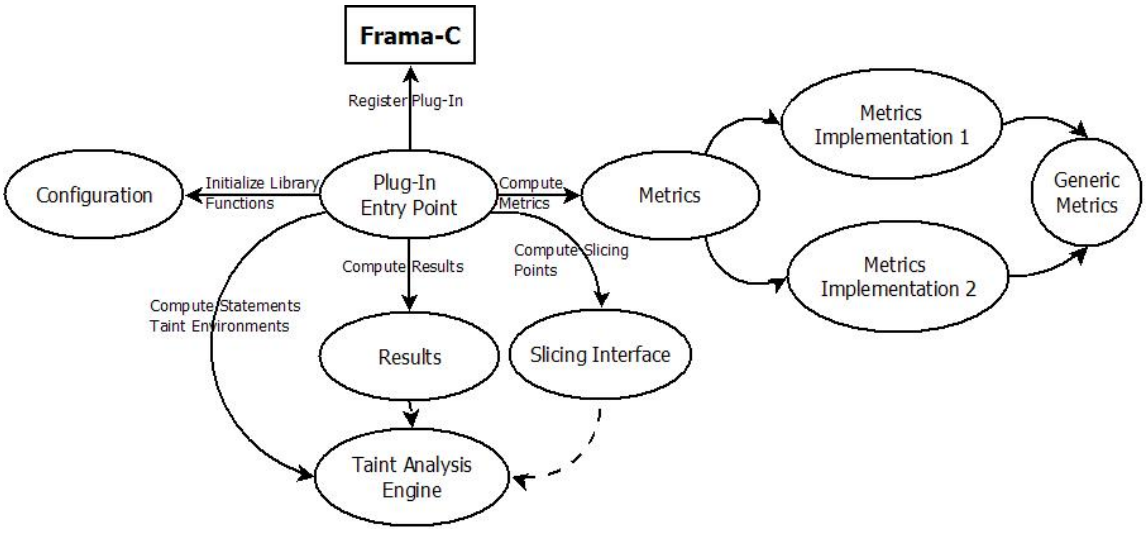


Figure 4.1: STAC - Software Architecture

4.2.1 Configuration Module

We start our presentation of STAC modules with the Configuration module. As we said until now, the main goal of STAC is to perform interprocedural analysis, so function calls must be taken into account. This is why the use of library functions creates a new problem for our approach. The reason for this is that usually we do not have access to the source code for the library functions. The solution we have chosen in order to avoid this problem is creating configuration files that store information (*summaries*) for library functions.

In order to understand how the configuration module works we have to take a look at the structure of the configuration files. The configuration file is built as a formatted text file with the following structure:

- for each library function the name, return type, return taint type and parameters are stored.
- for each library function parameter the name, type and taint type are stored. The parameter taint type (and also the return taint type) can be generic (dependent on some of the parameters), in this way allowing us to simulate different effects for the library function with respect to the calling context (this can also be seen as a kind of *polymorphism*).

Our tool comes with a set of predefined library calls summaries that is provided in the default configuration file.

Another functionality that requires additional configuration is the slicing interface (which will detect the possibly vulnerable points in the analyzed source file). One of the cases in which a statement can be vulnerable is when library function calls are made with unsafe (tainted) parameters. One such example is:

```
[...]
char str[1024];
scanf("%s", str);
printf(str);       // this is a string format
                   // vulnerability which may result in
                   // remote code execution
[...]
```

In the standard C library, the function `printf` accepts as first parameter a format string and afterward a variable number of parameters. If the format string comes from user-input and is not previously validated, a series of attacks that can lead to remote code execution can be performed [17]. In order for our tool to be able to detect such situations, STAC must be able to assign taint constraints to all possible vulnerable function calls. In this case we must be able to somehow say that the first parameter for the `printf` function should always be untainted. Creating such

constraints is also possible because of the Configuration module. The constraint configuration files have a similar structure to that of the library configuration files. Also, STAC comes with a set of predefined taint constraints that provided in the default constraints configuration file.

Our implementation of the configuration module consists of a small parser (that accepts structured configuration inputs as described above) and utilities for instantiating summaries and constraints and their corresponding taint environments for the library functions listed in the input configuration file.

4.2.2 Taint Analysis Engine

As shown in chapter 2, our approach for performing taint analysis consists of defining a type system in order to add additional taint type information to all the symbols in the analyzed program. This is implemented in the taint analysis engine module.

In a general sense, **Dataflow Analysis** is a technique for gathering information about the possible values computed at various points in a computer program, using the control flow graph for the program in order to see where the values can propagate. A simple way to perform dataflow analysis of programs is to set up dataflow equations for each node of the control flow graph and solve them by repeatedly calculating the output from the input locally at each node until the whole system stabilizes (it reaches a fixed point) [18]. In our case the values that we want to compute for all the points of the program are actually the taint environments associated to each statement in the program. The output for each node in the control flow graph is computed by applying the type inference rules described in 2 on the input for that node.

Our approach in performing the dataflow analysis tries to be an optimal one as shown in [5]. This means that a callgraph is initially computed for the analyzed program. As we saw in section 3.2 each strongly connected component in the callgraph corresponds to a set of mutually recursive functions. In order to minimize our computation we analyze the strongly connected components one by one in reverse topological ordering. This ensures the fact that when analyzing a component, all the components on which it depends have already been analyzed. For each strongly connected component, the analysis is performed in an iterative fashion until a fixed point is reached. For a better understanding we will use the sample code presented in section 3.2.3. We can collapse the graph to its strongly connected components and obtain the equivalent graph shown in figure 4.2.

A valid reverse topological oredering for this graph is `SC4`, `SC3`, `SC2`, `SC1`. Because the component `SC4` consists only of the function `taint` for which we have a sumarry, it will be skipped. So, our analysis will start with the `SC3` strongly con-

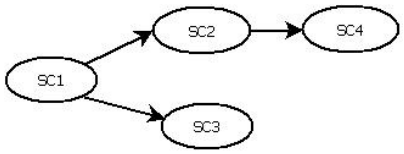


Figure 4.2: Equivalent SCC Graph

nected component and analyze the functions in it (`even` and `odd`) iteratively until a fixed point is reached for their taint environments. Afterward the `SC2` component is analyzed (the `compute` function is analyzed only once because it is a non-recursive function) and as you can see when analyzing `SC1` all the components on which it is dependent have already been analyzed. So, the last step in our analysis is to compute the environments for `SC1` (effectively for the `main` function).

The taint analysis engine module can be seen as consisting of four submodules as shown in figure 4.3. We will describe each of these submodules in the following subsections.

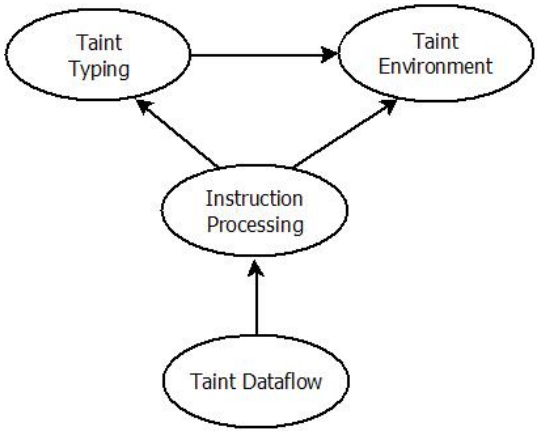


Figure 4.3: Taint Analysis Engine

4.2.2.1 Taint Environment

This submodule consists of the implementation of the type domain, type variables and environment for the taint analysis type system as described in chapter 2. As presented in the type system, we define two possible taint types T(tainted) and U(untainted) and type variables $G(x_i)$.

An environment is a mapping between symbols and their taint type. In our implementation we chose to build the environment as an association between the unique identifiers provided by CIL for program symbols and one of the previously defined

taint types. In order to track if the dataflow analysis has been performed on an environment, a flag is assigned to each environment in order to see if it was processed at least once during our analysis.

Because an environment is specific to each statement in the program we also had to create a mapping between unique statement identifiers provided by CIL and their corresponding environments. Also, as shown in chapter 2 we have to create a mapping Γ_{func} between functions and their associated environments.

