Defense against software exploits

El Habib Boudjema

UNIVERSITÉ PARIS EST
ÉCOLE DOCTORALE MSTIC

THÈSE DE DOCTORAT

pour obtenir le titre de

Docteur de l'Université Paris-Est

SPÉCIALITÉ : INFORMATIQUE

défendue par

BOUDJEMA El habib

Défense contre les attaques de logiciels

soutenue le ***04 mai 2018*** *devant le jury composé de :*

Prof. ZEADALLY Sherali	Rapporteur	University of Kentucky, USA
Dr. SAUVERON Damien	Rapporteur	Université de Limoges, France
Prof. BEN-OTHMAN Jalel	Président	Université Paris 13, France
Prof. HABBAS Zineb	Examinateur	Université de Lorraine, France
Prof. KHEDOUCI Hamamache	Examinateur	Université Lyon 1, France
Dr. VERLAN Serghei	Examinateur	Université Paris-Est, France
Prof. MOKDAD Lynda	Directrice de thèse	Université Paris-Est, France
Dr. FAURE Christèle	Co-directrice de thèse	Saferiver, France
Dr. DELEBARRE Véronique	Invitée	Dirigeante Saferiver, France

UNIVERSITÉ PARIS-EST

Thèse préparée à:
Saferiver, Montrouge. France
et
Laboratoire LACL Université Paris-Est, Créteil. France

Abstract

Defense Against Software Exploits

In the beginning of the third millennium, we are witnessing a new age. This new age is characterized by the shift from an industrial economy to an economy based on information technology. It is the *Information Age*. Today, we rely on software in practically every aspect of our life. Information technology is used by all economic actors: manufactures, governments, banks, universities, hospitals, retail stores, etc. A single software vulnerability can lead to devastating consequences and irreparable damage. The situation is worsened by the software becoming larger and more complex making the task of avoiding software flaws more and more difficult task. Automated tools finding those vulnerabilities rapidly before it is late, are becoming a basic need for software industry community.

This thesis is investigating security vulnerabilities occurring in C language applications. We searched the sources of these vulnerabilities with a focus on C library functions calling. We dressed a list of property checks to detect code portions leading to security vulnerabilities. Those properties give for a library function call the conditions making this call a source of a security vulnerability. When these conditions are met, the corresponding call must be reported as vulnerable. These checks were implemented in Carto-C tool and experimented on the Juliet test base and on real life application sources. We also investigated the detection of exploitable vulnerability at binary code level. We started by defining what exploitable vulnerability behavioral patterns are. The focus was on the most exploited vulnerability classes such as stack buffer overflow, heap buffer overflow and use-after-free. After, a new method on how to search for these patterns by exploring application execution traces is proposed. During the exploration, necessary information is extracted and used to find the patterns of the searched vulnerabilities. This method was implemented in our tool Vyper and experimented successfully on Juliet test base and real life application binaries.

Résumé

Défense contre les attaques de logiciels

Dans ce début du troisième millénium, nous sommes témoins d'un nouvel âge. Ce nouvel âge est caractérisé par la transition d'une économie industrielle vers une économie basée sur la technologie de l'information. C'est *l'âge de l'information*. Aujourd'hui le logiciel est présent dans pratiquement tous les aspects de notre vie. Une seule vulnérabilité logicielle peut conduire à des conséquences dévastatrices. La détection de ces vulnérabilités est une tâche qui devient de plus en plus dure surtout avec les logiciels devenant plus grands et plus complexes.

Dans cette thèse, nous nous sommes intéressés aux vulnérabilités de sécurité impactant les applications développées en langage C et particulièrement les vulnérabilités provenant de l'usage des fonctions de ce langage. Nous avons proposé une liste de vérifications pour la détection des portions de code causant des vulnérabilités de sécurité. Ces vérifications sont sous la forme de conditions rendant l'appel d'une fonction vulnérable. Des implémentations dans l'outil Carto-C et des expérimentations sur la base de test Juliet et les sources d'applications réelles ont été réalisées. Nous nous sommes également intéressés à la détection de vulnérabilités exploitables au niveau du code binaire. Nous avons défini en quoi consiste le motif comportemental d'une vulnérabilité. Nous avons proposé une méthode permettant de rechercher ces motifs dans les traces d'exécutions d'une application. Le calcul de ces traces d'exécution est effectué en utilisant l'exécution concolique. Cette méthode est basée sur l'annotation de zones mémoires sensibles et la détection d'accès dangereux à ces zones. L'implémentation de cette méthode a été réalisée dans l'outil Vyper et des expérimentations sur la base de test Juliet et les codes binaires d'applications réelles ont été menées avec succès.

Keywords:
static analysis, **vulnerability detection**, **software exploit**

Mots-clés :
analyse statique, **détection de vulnérabilité**, **attaque logicielle**

Contents

List of Figures

List of Tables

Abbreviations

ACE	Arbitrary code execution
ACSL	Annotated C Specification Language
AEG	Automatic Exploit Generation
ANSI	American National Standards Institute
ANTLR	ANother Tool for Language Recognition
ASLR	Address Space Layout Randomization
AST	Abstract Syntax Tree
BNF	Backus–Naur Form
CAS	Center for Assured Software
CFG	Control Flow Graph
CIA	Confidentiality Integrity Availability
CIL	C Intermediate Language
CVE	Common Vulnerability Exposures
CVSS	Common Vulnerability Scoring System
CWE	Common Weakness Enumeration
DEP	Data Execution Protection
DFG	Data Flow Graph
EDG	Edison Design Group
FP	False Positive
GCC	GNU Compiler Collection
GLR	Generalized LR
IDE	Integrated Development Environment

IETF	Internet Engineering Task Force
ISO	International Organization for Standardization
JIL	Java Intermediate Language
LLNL	Lawrence Livermore National Laboratory
LLVM	Low Level Virtual Machine
MAN	MANual
MSDN	MicroSoft Developer Network
NSA	National Security Agency
OWASP	Open Web Application Security Project
RAM	Random-Access Machine
RASP	Random-Access Stored-Program
SAMATE	Software Assurance Metrics and Tool Evaluation
SATE	Static Analysis Tool Exposition
VSA	Value-Set Analysis

Introduction

"Write something down, and you may have just made a mistake."

David Brumley.

In a more and more digitalized world, it has become necessary to ensure the security of computer and information systems. The concept of security in computer systems is built on Confidentiality, Integrity and Availability aka CIA criteria [1].

The security of an information system is a global property that must hold at every level of the system. In other words, security is seen as a chain that is as weak as its weakest ring [2]. A strategy that can be used to provide a security of a system is *Defense in depth* at every system level. This means that each system's component must not rely on other components security but has to ensure its own security. Software free from vulnerabilities and flaws is one of the building blocks of defense in depth tactics. Software vulnerabilities and flaws can be very expensive. A software flaw may lead spacecrafts to explode [3], make nuclear centrifuges spin out of control [4], or force a car manufacturer to recall thousands of faulty cars [5]. Worse, security-critical bugs tend to be hard to detect, harder to protect against, and up to one hundred times more expensive after the software is deployed [6]. The situation becomes worse with program's size and complexity growing very fast [7]. Senior code reviewers can no longer analyze applications with millions of code lines to find these nasty flaws in a reasonable time. This situation makes it urgent to have automated detection systems to be able to find vulnerabilities the earliest in software development cycle. For this reason, the solutions and tools

automatically finding vulnerabilities are gaining interest by different IT actors (researchers, engineers, developers, etc). Code analysis and more especially the *static code analysis* is a possible solution to this problematic situation. In this thesis, we explored this technique, proposed new solutions, implemented and tested new static analysis tools with a focus on *security vulnerabilities*.