The module provides functions for creating environments, querying or changing taint type information, comparing taint types or taint variables, comparing taint environments and also pretty printing envrionments and taint types.

4.2.2.2 Taint Typing

The taint typing submodule implements two very important functionalities.

First of all it provides an implementation for the $\oplus$ operator, both for taint types and taint environments, following its definition shown in chapter 2.

The second functionality is to create environment instances for function calls. As we stated in our taint analysis type system description, whenever a function call is reached, the environment for the callee is instantiated according to the taint types of the actual parameters. For instance, let's use again the sample code from section 3.2.3.

The function `even` is called from two different contexts: once from the `main` function and once again from the `odd` function. The computed environment for the `even` function contains the mapping `Symname: even = Gamma(x)` which states that the return taint type for `even` is dependent on the taint type of the `x` formal parameter. When the `even` function is called from `main` the taint type for the actual parameter is `T`, so the environment instance for the `even` function used by the function call statement will have the mapping `Symname: even = T` which means that in this context (with the given actual parameters) the `even` function returns a tainted result. On the other hand, when the `env` function is called from `odd`, the actual parameter taint is `Gamma(x)`, which means that it is dependent on the formal parameter of `odd`. So the environment instance for `even` used in this case will have the mapping `Symname: even = Gamma(x)`, meaning that the result of calling `even` in this context will depend on the taint type of `odd`'s formal parameter `x`.

4.2.2.3 Instruction Processing

The instruction processing submodule implements the type inference rules for our taint analysis type system by effectively performing the translation from the abstract

typing rules to the statements of the analyzed program. It actually represents the manipulation of the CIL provided control flow graph done by STAC. As we said in section 3.1.2, there are several kinds of CIL statements. The module receives as an input a statement and its current environment and computes the transformations made on the environment according to the type inference rules:

Instructions : can be of two types according to which, different transformations are performed:

- **assignment** instruction: using the current environment, the taint type for the right hand side expression is computed. The left hand side of the assignment can be a simple variable (in which case the taint type is assigned directly in the current environment to the variable), an array represented as a base and an offset (the new taint for the array will be obtained by applying the $\oplus$ operator between the right hand side expression taint and the offset taint type) or a structure field (in this case the field taint is computed and assigned to the whole structure if the structure is not already tainted).
- **function call** instruction: the previously computed environment for the called function is extracted. All type variables in the computed environment are instantiated according to the calling context (the type bindings for the actual parameters) and all the effects of the function are taken into account (the return value if present, the changes on the global variables and the changes on pointer parameters).

Return statements: if the returned expression is not nullable (when the function returns a non-void value), its taint is computed according to the current environment and the mapping for the function's return value is updated in the environment.

Jump statements (`goto`, `continue`, `break`): because this kind of statements do not change the environment, nothing is done in this case.

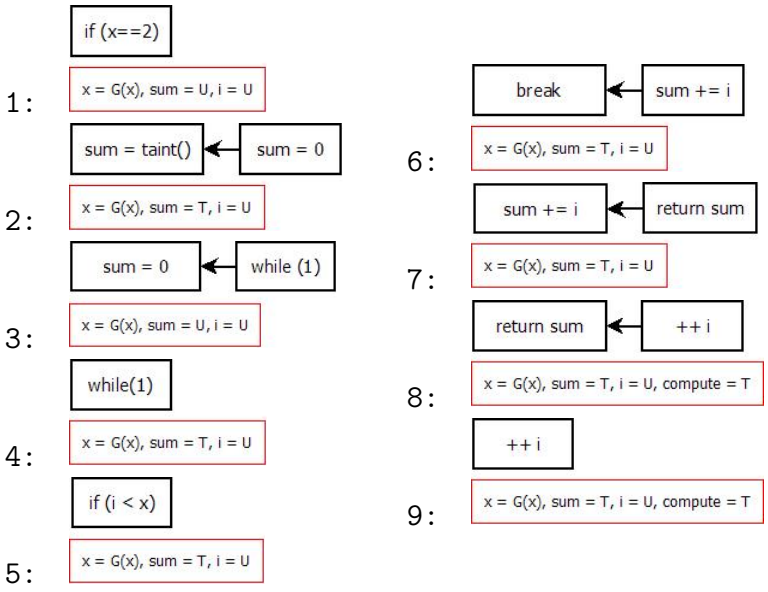
If and **Switch** statements: as shown in chapter 2, for each conditional statement, the condition taint type will be taken into account when computing the environments for the conditional statement branches. The approach chosen in our implementation is to maintain a stack of condition taint for each analyzed statement. The taint for the condition expression is pushed onto the stack whenever a conditional statement is reached and the stack is popped whenever analyzing a conditional statement finishes.

Loop statements: as we saw earlier in section 3.1.2, all the loops in the original program are changed to `while(1)` loops. This means that our fixed point computation as presented in 2 will be performed on the body of the loop, and the successor of the loop statement will be determined by the `break` instruction added by CIL.

4.2.2.4 Taint Analysis As Dataflow Analysis

This part of the module is responsible for performing dataflow analysis. The analysis is performed on the control flow graphs of each function. For each analysis of a given function a statement worklist is maintained and an environment is associated to each statement in the function. Initially the worklist contains only the entry point statement for the analyzed function. The analysis is performed iteratively until the worklist becomes empty. In each iteration, the first statement in the worklist is extracted and its effect on its associated environment is computed. If the associated environment changes or if there are any successors of the current statement that have not already been analyzed, all its successors are added at the end of the worklist. When adding successors to the worklist, if the current statement corresponds to a conditional statement, the conditional stacks associated to the successors are updated.

Let's take a closer look on how the worklist is maintained when the `compute` function is analyzed for the sample code in 3.2.3:



The worklist is detailed and for each step in the analysis, the red rectangle represents the environment associated to the head of the worklist after it has been analyzed.

- step 1: the worklist contains only the first statement of the function (`if(x == 2)`. The taint type for the conditional expression in this case is $G(x)$, due to the fact that `x` is a formal parameter. The successor statements, `sum=taint()` and `sum=0`, are added at the end of the worklist.

- step 2: the head of the worklist is the statement `sum=taint()`. The assignment is computed, thus the mapping for `sum` in the current environment is set to T. Because the environment has changed, the successor is added to the worklist (the loop statement).
- step 3: the `sum=0` statement is analyzed and the environment doesn't change.
- step 4: the `while(1)` statement is analyzed and the mapping for `sum` changes to T because of combining the T and U values for `sum` coming from the predecessors. Its successor is added to the worklist.
- step 5: the taint type for the condition is $G(x)$ because of the comparison to `x` which has the type $G(x)$. Both successors are added to the worklist.
- step 6: in the environment for the `break` statement the mapping for `sum` changes to T because of the environment from its predecessor.
- step 7: the `sum += i` statement is analyzed. Because `sum`'s mapping to T in the predecessor, the mapping for `sum` changes to T in the current environment. The successor, `++i` is added to the worklist.
- step 8: the `return sum` statement is analyzed. The mapping for the `compute` function return value is changed to T.
- step 9: the `++i` statement is analyzed changing the mapping for `i` to $G(x)$ because of the loop condition. Because `i`'s mapping changed, the loop will be analyzed once more but the other variable mappings will not change, so we will not present that part of the analysis.

4.2.3 Results Module

One of the goals of **STAC** is to determine which points in the analyzed program can become security breaches because of user supplied input. In order for us to see if it is a feasible approach we have created the **Results** module which computes the percentage of tainted instructions with respect to the number of executed instructions in all possible calling contexts. An instruction is considered tainted if it assigns (directly or indirectly through function calls) a variable with a tainted value. As we said, the module must compute the percentages in all calling contexts because the same instruction in a function can be a tainted instruction in one calling context and untainted in another calling context (depending on the supplied actual parameters).

In order to compute the number of tainted instructions we have decided to perform a traversal of the control flow graph of the analyzed program using as starting point the entry-node of the `main` function. In order to make our interpretation context-sensitive we have decided to define a *function stack* which consists of *function stack frames*. A *function stack frame* is actually the instance of the current

function taint environment obtained after instantiating all the taint type variables in the previously computed environment according to the taint types of the actual parameters. A similar approach for interpreting the results of user-input dependence analysis was used in [5] when computing the results for the context-sensitive case.

The main problem of analyzing all the function contexts arises because of the presence of recursive functions throughout the analyzed program. In order to solve this problem we have chosen to apply an approximation for our computation. Whenever a function call node is reached a search is made in the current function stack for a stack frame corresponding to a different environment instance for the callee function. If one such stack frame is found then the node is not expanded. In this manner we can be sure that our computation will finish at some point without looping indefinitely.

4.2.4 Slicing Interface

Another functionality that can be found in STAC is the slicing interface. As shown earlier Frama-C offers us slicing functionalities through the slicing plug-in presented in 3.5.5. In our current implementation we have chosen to interact with the slicing plug-in through *slicing pragmas*. As presented in the description of the configuration module, the slicing interface uses custom user defined constraints to identify the statements in the program that must be kept in the slice prepared for future analysis.

The method used to detect the slicing points is very similar to the one explained for the results module. The control flow graph is traversed and when a constraint is violated a new slicing point is stored for the current statement.

After computing the slicing points for the analyzed program, STAC outputs a source file annotated with slicing pragmas such that the slicing plug-in can be run on the new output in order to obtain a smaller more precise (in the sense that it will contain only the parts of the program that are likely to contain security vulnerabilities) program.

4.2.5 Generic Metrics Module

As stated in the beginning of this chapter the results obtained after running the taint analysis computation can be used for finding the execution path with the highest cost with respect to given metrics. In our approach we decided to implement a generic module for this kind of computation.

A global graph is built based on the control flow graph. All the blocks in the control flow graph are considered nodes in the graph, except function calls. Each

basic block that contains a function call is split into two nodes (a *BeginCall* node and an *EndCall* node). For each *BeginCall* node an edge is created from it to the entry point of the called function. For each *EndCall* node an edge is created from the exit point of the called function to it.

The generic computation module will require the other modules that use it to supply metrics values and metrics operators for each edge in the graph with respect to the edge margins.

Because the metrics module must compute the execution path with the highest cost, loops (as normal program constructs or as mutual recursive functions) represent a problem. So, at first the loops are detected using Tarjan's strongly connected components algorithm [19]. For each loop the back edges are removed so that the new graph will not contain any cycles. Afterward, the loop edges that remain in the graph will never be selected into the best path because of the removed back edges.

Now, with our new acyclic directed graph we can reduce the problem to a graph shortest path problem by negating the values on the remaining edges. After running Dijkstra's algorithm we obtain the best path with respect to our metrics in the acyclic directed graph.

The last step in our approach is to add the previously removed back edges that are on the best path, and for each back edge add the value of the loop it represents as a supplementary metric value.

The metrics module is parametrized by a given type `t` which corresponds to the metric type and by functions that can perform operations with metrics of type `t`. Some of the operations that a metrics implementation must provide are:

- **value_default** is the metric value that will be used as a default value for an edge
- **value_zero** is the metric value that will be used as an identity element for the operations between metric values.
- **value_add** is the operation that implements the addition of two metric values.
- **value_sub** is the operation that implements the subtraction of two metric values.
- **node_value** is the function that will be used to determine the metric value for a given graph node
- **edge_value** is the function that will be used to determine the metric value for an edge based on its limits.

4.2.6 Metrics Implementation

We have used the generic metrics module in order to implement the metrics computations for four kinds of metrics. These metrics implementations will be described in the following sections.

4.2.6.1 Min/Max Read Metrics

A *read operation* can be defined as a call to a function that requests user input, for instance `scanf`, `read` or `fscanf`. Testing a program can be a rather difficult task when the program uses a great amount of *read operations*. A solution to this problem could be to determine the execution path that contains the smallest amount of read operations. This can easily be reduced to a metrics computation problem by adding weights to the edges in our graph with respect to the number of reads done when executing that edge.

To exemplify the possible way of defining such a metrics we have built the min read metric module which implements the functionalities required by the generic metrics module with respect to the minimum read metrics described above. The module defines the type of a metric as the pair consisting of the number of read operations performed on edges that are not in loops and the number of read operations performed on edges the are part of loops in the execution graph.

On the other hand, during the testing process we may want to test the execution path that could be the most error-prone. With respect to the number of read operations this metrics could be defined in a similar way as the min read metrics, the only difference being in the fact that the cost on the determined execution path must be the highest possible. The generic metrics module allows us to implement such kinds of metrics by assigning negative values as weights on the edges of the graph. So for our max read metrics all we have to do is define a metrics that negates the one for min read metrics: the pair consisting of the **negated** number of reads performed on edges that are not in loops and the **negated** number of reads performed on edges the are part of loops in the execution graph.

4.2.6.2 Min/Max Taint Metrics

A more accurate way of defining metrics is using the results obtained from our taint analysis computation. In order to do that we have created the minimum taint metrics and maximum taint metrics implementations. These modules try to compute the execution path with the minimum (respectively maximum) number of tainted variables. For our approach we have chosen to implement the minimum taint metrics as a pair of components corresponding to the cost of the path in the non-cyclic part and in the cyclic part. Each component is built as a pair consisting of:

- the number of symbols dependent on the parameters from the current node. This is always `zero` for edges that end in nodes that are not function calls.
- the number of symbols that may become tainted because of the destination node, when the current edge is executed.

Probably this is not the most accurate metrics that we can build, but it shows how to link together the taint analysis and metrics computation.

By negating the minimum taint metrics we can define a new metrics for computing the execution path with the highest number of tainted variables (in a similar manner as shown for the max read metrics).

Chapter 5

Experimental Results

This chapter will present some of the results we have obtained after running STAC on different test cases. The results presented below have been computed using the results module described in section 4.2.3.

During our experiments we have used a few small size programs in order to test the functionalities of STAC, but also some medium size programs in order to see what the gain would be if taint analysis would be used to obtain a smaller part of the code that can be vulnerable to security attacks. The examples have been chosen in the following way: we selected some programs developed that implement complex algorithms (usually using arrays, because one of the most common vulnerability is the buffer overflow) and that require a reasonable amount of user input, but also we have chosen larger applications in order to see how STAC performs in a more realistic environment.

Program	#loc	#fun	#instr	#tainted instr	tainted percentage
Queue	227	10	80	55	68.7%
ABR	408	28	1937	1828	94.3%
Huffman	499	30	336	220	65.4%
Link Editor	6732	184	6686	4362	65.2%
mailx	14609	226	38147	22214	58.2%

Table 5.1: Taint Analysis experimental results

The notations used in table 5.1 are described below:

- **#loc** is the number of lines of code and **#fun** is the number of functions in the analyzed program
- **#instr** represents the number of CIL statements possibly executed in all calling contexts. As we have shown in section 4.2.3 a statement can be considered

tainted in one calling context and untainted in another one. This is why we have chosen to show the number of possibly executed statements in all calling context which can be greater or equal than the number of statements in the program if at least a function is called twice in the program.

- **#tainted instr** is the number of tainted statements during the program execution. This count takes into account the contexts in which a statement can be executed. For instance, if a statement is executed in two different contexts and in both of them it is a tainted statement then the number of tainted statements is incremented by two.

- **tainted percentage** is computed as the percentage of the #tainted instr from the #instr

Let us take some time and have a closer look at the applications we used in order to get our experimental results:

Queue : is a student developed application that implements a linked list data structure and its associated operations in order to simulate the behaviour of a printing systems with multiple requests. The user provided input is the number of requests that must be simulated.

ABR : is a student developed application that implements a binary tree data structure and its associated operations (for building trees, adding, removing, searching nodes) and computes some basic informations (like the number of leaves, the maximum height, the medium length of the paths in the tree, the number of nodes on each level) for a user supplied number of randomly generated trees.

Huffman : also a student developed application that implements the data structures and algorithms required for performing Huffman encoding. The program requires user input in the form of an input text file and will write the results to an output file.