Thesis contributions

Static analysis for security is the main subject of this thesis. Three topics were dealt with along this research work:

- **Security vulnerability detection in C language applications**
  In this contribution it is described a method and a tool based on abstract interpretation extended with security vulnerability property checks. The coverage of security vulnerabilities represents a key difference with existing tools such as (Polyspace [8, 9], Frama-C [10, 11] and Astrée [12]). The second novelty is that the proposed vulnerability checks are obtained by analyzing the language specification and its standard libraries documentation. This makes our work different and complementary to tools such as Fortify [13] and Coverity [14]. So, we focus on a set of vulnerabilities derived from the usage of C language library functions. We define properties that can be checked to locate these vulnerabilities. For each defined property, we provide the related attack scenario to show its effect on security. These properties were implemented and evaluated in Carto-C tool.

- **Exploitable vulnerability detection on binary code**
  This contribution describes how to detect exploitable vulnerabilities at binary code level with almost no false positive. The given solution makes use of concolic execution [15] in order to explore execution paths to compute reachable program states. On these computed states different behavioral patterns of vulnerable code can be recognized and reported. The proposed solution

also permits to a software analyst to easily confirm the reported vulnerability by providing him the input sample that can trigger it. Our solution can also be used to automatically sort true and false positive vulnerabilities obtained from other software analysis tools. This contribution is very related to the first and can be seen as a logical continuation. In the first contribution, we searched for large classes of vulnerability without questioning their exploitability. But, in real life situations, exploitable vulnerabilities must be treated and patched before non-exploitable ones to reduce the chances of successful attacks. This pushed to be more interested in exploitable vulnerabilities. The choice of analyzing binary code is motivated by the fact that at this low level all details are available to accurately qualify an exploitable vulnerability.

- **Tools implementation and evaluation**

  In this contribution, we present developments and experimentations done along this dissertation. We provide details on the Carto-C and Vyper tool implementations. The correctness of the proposed solutions are demonstrated via experimental evaluation using the publicly available Juliet test base [16]. The use of Juliet test base provides a reference benchmark for comparison with other existing tools. On the other hand, the developed tools were tested on real life applications showing their effectiveness and their limitations.

Thesis outline

The chapter 1 introduces the cyber security landscape. It defines precisely what is meant by *security vulnerability*. It shows how and where *static analysis* can be placed among dozens of inter-related cyber security domains. In chapter 2 it is given examples showing C language complexity and their effect on security. Different C language library functions are studied from a security point of view.

This chapter shows the necessity of tools checking for dangerous security flaws in application written in C language. In chapter 3, we give details on existing static analysis tools and techniques. It is dealt with main static analysis tool components: front-end, middle-end and back-end. The chapter 4 covers the first contribution on how to use source code static analysis tool to detect security vulnerabilities on C language applications using abstract interpretation extended with security property checks. In chapter 5, we describe the solution we propose for exploitable vulnerability detection on binary application code using concolic execution to recognize the pattern of searched vulnerabilities. The chapter 6 describes all what has been implemented and experimented. We conclude and discuss what have been done in the chapter 7. Chapters' dependency is described in the figure 1.

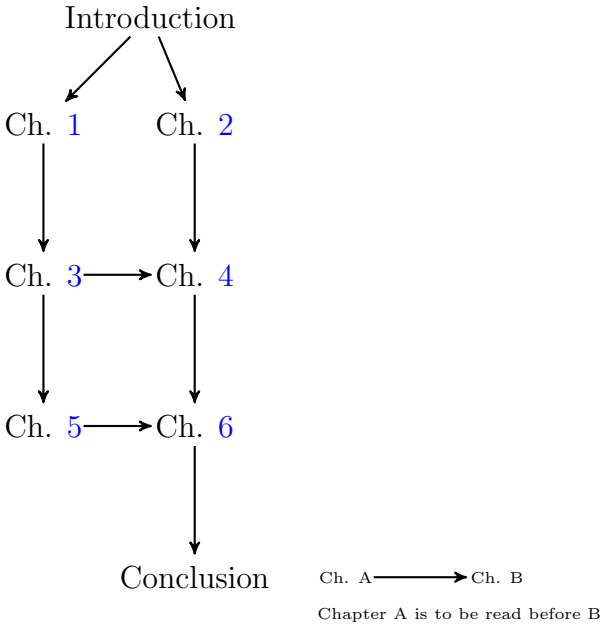

FIGURE 1: Chapter dependencies

Chapter 1

Overview of cyber security domains

"Security is a process, not a product."

Bruce Schneier.

The growing number of interconnected computer systems, and the increasing reliance upon them by individuals, businesses, industrial entities and governments means that there are an increasing number of systems at risk. According to Ponemon Institute's 2016 Cost of Data Breach Study [17], the average total cost of losing sensitive corporate or personal information is approximately $4 billion per year. Per stolen record, businesses and associations can spend anywhere between $145 and $158.

1.1 Cyber security: a growing large field

Cyber security aims the protection of computer systems from the theft and damage to their hardware, software or information they hold. This domain is of a great importance for every IT actor ranging from simple home users to big government

agencies. As shown in the figure 1.1 [18], this domain is very large and is getting larger and larger. Cyber security covers technical fields such as *security operations, security architecture* or *risk assessment*, regulatory domains such as *governance or standard compliance* and human related fields such as *career development or user education*. We focus on the risk assessment domain and more specially on vulnerability scan and source code scan sub-domains. The vulnerability scan is run generally using scanning tools [19] like Nessus, Qualys, etc. These tools will report already known vulnerabilities. These known vulnerabilities are published and documented on different on-line databases such as the Common Vulnerabilities and Exposures (CVE), which is maintained by the MITRE [20] with funding from the national cyber security division of the USA government. The task of vulnerability scanning needs to be run continuously to insure that the information system is at least immune against known vulnerabilities. On the other side, software code scanning either in white box with access to the source code or in black box with an access to only software binaries can be used to discover vulnerabilities generally new (called zero days by the security community). This is done using static analysis tools described in the chapter 3 or when applicable using manual auditing. The result of source scanning is a set of alerts or warnings explaining where the vulnerability is located. The use of Common Weakness Enumeration CWE [21] taxonomy gives software auditors ease to use different tools or techniques and keeping the same vocabulary to describe the same things.