Link Editor : a student project that takes multiple object files obtained after compilation and builds an executable file. The user input required by the program is in the form of the input object files.

mailx : is a Unix utility program for sending and receiving mail, also known as a Mail User Agent program. It is an improved version of the mail utility. Mailx allows one to send and read email. Mailx cannot, by itself, receive email from another computer. It reads messages from a file on the local machine, which are delivered there by a local delivery agent such as procmail. Besides the commands supplied by the user, mailx has as a source of tainted data the file containing the delivered messages thus making it a very good benchmark for us.

The obtained values are listed in table 5.1 As one can see the percentage of tainted statements ranges from 58.2% to 94.3%. Even if 94.3% seems like a very bad percentage this is partially because of the fact that the analyzed program is very small and implements different operations on a binary tree which has only tainted values as nodes. The last two test cases consist of two bigger programs (with respect to the number of lines), a link editor and the Linux mailx mail client.

The obtained results are also influenced by the fact that STAC does not implement alias analysis for the moment, so probably a number of false negatives have appeared. Also the results are dependent on how the library calls are annotated. In the case when a library call is not annotated STAC makes the worst case assumption that the return value of the function is tainted (and also the pointer parameters and globals).

We also used as a benchmark the NIST SAMATE Reference Dataset [32] which has as main purpose to provide users, researchers, and software security assurance tool developers with a set of known security flaws. The dataset includes production, synthetic (written to test or generated), and academic test cases. This database also contains real software application with known bugs and vulnerabilities. The dataset intends to encompass a wide variety of possible vulnerabilities, languages, platforms, and compilers. After running STAC on 307 testcases with the default configuration files provided in our implementation we obtained tainted percentages ranging from 10% (on the smaller test cases) to 75% with an average taint percentage of 46.47%. The results are encouraging based on the number of executed tests and a full set of results is provided in appendix A.3.

Chapter 6

Comparison With Related Work

In this chapter we will present some of the related work that has been done, both for taint analysis and for other analyses that use a similar approach as ours and try to make a comparison with what we have achieved. We will compare STAC with some existing tools for static and dynamic taint analysis and also with some non-interference implementations.

6.1 Taint Analysis

As we presented in chapter 2 there are two main methods of performing taint analysis: dynamically and statically. Generally speaking there are two approaches when speaking about taint analysis: with or without sanitization. Sanitization is the process in which tainted data can be untainted after some checks (called sanity checks) are performed on it. As we saw in chapter 4, STAC implements sanity checks by using summaries for library functions. A similar effect is obtained by other existing tools by using annotations like Parfait does [9]. Another way of performing sanity checks, although it is not a very safe approach, is making empirical assumptions as done by the Pixy implementation [12] (for instance that at the return of a function it is safe to assume that all variables are untainted). Another widely used approach is a mixture of the two presented above. For instance the **taint mode** in Perl in order to track user input dependent data, uses both empirical assumptions and programmer supplied annotations in order to detect the points in the program where the data should be untainted.

Another interesting problem is how the taint analysis is performed. In the static approach, most of the implementations reduce this problem to a dataflow analysis problem, a technique largely used in software verification tools and also in compiler optimizations. This is the case for the Pixy implementation and also the Parfait implementation (even though the Parfait approach is to reduce the dataflow problem to a graph reachability problem). The implementation of STAC is also built as a dataflow computation as described in chapter 4.

Most of the dynamic taint analysis implementations, like the ones supplied by Perl and BuzzFuzz [6], maintain additional runtime information about the values of the variables in the program in the current execution environment in order to track the user dependent variables.

6.1.1 Static: Parfait, Pixy

As the reader has probably seen by now, two of the implementations that are closest to STAC are the ones provided by Parfait and Pixy. Because Pixy deals with vulnerabilities in PHP scripts thus eliminating some of the problems that arise from the structure of C programs (for instance the presence of pointers), we will focus on comparing the implementation of STAC with the one supplied by Parfait.

As described in [5] Parfait uses the LLVM (Low Level Virtual Machine) as its front end for C programs. The LLVM provides an intermediate representation of the analyzed program in the form of SSA form (Static Single Assignment form) and also built in analyses for performing different helpful tasks like alias analysis.

The Parfait implementation uses a special form of SSA called Augmented SSA form (aSSA form). The aSSA form extends the phi-nodes built for the SSA form by adding control dependencies for all the variables that influence the phi-node (in the form of predicates that condition the effect of a variable in a phi-node).

Because the implementation of STAC is based on determining the type of a symbol (following the type system described in chapter 2), type that can be different for different statements in the analyzed program, our approach does not need to perform an additional computation in order to obtain the SSA form. We only need to have the control flow graph (which is supplied by CIL) and our type inference rules can be applied as shown in chapter 4.

For now, an important drawback for our approach is the fact that STAC does not perform any alias analysis. This is why on test cases similar in size as the ones used by Parfait, or even on the same test case (for instance when analyzing the `mailx` application) we seem to obtain significantly better results than Parfait. It is true that when not performing alias analysis a few false negatives may be overlooked but it is our belief that, when real life applications are tested, the influence of the existing aliases is not so critical and that our results will not differ very much from the ones presented in chapter 5.

Parfait implements two approaches when performing taint analysis. The first one is a context insensitive approach which does not keep track of the calling context for a function in order to compute its user input dependent variables, in order to obtain a faster (but more imprecise) computation and the second one is a context sensitive

one. As described in our type system specifications and also in the implementation of STAC, we have chosen to implement the context sensitive approach in order for us to obtain more precise results.

The main difference between Parfait and STAC is that STAC provides full taint typing information for each statement in the analyzed program. This means that after running the analysis we know at the statement level the exact taint value for all the variables.

6.1.2 Taint Analysis and test generation: BuzzFuzz

As we stated earlier in chapter 2, one of our goals is to perform automatic test generation based on the results provided by different program analyses (in this case taint analysis). An existing tool that performs automatic test generation and testing is BuzzFuzz [6]. BuzzFuzz performs program testing in multiple steps. First the program is run with one or more valid inputs and dynamic taint tracing is performed. Based on the gathered taint information and on empirical assumptions of possibly vulnerable library function calls (or user configured library calls) attack points are computed (points in the program where security breaches may occur). For each attack point and each sample input, BuzzFuzz computes the set of input bytes that affect the values at that attack point. Using this information, new inputs are generated as follows: each new input is identical to one of the sample inputs, except that the input bytes that affect the values at one or more attack points have been altered. Finally, BuzzFuzz runs the program on the newly generated inputs to see if any errors occur.

The main difference in our approach is caused by the fact that our analysis is a static one. Because of the enormous number of possible execution paths, we have decided, as shown in chapter 2, to determine the execution paths with the highest probability of generating security problems. For now, our approach is a simple one, but still uses the results provided by the static taint analysis performed by STAC. The next step would be to effectively generate test cases for the execution paths computed earlier.

6.2 Non Interference

Non Interference can be seen as a form of static analysis that could be integrated into a compiler to verify secure information flows in programs. One can easily compare non interference with the Bell-La Padula [20], which focuses on data confidentiality and access to classified information. One approach for implementing non interference is shown in [14] and formulates the non interference problem as defining a type system for which all well-typed programs respect the non interference property.

The difference between non interference and taint analysis as type system implementations is that when computing taint information we do not have to ensure the fact that for each language construction the security constraints hold. In the case of taint analysis the taint type information is only propagated to all statements in the program by applying the type inference rules. So all programs that follow the specification provided for the underlying programming language are well-typed with respect to the taint analysis type system.

Chapter 7

Conclusions

In this chapter we will make a small overview of what was achieved throughout the project and as well of the work that still needs to be done in order to obtain sound and more precise results.

7.1 Status Quo

In chapter 2 we have introduced a new approach on performing taint analysis by creating our taint analysis type system along with its inference rules. Initially, the system dealt with a subset of the C language, but as shown later it can be easily extended in order to keep in mind all the constructions that can exist in a real C program. The main advantage of this approach is that it creates a very well defined separation between the formal specifications of our problem (in this case performing taint analysis) and its implementation. We have also shown in a theoretical manner how the results of static analyses (in this case taint type information) can be used in order to perform automatic test generation for C programs.

Another achievement of our project is providing an implementation for the theoretical notions presented earlier. In our case, this is done with the help of STAC which, as we have shown, is a plug-in for the Frama-C analysis platform (described in chapter 3). STAC successfully implements our taint analysis type system, reducing it to a context sensitive interprocedural dataflow analysis which is implemented in an efficient manner as shown in chapter 4. As we have seen, STAC also provides functionalities for integrating additional analyses in order to create better and more precise software verification tools. Some of the most important features are:

- assigning taint informations for all the symbols at each statement in the analyzed program.
- allowing the user to create custom function summaries in order to obtain more precise results.

- allowing the user to supply constraints for function calls in order to determine the points in the analyzed program where security vulnerabilities may occur.
- interfacing the taint analysis with a more complex analysis provided by the Frama-C slicing plug-in in order to obtain smaller sources on which security vulnerabilities can be discovered easier.
- providing a generic framework for computing metrics for the execution paths most likely to generate security breaches. Also some metrics implementations are already provided by STAC, integrating the results provided by the taint analysis.

7.2 Future Work

We can split the work that still has to be done into two main categories: taint analysis features and test generation features. With respect to taint analysis there are still some aspects which are not dealt with:

- as shown earlier, our analysis isn't yet sound because it does not keep in mind the possibility of aliasing in C programs. For now STAC makes the assumption that no aliases exist in the program, that means that each variable has only one alias, itself. This can be solved in different ways: by adding some kind of alias analysis to our implementation or by integrating our implementation with the Value Analysis Frama-C plug-in which also computes the aliases for the variables in the program.
- another limitation of STAC is represented by the fact that the memory locations are considered independent (you cannot navigate from a variable to another variable using pointers). This limitation appears because of the use of the Frama-C memory model presented in chapter 3. A possible way of overcoming this limitation is to create a custom memory model for our implementation which deals with a more precise memory model. On the other hand the situations that are not dealt with by the Frama-C memory model are very rarely found in real life programs and could easily be considered as bad practices.
- we could improve the degree of preciseness of our taint analysis by using a more fine approach when assigning taint types. This is the case of structures and arrays, which in our current implementation are considered as whole entities. Probably better results can be obtained if the taint types are assigned directly to structure fields and not to structures as a whole. The same approach could be used for arrays too (although it is conditioned by the use of a very precise memory model in order to specify the smallest entity that can be considered a part of the array).

- STAC also does not offer any support for multi-threaded program analysis.

In order to achieve a fully functional implementation for automatic test generation, STAC has to be extended in such a way that it can extract the different conditions needed to activate the execution path computed by the metrics module. So as to be able to generate real tests, the previously computed conditions have to be analyzed in order to determine the input that will activate each of them.

Appendix A

Appendix

A.1 STAC User Manual

As all the analyzers built for Frama-C, the taint analysis plug-in has several sub-options that activate different behaviours for the tool. These can be viewed using the command: `frama-c.byte -help`. The available switches are detailed here:

- **taint-analysis**: this is the main switch for the plug-in. It enables the inter-procedural taint analysis.
- **-print-intermediate**: prints all the intermediate taint environments computed for each function's analysis. This is useful when we want to see the effect of recursive (or mutual recursive) functions on the taint flow.
- **-print-final**: for each function in the program, prints the taint values for its variables after the whole program has been analyzed. This provides a more general view on how each function changes its values according to the formal parameters and the global variables.
- **-print-source**: after the analysis is performed, the source code is rewritten adding taint annotations as C comments. A taint annotation will appear after each statement for each variable that changes its taint after executing the statement.
- **-do-results**: computes a percentage of tainted instructions with respect to the number of instructions executed in all possible contexts of the functions. Each function can called in any context so its statements may or may not taint some variables depending on the calling context. An instruction is considered tainted if it assigns (directly or indirectly - through function calls with pointers as parameters) a variable with a tainted value.
- **-do-prepare-slice**: prints a source file prepared for slicing with respect to vulnerable statements. This option is still experimental.

- **-do-min-read-metrics**: computes the execution path that will have the minimum number of read function calls (a read function is a function that provides user input which is always tainted). This is useful when trying to find the test-case that requires the minimum of user input.
- **-do-max-read-metrics**: computes the execution path that will have the maximum number of read function calls. This is also useful if we want to find the most accurate test-case (the execution path that tests the program with the maximum amount of user input).
- **-do-min-taint-metrics**: computes the execution path that has the minimum number of possibly tainted variables
- **-do-max-taint-metrics**: computes the execution path that has the maximum number of possibly tainted variables
- **-config-file**: specifies the path to the file containing the summaries for library functions. If this option isn't used the plug-in searches for a default configuration file (default.cfg) in the current directory. This configuration file is required because of our interprocedural approach. So, if we don't have access to a library call code to analyze it, the taint analysis plug-in requires a summary of how it changes the calling contexts environment with respect to its parameters.
- **-taint-debug**: turn ON debug logging.
- **-taint-info**: turn ON info logging.

A.2 Implementation Interfaces

A.2.1 Configuration Module

```
module Configuration = struct
  (* Return or parameter type and taint information. *)
  type info = string * Cil_types.typ * taintMetaValue
  (* Creates the library function summary based on *)
  (* the informations gathered by the configuration *)
  (* file parser from the input configuration. *)
  val add_function info : info list -> unit
  (* Initializes the parser and calls the functions *)
  (* that perform the configuration instantiation. *)
  val init : unit -> unit
end
```

A.2.2 Taint Analysis Engine

Taint Environment `module TaintEnv = struct`

```
(* Taint Domain values. *)
type taintValue = T | U | G of Cil_types.varinfo list
(* Taint Domain values for configuration use. *)
type taintMetaValue = M_T | M_U | M_G of string list
(* Taint Constraint types. *)
type taintConstraint = string * taintMetaValue
(* Taint environment. Is associated to each statement. *)
type environment = bool * ((int, taintValue) Hashtbl.t)
(* Mapping between statements and environments. *)
type statementsEnvironment = environment Inthash.t
(* Mappings between function ids and statements environments. *)
functionEnvironment = (environment * statementsEnvironment)
                          Inthash.t
(* Creates a new empty environment. *)
val create_env : unit -> environment
(* Returns the stored taint type value for the *)
(* given unique symbol id. *)
val get_taint : environment -> int -> taintValue
(* Sets the taint type value for the *)
(* given unique symbol id. *)
val set_taint : environment -> int -> taintValue -> unit
(* Compares two taint types. *)
val compare_taint : taintValue -> taintValue -> bool
(* Compares two environments. *)
val compare : environment -> environment -> bool
end
```

Taint Typing `module Typing = struct`

```
(* Applies the + operator between two taint type variables. *)
val combine_taint : taintValue -> taintValue -> taintValue
(* Applies the + operator between two taint environments. *)
val combine : environment -> environment -> environment
(* Instantiates a function environment wrt. symbols' bindings. *)
val instantiate_func_env : functionEnvironment ->
                                    (varinfo * taintValue) list
end
```

Instruction Processing `module InstrComputer = struct`

```
(* Creates an initial environment for a given function *)
(* definition. Adds all the formal parameters, local *)
(* variables and the return value to the environment. *)
```

```
    val create_initial_env : fundec -> environment
    (* Analyzes an expression according to the current *)
    (* environment. An  expression can also hide a function *)
    (* call in which case the parameter list isn't empty. *)
    val do_expr : environment -> exp -> exp list
            -> functionEnvironment -> environment
    (* Analyzes an instruction according to the current *)
    (* environment and the current conditional taint. *)
    val do_instr : environment -> instr -> taintValue
            -> functionEnvironment -> environment
    (* Analyzes a return instruction for a given current *)
    (* environment and function. *)
    val do_return_instr : environment -> fundec
            -> exp option -> taintValue
            -> functionEnvironment -> environment
    (* Analyzes an if instruction and returns the new *)
    (* conditional taint value to be pushed onto the stack. *)
    val do_if_instr : environment -> exp -> taintValue
            -> functionEnvironment -> taintValue
    (* Analyzes a loop instruction. *)
    val do_loop_instr : environment -> taintValue
            -> taintValue * environment
  end
```