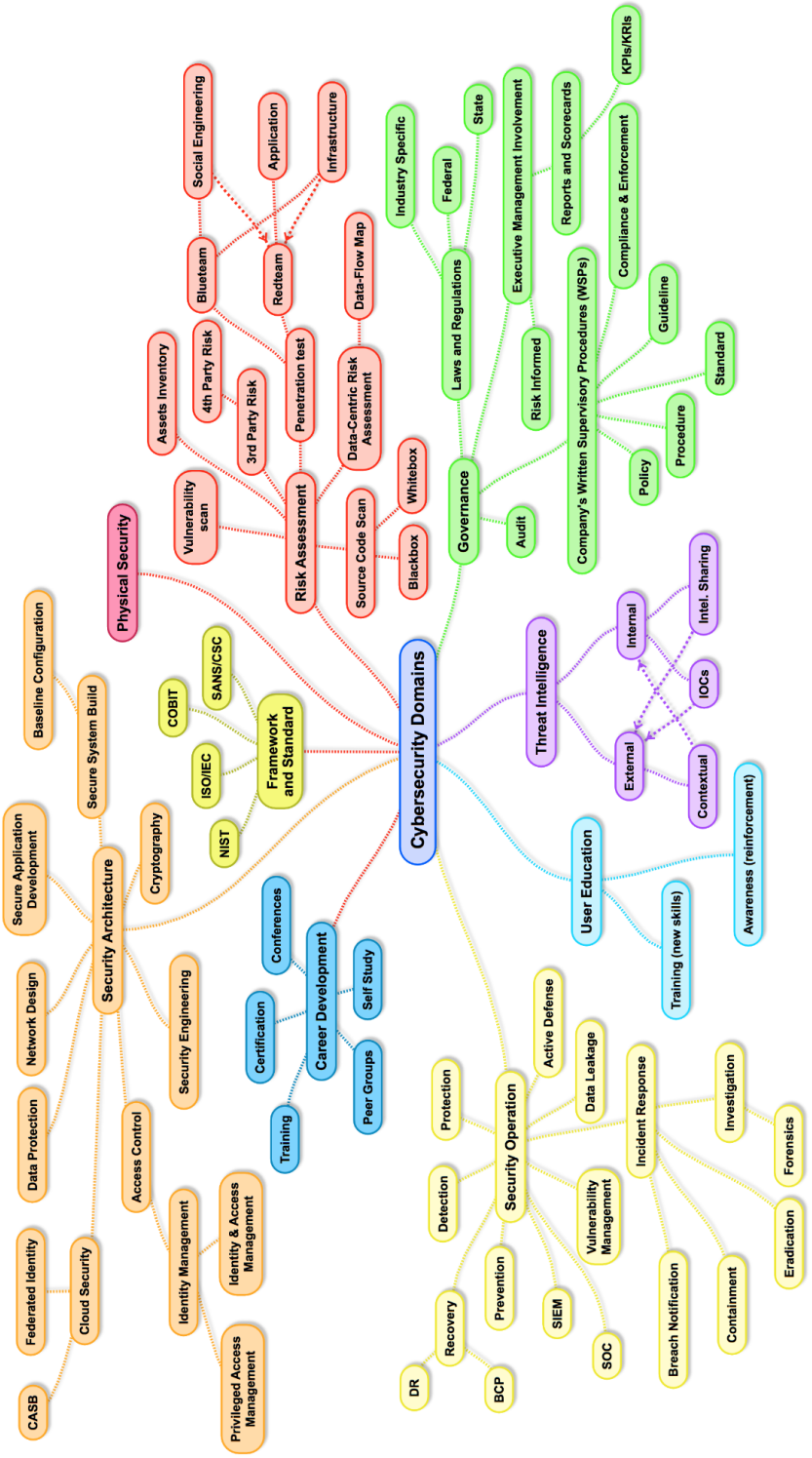

FIGURE 1.1: Cyber security domains

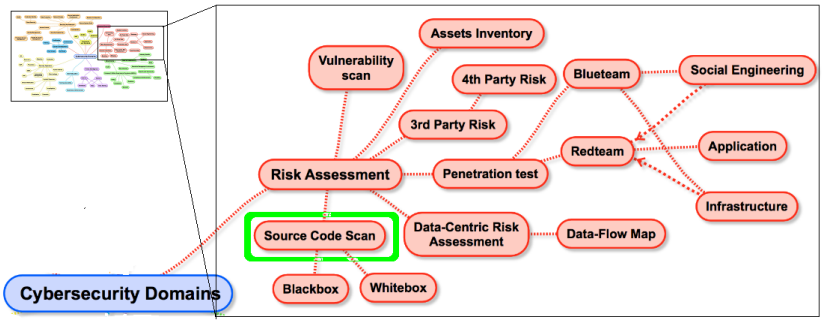

FIGURE 1.2: Risk assessment cyber security domain

1.2 Security vulnerabilities: public enemy of cyber security

The security of a system is a global property and no aspect can be neglected without exposing the whole system to a risk. It is often seen as a chain that is as weak as its weakest ring [2]. Having good quality software with robust security contributes a lot to the security of the whole system. Different domains in cyber security deal with software quality. The *risk assessment* branch as shown in the figure 1.2 deals with many cyber security related topics. Under the *vulnerability scan* sub-branch [22], security engineers are scanning their systems periodically or continuously to locate non-patched already known vulnerability or discover new zero-day vulnerabilities. This happens generally after that the software is deployed within the information system. The sub-branch *source code scan* means scanning the software source (white-box) or binaries (black-box) to discover security vulnerabilities early in the SDLC (Software Development Life Cycle) [23] and helps the developing team to patch it before the software is deployed. This will help to reduce the maintenance cost and avoid damage caused by an adversary (internal or external) successful exploit.

Security Vulnerability:

What's a security vulnerability? [24] Most people think this would be an easy question to answer, but in fact it turns out not to be. A security vulnerability is

a weakness in a system that could allow an attacker to compromise the integrity, availability, or confidentiality of that system. Security vulnerabilities involve inadvertent weaknesses; by-design weaknesses may sometimes occur in a product, but these are not security vulnerabilities. For example, the choice to implement a 40-bit cipher in a product would not constitute a security vulnerability, even though the protection it provides would be inadequate for some purposes. In contrast, an implementation error that inadvertently caused a 256-bit cipher to discard half the bits in the key would be a security vulnerability. Integrity refers to the trustworthiness of a resource. An attacker that exploits a weakness in a system to modify it silently and without authorization is compromising the integrity of that product. Availability refers to the possibility to access a resource. An attacker that exploits a weakness in a system, denying appropriate user access to it, is compromising the availability of that product. Confidentiality refers to limiting access to information on a resource to authorized people. An attacker that exploits a weakness in a system to access non-public information is compromising the confidentiality of that product. As we notice, integrity, availability, and confidentiality are the three main goals for security. If one or more of these three elements lacks, there is a security vulnerability. A single security vulnerability can compromise one or all these elements at the same time. For instance, an information disclosure vulnerability would compromise the confidentiality of a product, while a remote code execution vulnerability would compromise its integrity, availability, and confidentiality.

Exploitable vulnerability:
Exploitable vulnerability is the intersection of three elements: a security vulnerability, attacker access to the flaw, and attacker capability to exploit the flaw. To exploit a vulnerability, an attacker must have at least one applicable tool or technique that can reach a system weakness [25].

1.3 Causes of security vulnerabilities

Unfortunately, there is no single source of security vulnerabilities. Vulnerabilities occur at every stage of the SDLC (Software Development Life Cycle). At the development stage, these vulnerabilities may occur at every location in the code even in code located at a branch that is executed once in a decade. Worse, the most devastating vulnerabilities are those occurring in software security organs under very specific condition. A vulnerability can stay dormant for decades until it detonates and burns everything that was on its path. For example the Heart bleed [26] that affected the widely used cryptographic library OpenSSL stayed in the code for at least a decade before being publicly acknowledged and patched. The following paragraph summarizes the most important security vulnerability sources and causes:

- *Insecure design*:

  A lot of software designers do not think the security at the beginning. This a natural human bias where the designer wants to have a working software and forget about its security. The other cause of this behavior is that security is contrasted with the usability i.e. the more your software is secure, the less it is easily usable and vice versa. Designers taking this trade-off generally sacrifices the security against the usability. For example, the major Internet protocols were designed with very little security in mind and aimed usability [27].