Taint Dataflow `module TaintComputer = struct`

```
    (* Implements the iterative dataflow analysis. Receives *)
    (* the current worklist and performs the necessary environment *)
    (* updates. *)
    val compute : (stmt * (taintValue list)) list -> unit
    (* Applies the dataflow analysis on a specific basic block. *)
    (* Returns the modified environment and the actions that must *)
    (* be done on the associated condition taint stack. *)
    val do_stmt : stmt -> environment -> taintValue
            -> (environment * taintStack)
  end
```

A.2.3 Results Module

```
module ResultsComputer = struct
  (* Symbolic execution stack frame. *)
  type func_frame = Frame of fundec
  (* Symbolic execution stack. *)
  type func_stack = func_frame list
  (* Checks if the taint value associated to the given statement *)
```

```
  (* is Tainted. *)
  val check_tainted : taintValue -> stmt -> bool
  (* Performs a symbolic execution of the given statement *)
  (* according to the current environment. The execution *)
  (* stack is used if the current statement is a function call. *)
  val do_instr : environment -> stmt -> func_stack
                 -> fundec -> instr -> unit
  (* Performs the breadth first traversal and mantains the *)
  (* execution stack. *)
  val compute : environment -> func_stack
                 -> fundec * stmt list -> unit
end
```

A.2.4 Generic Metrics Module

```
module MetricComputer = struct
  (* Creates the main graph based on the CFGs for all the functions *)
  (* in the program. *)
  val create_graph : fundec list -> graphType
  (* Removes all the back edges, thus eliminating all loops and *)
  (* returns the modified graph and the list of removed edges. *)
  val break_loops : graphType -> graphType * edgeType list
  (* Computes the best path for an directed acyclic graph and returns it *)
  (* and its value. *)
  val get_best_path : graphType -> stmt -> stmt
                 -> (nodeType list) * metricType
  (* Adds the previously removed back-edges for the loops that are *)
  (* on the best path and changes the path metric accordingly. *)
  val add_removed_edges : graphType -> nodeType list -> metricType
                 -> edgeType list -> (nodeType list) * metricType
end
```

A.3 STAC SAMATE results

Program	#loc	tainted percentage
./315/memccpy-fix1.c	22	20.0%
./437/threaded_locked_double_strncat_bad1.c	73	61.5%
./434/threaded_locked_double_strcpy_bad1.c	61	60.0%
./319/strcat-bad1.c	33	40.0%
./574/into2-ok.c	63	62.5%
./289/ns-lookup-bad.c	384	34.3%
./548/ahscpy1-bad.c	54	46.1%

./333/threaded_bcopy_bad1.c	54	60.0%
./627/snp6-ok.c	53	62.5%
./342/threaded_strcat_bad1.c	62	61.5%
./500/Figure3-2-unix.c	39	50.0%
./348/threaded_strcpy_fix1.c	60	60.0%
./364/threaded_locked_strcat_bad1.c	63	61.5%
./316/memmove-bad1.c	26	37.5%
./818/fmt3-bad.c	53	28.5%
./813/Buffer_overflow.c	7	40.0%
./324/strcpy-bad1.c	27	28.5%
./630/snp8-bad.c	57	22.2%
./338/threaded_memmove_bad1.c	60	60.0%
./648/tain2-ok.c	65	45.4%
./320/strcat-bad2.c	37	50.0%
./406/threaded_locked_asym2_realpath_bad1.c	57	60.0%
./374/threaded_locked_strncpy_bad1.c	59	60.0%
./889/path_basic_bad.c	48	35.7%
./901/resource_injection_scope.c	44	36.3%
./591/mem3-ok.c	54	75.0%
./405/threaded_locked_asym2_memmove_bad2.c	56	60.0%
./413/threaded_locked_asym2_strcpy_bad2.c	59	60.0%
./400/threaded_locked_asym2_bcopy_bad2.c	58	60.0%
./881/os_cmd_local_flow.c	41	12.5%
./588/mem2-bad.c	52	57.1%
./641/spr4-bad.c	54	62.5%
./845/heap_overflow_cplx.c	51	57.1%
./593/nonul1-ok.c	64	20.0%
./615/scpy9-ok.c	67	63.1%
./614/scpy9-bad.c	67	63.1%
./621/snp3-ok.c	52	28.5%
./353/threaded_strncpy_fix1.c	57	60.0%
./397/threaded_locked_asym1_strncat_bad3.c	68	61.5%
./335/threaded_memccpy_bad1.c	58	60.0%
./368/threaded_locked_strcpy_bad1.c	61	60.0%
./814/Use_of_hard-coded_password1.c	38	25.0%
./841/hardcoded_pass_loop.c	31	71.4%
./283/call_fb_realpath.c	108	33.3%
./544/ahdec1-bad.c	79	06.6%
./382/threaded_locked_asym1_memmove_bad1.c	61	60.0%
./290/ns-lookup-ok.c	370	35.3%
./352/threaded_strncpy_bad1.c	58	60.0%

./805/path_aliaslevel_good.c	85	57.1%
./317/memmove-bad2.c	27	28.5%
./545/ahdec1-ok.c	79	06.6%
./617/snp1-ok.c	52	28.5%
./343/threaded_strcat_bad2.c	65	61.5%
./639/spr3-bad.c	55	37.5%
./838/hardcoded_pass_container_good.c	42	33.3%
./586/mem1-ok.c	54	37.5%
./407/threaded_locked_asym2_scpy_fix3.c	71	60.0%
./339/threaded_memmove_bad2.c	57	60.0%
./849/improper_null_term_basic.c	30	40.0%
./363/threaded_locked_scpy_fix3.c	71	60.0%
./625/snp5-ok.c	54	37.5%
./393/threaded_locked_asym1_strncat_bad1.c	72	61.5%
./864/lock_resource_good.c	60	05.5%
./834/fmt_string_local_control_flow_good.c	36	25.0%
./314/memccpy-bad2.c	26	28.5%
./887/path_aliaslevel_bad.c	72	54.8%
./748/Format_string_problem.c	18	16.6%
./897/resource_injection_basic.c	40	11.1%
./903/stack_overflow_array_index.c	23	50.0%
./623/snp4-ok.c	75	12.5%
./613/scpy8-ok.c	58	20.0%
./430/threaded_locked_double_strcpy_bad1.c	61	60.0%
./344/threaded_strcat_fix1.c	67	61.5%
./556/fmt1-ok.c	47	33.3%
./311/bcopy-bad1.c	32	37.5%
./636/spr1-bad.c	52	28.5%
./511/Figure5-1-unix.c	34	60.0%
./402/threaded_locked_asym2_memccpy_bad2.c	57	60.0%
./898/resource_injection_basic_good.c	59	56.2%
./564/gets1-ok.c	47	16.6%
./554/fmt2-bad.c	47	33.3%
./372/threaded_locked_strncat_bad1.c	72	61.5%
./395/threaded_locked_asym1_strncpy_fix1.c	57	60.0%
./842/hardcoded_pass_loop_good.c	39	60.0%
./506/Figure4-12-unix.c	38	45.4%
./379/threaded_locked_asym1_memccpy_bad1.c	59	60.0%
./811/Use_of_hard-coded_password1.c	38	25.0%
./846/heap_overflow_cplx_good.c	50	57.1%
./737/heapinspection.c	75	14.2%