- *Inherently insecure languages*:

  Developing software using some languages makes it more probable to commit a mistake leading to a vulnerability. This is caused by the language's specification complexity and/or ambiguity. C language is an example of a language where it is easy to commit an error [28] leading to a security vulnerability. Programming language choice is sometimes a good starting point to eliminate some kinds of vulnerabilities. The chapter 2 gives a clear view on how a programming language can let the developer commit fatal errors.

- *Errors and failures*:

  A copy-paste error [29], typing 0 instead of 1, syntactic error in variable names and all other mistakes that a human can possibly commit may be sources of a security vulnerability. Of course, not all mistakes will lead to vulnerabilities. These mistakes may lead to *bugs* that are the parent class of security vulnerabilities.

- *Poor testing*:

  Every software need to be tested and validated before being deployed [30]. A poor testing methodology can miss trivial security holes. Exploiting security breaches on sloppily tested software may be very easy and practically with no cost for an adversary. On the other hand, well-tested software may make the cost of a successful attack so high pushing the attacker to abandon that well-tested point and search for other attacking points.

- *Deliberate*:

  In this case, the security vulnerability is seen as a hidden software feature inserted by malicious insider [31]. This kind of vulnerabilities can be very difficult to eradicate especially if the insider is highly skilled and determined.

1.4 High level security measures

To eradicate security vulnerability in software different actions and decisions must be taken. These actions and decisions concern every stage of SDLC and every part of the organization developing or using that software. In the following, we present some measures that may contribute to improve a software security [30]:

- *Good design practice*:

  Think the security at the beginning of the design. Organizations must have guidelines related to security that must be applied for every software design.

- *Language choice*:

  When it is possible, developers should use programming languages known to

cause the least impact on security. For example, in some cases, *Java* language can be a good candidate to build secure software. Functional languages such as *Ocaml* can also be used to produce secure software.

- *Coding standard and rules*:

  When it is not possible to choose a secure language. It is still possible to produce a secure code using insecure language. This is achieved by having coding rules such as *CERT-C coding rules* [32] or the *MISRA C* [33]. These rules forbid the usage of known deprecated or dangerous code constructions and enforce the developer to follow good development practices (variable naming, code commenting, code structure, etc.).

- *Software testing*:

  Testing is important to ensure that the software implementation complies with the design [30]. This can be done while developing i.e. unit testing or at the end of the development, i.e. integration testing. Software testing may be accomplished manually or assisted with various tools and frameworks. This technique can be very useful to find obvious security vulnerabilities but it is inefficient when dealing with complex and large software.

- *Software analysis*:

  In this technique, the software codes or binaries are analyzed without being executed, this is the static analysis or by being executed and monitored this is the dynamic analysis. These techniques have shown good results specially when dealing with security vulnerabilities. More details on the *static analysis technique* will be given in the chapter 3.

Summary

The security of computers and systems is a complex domain. This domain is vast and has different branches. One of these branches is the software security. The software insecurity comes from different causes. All of these causes must

be dealt with using the appropriate means. One of the means used to improve the software security is *static analysis*. Static analysis is a technique to analyze software and find security vulnerabilities within it without actually executing it [34]. In the following chapter, we present how this technique work and how it can be implemented and evaluated.

Chapter 2

Source of vulnerabilities in C language

Introduction

"Better be ignorant of a matter than half know it."

Publilius Syrus.

Developers using the C language can write functioning and correctly behaving applications by knowing only part of C language specifications and semantics. By diving into this language specification we are able to show its complexity. This complexity can be a source of dangerous coding habits that can lead to disasters specially when related to critical systems (industrial, banking, health care, etc.). In this chapter we will present an analysis of some of these dangers, how it causes harm and briefly how it can be detected and prevented. We will focus on safety and security issues. Safety issues concerns availability and resilience, i.e. there is no safety issues when the program does not crash or misbehave of its own. The security issues concerns resistance to an attacker pushing a program to behave in a manner he wants.

2.1 C language definition

A programming language generally consist of instructions for a computer. The earliest known programmable machine was the automatic flute player described in the 9th century by the brothers Musa in Baghdad, during the Islamic Golden Age [35]. From the early 1800s, *programs* were used to direct the behavior of machines such as Jacquard looms and player pianos [36]. Thousands of different programming languages have been created, mainly in the computer field, and many more still are being created every year. Many programming languages require computation to be specified in an imperative form (i.e., as a sequence of operations to perform) while other languages use other forms of program specification such as the declarative form (i.e. the desired result is specified, not how to achieve it).
C language is a general-purpose, imperative computer programming language, supporting structured programming, lexical variable scope and recursion, while a static type system prevents many unintended operations. C was originally developed by Dennis Ritchie between 1969 and 1973 at Bell Labs [37], and used to reimplement the Unix operating system. It has since become one of the most widely used programming languages of all time [38]. C compilers from various vendors are available for the majority of existing computer architectures and operating systems. C has been standardized by the ANSI (American National Standards Institute) since 1989 (see ANSI C [39]) and subsequently by the International Organization for Standardization (ISO). C is an imperative procedural language. It was designed to be compiled using a relatively straightforward compiler, to provide low-level access to memory, to provide language constructs that map efficiently to machine instructions, and to require minimal run-time support. Despite its low-level capabilities, the language was designed to encourage cross-platform programming. A standards-compliant and portably written C program can be compiled for a very wide variety of computer platforms and operating systems with few changes to its source code. The language has become available on a very wide range of platforms, from embedded micro controllers to supercomputers.

The specification of the C language is complex and developers know rarely the deep semantics of the code they are writing. This makes them more exposed to commit mistakes with a devastating effect on application's security.
C compilers are available for large number of different OS (Operating Systems) and platforms [40]. The C language has different variants called by the developer community *flavors*:

- **ANSI C**: this is the basic and central C language specification. It is available in the standard ISO/IEC 9899:TC2 (named C99) [39].
- **POSIX C**: this is an extension of the ANSI C, it is documented in IEEE Standard 1003.1 [41], this C flavor is widely implemented in Unix Operating Systems.
- **GNU C**: also called LINUX C. This is another extension implemented by the GNU C library [42] widely used in Linux systems [43].
- **WINDOWS C**: it is implemented by Visual Studio (Microsoft C/C++ compiler) and documented on the Microsoft MSDN platform [44].

To show the complexity of some C language constructs we will describe in the following three families of functions: *formatted I/O functions*, *command and program execution functions* and *memory manipulation functions*.

2.2 Formatted output functions in C language

A formatted input/output function is a special kind of functions that takes a variable number of arguments, one special argument is called format string. In the case of output function the format is used to convert primitive data types in a human readable string representation and writes it to the output argument (file stream, console output, buffer). When used with input functions the format will guide how the input stream must be parsed and written to the given arguments. This study will focus on giving an in depth study of the specification of these

functions in C language.