./328/strncat-bad2.c	33	40.0%
./573/into2-bad.c	64	62.5%
./609/scpy6-bad.c	63	30.0%
./828/dble_free_local_flow_good.c	51	21.0%
./425/threaded_locked_double_memccpy_fix1.c	61	60.0%
./906/stack_overflow_array_length_good.c	28	55.5%
./419/threaded_locked_asym2_strncpy_bad2.c	57	60.0%
./575/into3-bad.c	64	62.5%
./605/scpy4-bad.c	63	70.5%
./361/threaded_locked_memmove_bad2.c	57	60.0%
./632/snp9-bad.c	78	11.5%
./436/threaded_locked_double_strcpy_fix1.c	64	60.0%
./558/fmt2-ok.c	47	33.3%
./365/threaded_locked_strcat_bad2.c	71	61.5%
./416/threaded_locked_asym2_strncat_bad2.c	69	61.5%
./284/call_fb_realpath.c	108	33.3%
./377/threaded_locked_asym1_bcopy_bad1.c	58	60.0%
./336/threaded_memccpy_bad2.c	57	60.0%
./600/scpy1-bad.c	52	28.5%
./389/threaded_locked_asym1_strcat_fix2.c	71	61.5%
./441/threaded_locked_double_strncpy_bad2.c	59	60.0%
./855/improper_null_term_basic_container_good.c	38	50.0%
./610/scpy6-ok.c	60	30.0%
./830/dble_free_loop_good.c	30	14.2%
./640/spr3-ok.c	54	37.5%
./931/os_cmd_injection_basic_good.c	45	38.4%
./357/threaded_locked_memccpy_bad1.c	59	60.0%
./323/strcat-fix1.c	28	50.0%
./351/threaded_strncat_bad3.c	65	61.5%
./367/threaded_locked_strcat_fix2.c	71	61.5%
./557/fmt2-bad.c	47	33.3%
./435/threaded_locked_double_strcpy_bad2.c	63	60.0%
./550/chroot1-bad.c	55	50.0%
./833/fmt_string_local_control_flow.c	36	25.0%
./428/threaded_locked_double_realpath_bad1.c	61	60.0%
./375/threaded_locked_strncpy_bad2.c	58	60.0%
./330/strncpy-bad1.c	27	37.5%
./412/threaded_locked_asym2_strcpy_bad1.c	58	60.0%
./422/threaded_locked_double_bcopy_bad2.c	60	60.0%
./611/scpy7-bad.c	57	22.2%
./782/Unintentional_pointer_scaling.c	16	40.0%

./861/leftover_debug.c	55	64.2%
./563/gets1-bad.c	47	16.6%
./394/threaded_locked_asym1_strncat_bad2.c	67	61.5%
./313/memccpy-bad1.c	27	37.5%
./424/threaded_locked_double_memccpy_bad2.c	59	60.0%
./322/strcat-fix2.c	28	50.0%
./312/bcopy-bad2.c	27	28.5%
./862/leftover_debug_good.c	65	64.2%
./349/threaded_strncat_bad1.c	66	61.5%
./409/threaded_locked_asym2_strcat_bad2.c	71	61.5%
./332/strncpy-fix2.c	26	25.0%
./356/threaded_locked_bcopy_bad2.c	60	60.0%
./604/scpy3-ok.c	53	25.0%
./391/threaded_locked_asym1_strcpy_bad2.c	62	60.0%
./647/tain2-bad.c	52	25.0%
./326/strcpy-fix1.c	28	37.5%
./345/threaded_strcat_fix2.c	67	61.5%
./553/fmt1-ok.c	47	33.3%
./561/fmt4-ok.c	55	42.8%
./341/threaded_scpy_fix3.c	69	60.0%
./603/scpy3-bad.c	53	25.0%
./438/threaded_locked_double_strncat_bad2.c	69	61.5%
./736/heapinspection.c	75	14.2%
./607/scpy5-bad.c	62	70.5%
./850/improper_null_term_basic_@alias.c	29	40.0%
./589/mem2-ok.c	54	75.0%
./628/scpy7-bad.c	57	22.2%
./626/snp6-bad.c	55	62.5%
./373/threaded_locked_strncat_bad2.c	69	61.5%
./749/Format_string_problem.c	18	16.6%
./585/mem1-bad.c	54	42.8%
./566/gets2-ok.c	47	16.6%
./321/scpy-fix3.c	33	70.5%
./285/call-realpath-bad.c	132	33.3%
./369/threaded_locked_strcpy_bad2.c	62	60.0%
./831/fmt_string_local_container.c	32	20.0%
./899/resource_injection_container.c	48	18.1%
./403/threaded_locked_asym2_memccpy_fix1.c	59	60.0%
./856/improper_null_term_basic_good.c	33	50.0%
./599/race1-ok.c	56	25.0%
./815/submitted_path_aliaslevel_bad.c	67	54.8%

./882/os_cmd_local_flow_good.c	56	36.8%
./804/path_aliaslevel_bad.c	67	54.8%
./423/threaded_locked_double_memccpy_bad1.c	59	60.0%
./552/fmt1-bad.c	47	33.3%
./602/scpy2-ok.c	56	37.5%
./366/threaded_locked_strcat_fix1.c	72	61.5%
./832/fmt_string_local_container_good.c	32	20.0%
./404/threaded_locked_asym2_memmove_bad1.c	58	60.0%
./622/snp4-bad.c	74	12.5%
./598/race1-bad.c	56	25.0%
./857/improper_null_term_basic_taint.c	30	20.0%
./392/threaded_locked_asym1_strcpy_fix1.c	64	60.0%
./638/spr2-ok.c	59	28.5%
./551/chroot1-ok.c	57	55.5%
./415/threaded_locked_asym2_strncat_bad1.c	68	61.5%
./331/strncpy-bad2.c	26	28.5%
./359/threaded_locked_memccpy_fix1.c	61	60.0%
./848/heap_overflow_location_good.c	50	57.1%
./427/threaded_locked_double_memmove_bad2.c	58	60.0%
./835/hardcoded_pass_buffer.c	35	17.6%
./631/snp8-ok.c	57	22.2%
./756/write_what_where.c	32	53.3%
./401/threaded_locked_asym2_memccpy_bad1.c	58	60.0%
./825/dble_free_local_flow_good.c	51	21.0%
./429/threaded_locked_double_scpy_fix3.c	72	60.0%
./843/heap_overflow_array.c	30	25.0%
./410/threaded_locked_asym2_strcat_fix1.c	72	61.5%
./854/improper_null_term_basic_container.c	35	40.0%
./446/badfree_013.c	32	66.6%
./507/Figure4-17-unix.c	42	60.0%
./376/threaded_locked_strncat_bad3.c	70	61.5%
./888/path_aliaslevel_good.c	90	57.1%
./829/dble_free_loop.c	30	15.3%
./560/fmt3-ok.c	53	28.5%
./620/snp3-bad.c	55	28.5%
./440/threaded_locked_double_strncpy_fix1.c	59	60.0%
./418/threaded_locked_asym2_strncpy_fix1.c	59	60.0%
./629/snp7-ok.c	52	28.5%
./286/call-realpath-ok.c	131	50.0%
./608/scpy5-ok.c	62	70.5%
./371/threaded_locked_strncpy_fix1.c	59	60.0%

./645/tain1-bad.c	52	25.0%
./616/snp1-bad.c	52	28.5%
./411/threaded_locked_asym2_strcat_fix2.c	69	61.5%
./847/heap_overflow_location.c	51	57.1%
./816/submitted_path_aliaslevel_good.c	85	57.1%
./398/threaded_locked_asym1_strncpy_bad2.c	56	60.0%
./318/realpath-bad1.c	28	28.5%
./885/os_cmd_scope.c	36	12.5%
./547/ahgets1-ok.c	51	50.0%
./433/threaded_locked_double_strcat_fix2.c	71	61.5%
./350/threaded_strncat_bad2.c	65	61.5%
./578/into4-ok.c	64	62.5%
./559/fmt3-bad.c	53	28.5%
./812/Use_of_hard-coded_password1.c	38	25.0%
./417/threaded_locked_asym2_strncpy_bad1.c	59	60.0%
./577/into4-bad.c	64	62.5%
./562/fmt5-ok.c	50	55.5%
./905/stack_overflow_array_length.c	28	55.5%
./895/resource_injection_@alias.c	44	10.0%
./414/threaded_locked_asym2_strcpy_fix1.c	63	60.0%
./388/threaded_locked_asym1_strcat_fix1.c	72	61.5%
./633/snp9-ok.c	79	11.5%
./810/Use_of_hard-coded_password1.c	38	25.0%
./606/scpy4-ok.c	62	70.5%
./358/threaded_locked_memccpy_bad2.c	59	60.0%
./347/threaded_strcpy_bad2.c	60	60.0%
./858/improper_null_term_basic_taint_good.c	32	33.3%
./900/resource_injection_container_good.c	69	55.5%
./355/threaded_locked_bcopy_bad1.c	58	60.0%
./420/threaded_locked_asym2_strncat_bad3.c	70	61.5%
./576/into3-ok.c	64	62.5%
./383/threaded_locked_asym1_memmove_bad2.c	57	60.0%
./572/into1-bad.c	59	62.5%
./592/nonul1-bad.c	63	22.2%
./637/spr2-bad.c	62	28.5%
./840/hardcoded_pass_local_flow_good.c	38	62.5%
./354/threaded_strncpy_bad2.c	58	60.0%
./386/threaded_locked_asym1_strcat_bad1.c	64	61.5%
./612/scpy8-bad.c	58	20.0%
./442/threaded_locked_double_strncat_bad3.c	70	61.5%
./929/unchecked_error_condiftion_good.c	29	10.0%

./432/threaded_locked_double_strcat_fix1.c	72	61.5%
./890/path_basic_good.c	69	44.4%
./390/threaded_locked_asym1_strcpy_bad1.c	61	60.0%
./512/Figure5-24-unix.c	40	42.8%
./325/strcpy-bad2.c	28	37.5%
./650/tain4-bad.c	45	16.6%
./385/threaded_locked_asym1_scpy_fix3.c	71	60.0%
./601/scpy2-bad.c	59	37.5%
./399/threaded_locked_asym2_bcopy_bad1.c	58	60.0%
./635/snp10-ok.c	58	50.0%
./886/os_cmd_scope_good.c	54	36.8%
./439/threaded_locked_double_strncpy_bad1.c	59	60.0%
./883/os_cmd_loop.c	34	62.5%
./634/snp10-bad.c	60	50.0%
./839/hardcoded_pass_local_flow.c	32	80.0%
./378/threaded_locked_asym1_bcopy_bad2.c	60	60.0%
./346/threaded_strcpy_bad1.c	59	60.0%
./396/threaded_locked_asym1_strncpy_bad1.c	57	60.0%
./370/threaded_locked_strcpy_fix1.c	64	60.0%
./340/threaded_realpath_bad1.c	58	60.0%
./902/resource_injection_scope_good.c	66	70.0%
./844/heap_overflow_array_good.c	29	25.0%
./426/threaded_locked_double_memmove_bad1.c	61	60.0%
./590/mem3-bad.c	57	66.6%
./334/threaded_bcopy_bad2.c	59	60.0%
./837/hardcoded_pass_container.c	36	10.5%
./565/gets2-bad.c	47	16.6%
./884/os_cmd_loop_good.c	51	57.8%
./549/ahscpy1-ok.c	56	40.0%
./387/threaded_locked_asym1_strcat_bad2.c	71	61.5%
./851/improper_null_term_basic_@alias_good.c	31	50.0%
./852/improper_null_term_basic_buffer@type.c	31	40.0%
./384/threaded_locked_asym1_realpath_bad1.c	58	60.0%
./431/threaded_locked_double_strcat_bad2.c	71	61.5%
./803/path_basic_good.c	64	44.4%
./362/threaded_locked_realpath_bad1.c	58	60.0%
./380/threaded_locked_asym1_memccpy_bad2.c	59	60.0%
./642/spr4-ok.c	53	62.5%
./408/threaded_locked_asym2_strcat_bad1.c	63	61.5%
./928/unchecked_error_condiftion.c	24	14.2%
./555/fmt1-bad.c	47	33.3%

./327/strncat-bad1.c	33	45.4%
./421/threaded_locked_double_bcopy_bad1.c	59	60.0%
./896/resource_injection_@alias_good.c	65	52.9%
./329/strncat-bad3.c	33	45.4%
./624/snp5-bad.c	55	37.5%
./546/ahgets1-bad.c	50	53.8%
./836/hardcoded_pass_buffer_good.c	49	36.8%
./802/path_basic_bad.c	43	35.7%
./381/threaded_locked_asym1_memccpy_fix1.c	61	60.0%
./649/tain3-bad.c	45	16.6%
./337/threaded_memccpy_fix1.c	61	60.0%
./360/threaded_locked_memmove_bad1.c	61	60.0%
./904/stack_overflow_array_index_good.c	23	50.0%

References

[1] Aleph One (Levy E.): Smashing the Stack for Fun and Profit, (1996). 1.1

[2] Buffer Overrun in JPEG Processing (GDI+), Microsoft Security Bulletin MS04-028, (2004). 1.1

[3] Moore, D., Paxson, V., Savage, S., Shannon, C., Staniford, S., Weaver, N.: Inside the Slammer Worm. IEEE Security and Privacy 1 (2003) 1.1

[4] Tsipenyuk K., Chess B., McGraw G.: Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors, IEEE Security & Privacy, vol 3, no 6, (2005). 2.1

[5] Scholz B., Zhang C., Cifuentes C.: User-Input Dependence Analysis via Graph Reachability, SMLI TR-2008-171 (2008). 2, 2.1.2, 4.2.2, 4.2.3, 6.1.1

[6] Ganesh V., Leek T., Rinard M.: Taint-based Directed Whitebox Fuzzing, ICSE (2009). 2.1.1, 2.3, 6.1, 6.1.2

[7] Newsome J., Song D.: Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software, NDSS (2005). 2.1.1

[8] Clause J., Li W., Orso A.: Dytan: A Generic Dynamic Taint Analysis Framework, Proceedings of the 2007 international symposium on Software testing and analysis, (2007). 2.1.1

[9] Cifuentes C., Scholz B.: Parfait: Designing a scalable bug checker, Eighth IEEE International Working Conference, (2008). 2.1.2, 6.1

[10] Shankar U., Talwar K., Foster J. S., Wagner D.: Detecting Format-String Vulnerabilities with Type Qualifiers, 10th USENIX Security Symposium, (2001). 2.1.2

[11] Chang R., Jiangy G., Ivancic F., Sankaranarayanany S., Shmatikov V.: Inputs of Coma: Static Detection of Denial-of-Service Vulnerabilities, CSF (2009). 2.1.2

[12] Jovanovic N., Kruegel C., Kirda E.: Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities, IEEE Symposium (2006). 2.1.2, 6.1

[13] Pierce, B.: Pixy: Types and Programming Languages, MIT Press (2002). 2.2

[14] Volpano D., Smith G., Irvine S.: A Sound Type System for Secure Flow Analysis, Journal of Computer Security, Vol 4, No 3 (1996). 2.1.2, 2.2.3, 6.2

[15] Molnar D. A., Wagner D.: Catchconv: Symbolic execution and run-time type inference for integer conversion errors, (2007). 2.3.1

[16] Necula G. C., McPeak S., Rahul S.P.,Weimer W.: CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs, Proceedings of Conference on Compiler Construction, (2002). 1.1, 3, 3.3, 3.6.1

[17] Foster J. C., Osipov V., Bhalla N., Heinen N.: Buffer Overflow Attacks: Detect, Exploit, Prevent, Syngress Publishing, (2005). 4.2.1

[18] Kildall, G.: A Unified Approach to Global Program Optimization, Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. 4.2.2

[19] Tarjan R.: Depth-first search and linear graph algorithms, SIAM Journal on Computing Vol 1 (1972), No 2, P. 146-160. 4.2.5

[20] Bell D. E., La Padula L. J.: Secure Computer Systems (1973). 6.2

[21] http://www.owasp.org, Open Web Application Security Project website. 2.1

[22] http://bitblaze.cs.berkeley.edu/, BitBlaze: Binary Analysis for Computer Security website. 2.1.1

[23] http://www.splint.org/, Open Web Application Security Project website. 2.1.2

[24] http://frama-c.cea.fr, Frama-C website. 1.1, 3.3

[25] http://caml.inria.fr/ocaml/index.en.html, Objective Caml website. 3.3

[26] http://frama-c.cea.fr/download/frama-c-manual-Lithium-en.pdf, Frama-C user manual. 3.4

[27] http://frama-c.cea.fr/download/plug-in_development_guide.pdf, Frama-C plug-in development manual. 3.6

[28] http://frama-c.cea.fr/acsl.html, ANSI/ISO C Specification Language. 3.5.7, 3.6.1

[29] http://why.lri.fr/, Why platform website. 3.5.7

[30] http://ocamlgraph.lri.fr/, OCamlGraph website. 3.2.2

[31] http://vulcain.imag.fr/index.html, VULCAIN website. 1.2

[32] http://samate.nist.gov/SRD/view.php, NIST SAMATE Reference Dataset website. 5