All the information that will be given in this section is extracted from the specification available in the ANSI C standard ISO/IEC 9899:TC2 (named C99)[39] under the paragraph **(7.19.6 Formatted input/output functions)**. The figure 2.1 lists the formatted output functions signature declarations.

```
1     // HEADER : <stdio.h>
2     int fprintf(FILE * restrict stream, const char * restrict format,
      ...);
3     int printf(const char * restrict format, ...);
4     int snprintf(char * restrict s,
5         size_t n, const char * restrict format, ...);
6     int sprintf(char * restrict s, const char * restrict format, ...)
     ;
7     int vfprintf(FILE * restrict stream,
8         const char * restrict format, va_list arg);
9     int vprintf(const char * restrict format, va_list arg);
10    int vsnprintf(char * restrict s, size_t n,
11        const char * restrict format, va_list arg);
12    int vsprintf(char * restrict s, const char *
13        restrict format, va_list arg);
14
15    // HEADER <wchar.h>
16    int fwprintf(FILE * restrict stream,const wchar_t * restrict
     format, ...);
17    int swprintf(wchar_t * restrict s, size_t n,
18        const wchar_t * restrict format, ...);
19    int vswprintf(wchar_t * restrict s,
20        size_t n,const wchar_t * restrict format, va_list arg);
21    int vwprintf(const wchar_t * restrict format,va_list arg);
22    int wprintf(const wchar_t * restrict format, ...);
23    int vfwprintf(FILE * restrict stream,
24        const wchar_t * restrict format, va_list arg);
25
26
```

FIGURE 2.1: List of output formatted functions.

According to the specification of these functions the most interesting points are:

- All of these functions have a *format* argument of type *const char ** or *const wchar_t **.
- The difference between fprintf and printf is that the former writes to a FILE descriptor and the latter writes to the standard output "STDOUT".

- "vprintf" and other functions prefixed with "v" take as an argument a va_list and not a variable number of arguments. va_list is a special primitive type defined in the ANSI C to carry an arguments list of variable size.

- "sprintf", "snprintf", "vsnprintf", "vsprintf", "swprintf", "vswprintf" write their outputs to the buffer pointed by the given the first parameter of type "*char **".

- "wprintf" and other "w" prefixed functions present in *wchar.h* have exactly the same semantics except that they take wchar_t (wide character)[1] format string as parameter and their output is also wchar_t stream.

In the following we discuss the structure of the output format string, the usage problems with this family of functions and we give information on how to check and report issues.

2.2.1 Output format string structure

The format argument has a very precise syntax and semantics. Literally it is specified as the following: "*The format is composed of zero or more directives: ordinary multi-byte characters (not %), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments, converting them, if applicable.*" [39].

The conversion specification (most important part) is described in detail in the standard [39]. Each conversion specification is introduced by the character %. After the %, we have the following parts (ordered as introduced in the figure 2.2):

- **flag** *(optional)*: acceptable characters are ['+','0','#','-',' ']. If misused they can lead to "undefined behavior" as it will be explained in the next paragraph.

[1] wide character is special C primitive type that is used to represent characters from non-ASCII alphabets that need more than 1 byte (2 bytes or 4 bytes)

format = ((TXT)(CS)*)+

- TXT: ordinary multi-byte characters (possibly empty).
- CS: Conversion specification, where the first character is a '%'

CS = %[flag][width][precision][length_modifier]conversion_specifier

all parts between brackets are optional only the last part "conversion_specifier" is required

FIGURE 2.2: Output function format argument structure

- **width** *(optional)*: If the converted value has fewer characters than the field width, it is padded with spaces (to the field width). Width could be a valid decimal or '*' , if width == '*' then an argument of type *int* is consumed from the argument list.
- **precision** *(optional)*: that gives the minimum number of digits to appear for the 'd', 'i', 'o', 'u', 'x', and 'X' conversions. Precision could be a valid decimal preceded by '.' or '.*' , if precision == '.*' then an argument of type *int* is consumed from the argument list.
- **length modifier** *(optional)*: that specifies the size of the argument, the possible values are : ['h',"hh",'l',"ll",'j','z','t','L']. These are useful to specify the length of type. For example: "%hd" must be used with "signed short" and "%Lf" with a "double" and not a float.
- **conversion specifier** *(required)* : a character that specifies the type of conversion to be applied, the possible values are : ['d', 'i', 'o', 'u', 'x', 'X', 'f', 'F', 'e', 'E', 'g', 'G', 'a', 'A', 'c', 's', 'p', 'n']. the standard specifies for each conversion specification the precise expected type. The table 2.1 gives a simplified type matching for each conversion specifier.

The following are examples of valid and bad output formats:

Conversion specifier	Expected type
‘d’,‘i’,‘o’,‘u’,‘x’,‘X’	(un)signed int.
‘f’,‘F’,‘e’,‘E’,‘g’,‘G’,‘a’,‘A’	float (double, when prefixed with ‘L’ .)
‘c’	a single character.
‘s’	a pointer to a null-terminated string of ’char’.
‘p’	a “* void”
‘n’	“* int”,The argument shall be a valid integer pointer

TABLE 2.1: Format conversion type matching.

- valid formats:

```
1  "you have %d points"
2  "%s %s!"
3  "Total: %10d"
4  "%.10d"
5  "rate : %.3f%%"     // note that %% means the ASCII symbol '%'
6  "%\#30.30lld"
```

- bad formats:

```
1   "%ll34d"  // the correct order is not respected.
2   "%10.10k"   // unknown conversion specifier 'k'.
```

We give in the next section the usage problems of this kind of functions.

2.2.2 Output functions usage problems

Formatted output functions must be used with care because many errors may arise. The problems caused by a misused call to *printf* family functions can be classified into two categories:

- Safety problems

  These are all the problems caused by the reliance on undefined behaviors

described in the ANSI C standard [45]. In the case of formatted output the standard gives explicitly cases that lead to undefined behaviors:

- Format string value:
  - the format must parse correctly according to the format detailed above.
  - the flag '#' is only to use with these conversion modifiers : [*oxXaAeEfFgG*] (not with [*diucspn*]).
  - the precision field is only to use with these conversion modifiers : [*diouxXaAeEfFs*] (not with : [*gGcpn*]).
  - the flag '0' is only to use with these conversion modifiers : [*diouxXaAeEfFgG*] (not with : [*scpn*])
  - if the length modifier is in [*h,hh,ll,z,t,j*] then conversion modifier cannot be in [*aAeEfFgGscp*]
  - if the length modifier is '*l*' then conversion modifier cannot be in [*p*]
  - if the length modifier is '*L*' then conversion modifier cannot be in [*diuoXxncps*]
- Argument type mismatch :

  As we see above, each conversion specifier expects a special C primitive type, any mismatch leads to an undefined behavior of the application.
- Sprintf buffer related problems:

  The functions "sprintf", "vsprintf", "snprintf", "wsprintf" write their output to a fixed length array, leading to an out-of-bounds write in the case where the output size is greater than the given buffer size.

- Security problems

  At the difference with safety issues, the application is supposed to be a target of an external malicious user aiming to exploit an exposed flaw to breach its Confidentiality, Integrity, or Availability. There will be a security issue when an external user is able to control completely or partially a format string value as stated in the "CWE-134: Uncontrolled Format String" [46].

We will introduce later some additional concepts such as attack surface, attack vectors, input vectors, variable dependency, etc. to have a deep understanding of user inputs and their effect on the security of an application.

2.2.3 Checking and reporting issues

From the description of safety security issues one could dress a list of checks to be performed. For the reporting, each check will be attributed the most corresponding CWE (Common Weakness Enumeration) identifier [21]. CWE naming will help an application auditor to understand the emitted warnings with more precision. The table 2.2 contains examples of checks to be performed on output formatted function calls. A detailed checking and reporting of this type of vulnerabilities is detailed in the chapter 4.

Problem	Check	CWE
Format string value	check that no undefined behavior could be triggered and the format parses correctly.	CWE-758: Reliance on Undefined, Unspecified, or Implementation-Defined Behavior.
Argument type mismatch	check that the given argument at the given order matches the given format.	CWE-686: Function Call With Incorrect Argument Type, CWE-685: Function Call With Incorrect Number of Arguments, CWE-683: Function Call With Incorrect Order of Arguments.
Buffer overflow	Check that the format and the given arguments do not generate an output with a size greater than the size of the given buffer.	CWE-787: Out-of-bounds Write.

TABLE 2.2: Format output functions checking.

```
1         //HEADER <stdio.h>
2   int fscanf(FILE * restrict stream,const char * restrict format,
     ...);
3   int scanf(const char * restrict format, ...);
4   int sscanf(const char * restrict s,const char * restrict format,
     ...);
5   int vfscanf(FILE * restrict stream,
6             const char * restrict format, va_list g);
7   int vscanf(const char * restrict format, va_list arg);
8     int vsscanf(const char * restrict s,const char * restrict format,
      va_list arg);
9
10  //HEADER : <wchar.h>
11  int fwscanf(FILE * restrict stream,
12            const wchar_t * restrict   format, ...);
13  int swscanf(const wchar_t * restrict s,
14            const wchar_t * restrict format, ...);
15  int vfwscanf(FILE * restrict stream,
16            const wchar_t * restrict format, va_list arg);
17  int vswscanf(const wchar_t * restrict s,
18            const wchar_t * restrict format, va_list arg)
19  int vwscanf(const wchar_t * restrict format,va_list arg);
20  int wscanf(const wchar_t * restrict format, ...);
21
```

FIGURE 2.3: List of input formatted functions.

2.3 Input formatted functions in C language

The other category of formatted functions is input functions. These category of functions reads data from input streams and parses it according to the given format. The figure 2.3 lists the input formatted functions declarations of the ANSI C.

According to the specification of these functions the most interesting points are:

- All of these functions have a "format" argument, this is the main cause that they could cause a format string vulnerability.
- The difference between *fscanf* and *scanf* is that the former reads from a FILE descriptor and the latter reads from the standard input "STDIN".

- "vscanf" and other functions prefixed with "v" take as an argument a va_list and not a variable number of arguments. va_list is a special type defined in the C99 standard.
- "sscanf" reads from a "char *" (the fist parameter), and writes values read to its arguments according to the given format.
- "wscanf" and other "w" prefixed functions present in the *wchar.h* have exactly the same semantics expect that it takes wchar_t (wide character) format string as parameter.

In the following we discuss the structure of the input format string, the usage problems with this family of functions and we give information on how to check and report these issues.

2.3.1 Input functions format string structure

The format argument has a very precise syntax and semantics. Literally it is specified as the following: *"The format is composed of zero or more directives: one or more white-space characters, an ordinary multi-byte character (neither % nor a white-space character), or a conversion specification. Each conversion specification is introduced by the character %. After the %, the following appear in sequence"* [39]. The roles and semantics of input format elements as detailed in the figure 2.4 are:

- **assignment-suppressing** *(optional)*: '*' is the only accepted character. This allows to skip an input of the type specified by the conversion specifier. This has an effect on argument type matching and could lead to an undefined behavior as explained in the next paragraph.
- **width** *(optional)*: decimal integer greater than zero that specifies the maximum field width (not the same as the width field used with printf format where the value can be '*').

format = ((TXT)(CS)*)+

- TXT: ordinary multi-byte characters (possibly empty).
- CS: Conversion specification, where the fist character is a '%'

CS = %[assignment-suppressing][width][length_modifier]conversion_specifier

all parts between brackets are optional only the last part "conversion_specifier" is required

FIGURE 2.4: Input function format argument structure

Conversion specifier	expected type
'd','i','o','u','x','X'	a pointer to (un)signed int.
'f','F','e','E','g','G','a', 'A'	a pointer to float (double, when prefixed with 'L' length modifiers.)
'c'	a pointer to single character.
's'	a pointer to string of 'char', this may be dangerous and could lead to buffer overflow.
'p'	a pointer to "* void"
'n'	a pointer to "int". No input is consumed. The corresponding argument shall be a pointer to signed integer
[[a-z]]	a pointer to string of 'char'.

TABLE 2.3: Input format conversion type matching

- **length modifier** *(optional)*: that specifies the size of the receiving object. Possible values are : ['h', 'hh', 'l', 'll', 'j', 'z', 't', 'L']. Exactly the same possible values and semantic compared to printf format.
- **conversion specifier** *(required)*: specifies the type of conversion to be applied. Possible values are : ['d', 'i', 'o', 'u', 'x', 'X', 'f', 'F', 'e', 'E', 'g', 'G', 'a', 'A', 'c', 's', 'p', 'n'] and a set of acceptable characters enclosed between "[]" (e.g. : [0123456789]).

The table 2.3 presents a simplified type matching for each conversion specifier.

The following are examples of valid and bad formats:

- valid formats:

```
1  "%d %s"
2  "%s %s!"
3  "Total: %10d"
4  "%10d"
5  "rate : %.3f%%"     // note that %% means the ASCII symbol '%'
6  "%\#30lld"
7  "%[0123456789]"
```

- bad formats:

```
1   "%ll34d"  // the correct order is not respected.
2   "%10t"   // unknown conversion specifier 't'.
```

2.3.2 Input formatted functions usage problems

The problems caused by a misused call to input family functions are classified into two categories:

- Safety problems

  These are all the problems caused by the reliance on undefined behaviors [45] described in the ANSI C standard. In the case of formatted input functions the standard gives explicitly cases that lead to undefined behaviors:

  - Format string value:
    - the format must parse correctly according to the format detailed above.
    - if the length modifier is in [*'h', "hh", "ll", 'z', 't', 'j'*] then conversion modifier cannot be in [*'a', 'A', 'e', 'E', 'f', 'F', 'g', 'G', 's', 'c', 'p'*]
    - if the length modifier is *'l'* then conversion modifier cannot be in [*'p'*]

  - if the length modifier is *'L'* then conversion modifier cannot be in [*'d','i','u','o','X','x','n','c','p','s'*]
  - If the conversion specification includes an assignment-suppressing character or a field width and the conversion specifier is *'n'*.

- Argument type mismatch:
  As we see above each conversion specifier expects a special C primitive type, any mismatch leads to an undefined behavior of the application.
- "sscanf" buffer related problems:
  The functions "sscanf", "vsscanf", "swscanf", "vswscanf" read their input from a fixed length array. Leading to an out-of-bounds read in the case where the input buffer size is smaller than the size expected by the given format.
- General input function buffer problems:
  Due to the fact that an input function is reading data from user input and writing it to fixed size variables, the risk of out-of-bounds write is very high with all input functions family. For example:

```
1       char buf[128];
2       scanf("%s",buf); // this call to scanf can lead to
    stack overflow.
3
```

- Security problems
  Security issues will arise if an external user has a complete or partial control of a format string value. Even that the input family functions are not stated in the "CWE-134: Uncontrolled Format String" [46], but due to their specification a user who controls the format value will be surely able to interfere with the security (Confidentiality, Integrity, Availability) aspects of the targeted application. Same remark as for output family security issues: to go deeper some new concepts are needed, they will be detailed later.

Problem	Check	CWE
scanf buffer overflow	Check that the size of received data is not greater than the the size of the corresponding argument.	CWE-787: Out-of-bounds Write
sscanf buffer problems	Check the the size of buffer that these function read from is greater than the one expected by the given format.	CWE-125: Out-of-bounds Read

TABLE 2.4: Format input functions checking.

2.3.3 Checking and reporting issues

Some checks that must be performed are the same as for the output family. The first checks are on the value of format string that it parses correctly and does not lead to undefined behaviors. Also we have the check on type mismatch that must be performed. Additionally the table 2.4 lists the new checks to be performed. A detailed checking and reporting of this type of vulnerabilities is presented in the chapter 4.

2.4 POSIX formatted functions

2.4.1 Format string structure difference

The format string structure in POSIX is the same as in ANSI C (C99) and it adds the following extensions:

- the "%n$" new format structure:
  The POSIX adds the following: *"the conversion specifier character '%' can be replaced by the sequence '%n$', where n is a decimal integer in the range [1,NL_ARGMAX], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments*

*in an order appropriate to specific languages ... The format can contain either numbered argument conversion specifications (that is, '%n$' and '*m$'), or unnumbered argument conversion specifications (that is, % and *), but not both"*[47]. The value of fields "width" and "precision" in the format structure can be '*m$' in place of just '*' and so will take the value of the m^{th} argument at runtime.

- These new conversion specifiers are introduced:
  - 'C' : Equivalent to "lc" (the length modifier 'l' concatenated with the conversion specifier 'c')
  - 'S' : Equivalent to "ls" (the length modifier 'l' concatenated with the conversion specifier 's')
- Examples:
  - valid formats:

```
1  "%d %s"          // valid in POSIX because it is a valid
     C99 format
2  "%2$d %1$d "     // valid POSIX format using the new %n$
     notation ,
3                   // it will print the 2nd argument
4                   //in the 1st place and the 1st at the 2nd
     place.
5  "%S %S"    // valid format using the new conversion
     specifier 'S'.
```

  - bad formats:

```
1  "%d %1$s"          // mixed %n$ and % are not permitted.
2  "%k "           // invalid C99 format , is also invalid in
     POSIX.
```

2.4.2 List of additional functions

The POSIX extension adds the following functions:

```
1     //HEADER <syslog.h>
2     void syslog(int priority, const char *message, ... /* arguments
    */);
```

This function is used to write information into the OS logs. the argument message of type *'const char *'* contains the format string used when writing data (the remaining arguments). Format sting passed to the function has the same POSIX structure defined above, except that the additional conversion specification %m shall be recognized; it shall convert no arguments, shall cause the output of the error message string associated with the value of *errno* on entry to *syslog()* [48].

2.4.3 Safety and security issues

POSIX formatted I/O functions are an extension of the ANSI C formatted functions. All safety and security issues mentioned above are present. Due to the extension of format structure we have also the following safety issues:

- New undefined behavior with the new "%n$' notation: The format can contain either numbered argument conversion specifications ("%n$" or "*m$"), or unnumbered argument conversion specifications (% or *), but not both.
- Argument type matching: The argument matching must be adapted in case where the "%n$" notation is used.

2.4.4 Checking and reporting issues

To check and report POSIX formatted I/O functions safety and security issues, we perform the same checks as in the ANSI C case, with little modification to include the new safety issues found.

2.5 GNU/Linux formatted functions

2.5.1 Format string structure difference

The Linux format string implements POSIX format structure. We can notice no visible difference on the MAN (MANual) page [49] (Linux manual pages describing standard library functions) of *printf* or *scanf* functions families.

2.5.2 List of additional functions

The Linux extension is bigger. It adds many new useful functions widely used in different GNU/Linux applications. The listing in figure 2.5 contains some of these added functions.

```
1     //HEADER :<err.h>
2 void err(int __status, const char *__format, ...);
3 void error(int, int, const char *, ...);
4 void errx(int __status, const char *__format, ...);
5 void warn(const char *__format, ...);
6 void warnx(const char *__format, ...);
7
8 //HEADER : <stdio.h>
9 int asprintf(char ** restrict ptr, const char * restrict format, ...)
    ;
10 int dprintf(int fd, const char *format, ...);
11 int vdprintf(int fd, const char *format, va_list ap);
12
13 //HEADER : <linux/printk.h>
14 int printk(const char *fmt, ...);
15
16
```

FIGURE 2.5: List of Linux format functions

All these functions has the same role as the *printf* function. All have a format string parameter. Some special cases:

- asprintf : has the same role as ANSI C sprintf, but it allocates dynamically its output buffer.

- dprintf : have the same role as *printf* but write their output to a file descriptor *fd*.

2.5.3 Safety and security issues

Linux formatted I/O functions does not modify the syntax of semantics of format argument compared the the POSIX specifications. Because of that this family of functions has the same issues as POSIX functions.

2.5.4 Checking and reporting issues

The checks that can be performed on GNU/Linux formatted output functions are the same as for POSIX C functions. This is due to the fact that the GNU/Linux extension does not add a lot to the specification and it adds only some new functions.

2.6 Windows formatted function

The following information is based on the online MSDN (MicroSoft Developer Network) documentation [50].

2.6.1 Format string structure difference

Microsoft Visual Studio C compiler does not implement entirely the ANSI C. Formatted I/O functions specification has the following difference in comparison with ANSI C specification:

- For the "length modifier" part we have these possible values: *['h','l', "ll", 'w', 'I', "I32", "I64"]*.

- The “width” and “precision” arguments have no difference with the ANSI C specification.
- For “conversion specifier” we have the one new value: ‘Z’ that must be matched with the type “ANSI_STRING”[2]
- The ‘n’ conversion specifier is disabled by default for security reason.
- for scanf family we have the same format structure as in ANSI C scanf family except that the length modifier has this list of possible values: *[‘h’,‘l’, “ll”, “I64”, ‘L’]* and the ‘a’, ‘A’] are not possible values for the conversion specifier.

2.6.2 List of additional functions

The Windows extension adds many new functions widely used. The listing in figure 2.6 contains some of these added functions.

[2]ANSI_STRING is C structure defined in ”Ntdef.h” used only in driver debugging functions that use a format specification.

```
1  //HEADER : <stdio.h>
2 int _printf_l(const char *format, locale_t locale [,argument]... );
3 int _wprintf_l(const wchar_t *format, locale_t locale [, argument]...
     );
4 int _scanf_l(const char *format, locale_t locale [, argument]...  );
5 int _wscanf_l(const wchar_t *format, locale_t locale [, argument]...
     );
6 int _sprintf_l(char *buffer, const char *format, locale_t locale [,
    argument] ... );
7 int _swprintf_l(wchar_t *buffer, size_t count, const wchar_t *format,
     locale_t locale [, argument] ... );
8 int __swprintf_l(wchar_t *buffer, const wchar_t *format, locale_t
    locale [, argument] ... );
9 int _sscanf_l(const char *buffer, const char *format,locale_t locale
    [, argument ] ... );
10 int _swscanf_l(const wchar_t *buffer, const wchar_t *format, locale_t
     locale [, argument ] ... );
11 int _vsprintf_l(char *buffer, const char *format, locale_t locale,
    va_list argptr );
12 int _vswprintf_l(wchar_t *buffer, size_t count, const wchar_t *format
    , locale_t locale, va_list argptr );
13 int __vswprintf_l(wchar_t *buffer, const wchar_t *format, locale_t
    locale, va_list argptr );
14 int _vfprintf_l(FILE *stream, const char *format, locale_t locale,
    va_list argptr );
15 int _vfwprintf_l(FILE *stream, const wchar_t *format, locale_t locale
    , va_list argptr );
16 int _fprintf_l(FILE *stream, const char *format, locale_t locale [,
    argument ]...);
17 int _fwprintf_l(FILE *stream, const wchar_t *format, locale_t locale
    [, argument ]...);
18 int _fscanf_l(FILE *stream, const char *format, locale_t locale [,
    argument ]... );
19 int _fwscanf_l(FILE *stream, const wchar_t *format, locale_t locale
    [, argument ]... );
20 int _vprintf_l(const char *format, locale_t locale, va_list argptr );
21
```

FIGURE 2.6: List of Windows I/O formatted functions.

2.6.3 Safety and security issues

The analysis of the specification [50] of formatted functions leads to the conclusion that all safety and security issues stated for ANSI C remain correct due to close similarity of the two specifications. The specification of Windows C language standard library is available online [44]. This library is available on all Microsoft Windows system and supported by the Visual studio IDE (Integrated Development Environment).

2.6.4 Checking and reporting issues

Methods and techniques to check and report issues of using Windows formatted functions are the same as checks for ANSI C formatted functions. There will be some difference on how to perform it operationally due to the newly added functions.

2.7 Command and program execution functions

This section provides a security and safety overview of functions doing command execution within a program. This feature exists in almost all major programming language. This study will focus on C language. Functions implementing this feature allow to launch a command given as string argument by using the system default shell. It is also considered that functions that launch another program are also a sort of command execution feature.

Command execution is implemented in the C language in its different flavors. This paragraph focuses on the three most used C flavors: ANSI C (C99)[39], POSIX C[41] and GNU C[42].

According to the standard C language specification [39], we have only one function that could lead to command injection vulnerability:

```
1  #include <stdlib.h>
2  int system(const char *string);
```

In the paragraph about these functions in C99 [39] we have: "*If string is a null pointer, the "system" function determines whether the host environment has a command processor. If string is not a null pointer, the system function passes the string pointed to by string to that command processor to be executed in a manner which the implementation shall document; this might then cause the program calling system to behave in a non-conforming manner or to terminate*".

For example:

```
1  #include <stdlib.h>
2  int main(){
3  int r = system("ls -a"); //will print the content of the working
    directory.
4  }
```

In addition to the function "system" function, POSIX and LINUX extensions define the following new functions:

```
1 //POSIX
2 int execl(const char *path, const char *arg, ...);
3 int execlp(const char *file, const char *arg, ...);
4 int execle(const char *path, const char *arg,..., char * const envp
    []);
5 int execv(const char *path, char *const argv[]);
6 int execvp(const char *file, char *const argv[]);
7 int execvpe(const char *file, char *const argv[],char *const envp[]);
8 int fexecve(int fd, char *const argv[], char *const envp[]);
9 FILE *popen(const char *, const char *);
10 int posix_spawn(pid_t *restrict, const char *restrict,
11           const posix_spawn_file_actions_t *,
12           const posix_spawnattr_t *restrict,
13           char *const [restrict], char *const [restrict]);
```

```
14 int posix_spawnp(pid_t *restrict, const char *restrict,
15           const posix_spawn_file_actions_t *,
16           const posix_spawnattr_t *restrict,
17           char *const [restrict], char *const [restrict]);
18
19 //LINUX
20 int execvpe(const char *file, char *const argv[],
21     char *const envp[]);
```

The listed *execl** functions load and launch a program that replaces the current process image with a new process image. The specification of these functions gives conventions on how these functions should be called. For example, to correctly call *execl*, the first argument after *path* should be the same as the program name and the last should be *null*.

2.7.1 Safety and security issues

From the specification summarized above, the following safety and security issues arise:

- Safety problems

  There is no undefined behavior that can be caused by a call to "system", but the standard states that the behavior of this function is implementation-defined. This must be taken in consideration when dealing with special C99 implementation. For the *execl** functions, the calling conventions should be respected or the application could have an undefined behavior.

- Security problems

  Any call to "system" is potentially dangerous and developers are advised not to use it. For example, the code listing in figure 2.7 make a call to "system" in an insecure way. This vulnerability will be later detailed in the chapter 4. Example of vulnerable command execution function calls are:

– An external user has a complete or a partial control of the command argument: this vulnerability is known as *CWE-78: Improper Neutralization of Special Elements used in an OS Command "OS Command Injection"*[51].

– The command value is constant, but the attacker can interfere with the command by changing the context of the application (file system, environment variables, etc.).

```
1 ...
2  system("./myprogram"); // this call launches another program
3                         // found in the same working directory.
4                         // An attacker can erase this file and
5                         // replace it his own program.
```

FIGURE 2.7: Vulnerable program launching example

2.8 Memory manipulation functions

It is considered a memory manipulation function any function that deals with a set of unstructured data. So, functions taking as an argument or returning a pointer to an unstructured memory region (char *, void *, int *, etc.) are considered. For example all of *malloc, calloc, memcpy, strcpy, memmove* are functions that manipulate memory, it will be showed later that in Libc [42] (C99 [39], Posix [41], Linux [43]) there are many non-trivial functions that manipulate memory. The figures 2.8 and 2.9 list respectively memory manipulation functions for C99/POSIX and LINUX of the C language standard library.

```
malloc, free, calloc, realloc, fscanf, fgets, sscanf, scanf, gets,
fread, sprintf, snprintf, memcpy, memmove, strcpy, strncpy,
strcat, strncat, memset, mbstowc, wcstombs, strxfrm, strftime,
fwscanf, vfscanf, vfwscanf, fgetws, swscanf, vsscanf, vswscanf,
wscanf, vscanf, vwscanf, vsprtinf, swprintf, vswprintf, vsnprintf,
wmemcpy, wmemmove, wcscat, wcsncat, wmemset, wcscpy, wcsncpy,
wcsxfrm.
```

FIGURE 2.8: Memory related functions of C99 and POSIX

```
aio_read, confstr, fgetws, fmemopen, open_memstream,
open_wmemstream, strfmon, strfmon_l, getline, getdelim,
iconv, memccpy, mmap, munmap, posix_memalign, valloc,
aligned_alloc, memalign, pvalloc, pread, readv, preadv,
readlink, realinkat, recv, recvfrom, recvmsg, strdup,
strndup, strdupa, strndupa, wcpcpy, alloca, asprintf,
vasprintf, bcopy, memfrob, bzero, mempcpy, wmempcpy,
mtrace, sctp_recvmsg, stpcpy.
```

FIGURE 2.9: Memory related functions of Linux

2.8.1 Safety and security issues

It considered as a safety issue, any function call leading to undefined [45], unspecified or implementation-defined behaviors. This is relatively easy to extract, as, generally this is clearly said in the function description. To give more meaning to the found issues one CWE [21] identifier (or more) is attributed for each found vulnerability source. In table 2.5 we give an example of safety issues found in analyzed functions.

A security issue is harder to define precisely. It is considered all function parameters where a direct control from an external agent leads to a safety issue, or a behavior not expected by the developer/designer of the application.

Description	Affected functions	CWEs
Referencing a freed object with a call to "free" or realloc causes an undefined behavior	free, realloc, munmap	Double Free - (415), Use After Free - (416), CWE-758: Reliance on Undefined, Unspecified, or Implementation-Defined Behavior

TABLE 2.5: Memory manipulation safety issue example

Description	Affected functions	CWEs
An external user control the amount of allocated storage (via the "size" parameter in function that allocates memory).	malloc, calloc, realloc, posix_memalign, aligned_alloc, valloc, memalign, pvalloc, open_memstream, getline, getdelim, mmap, strdup, strndup, strdupa, strndupa, alloca, asprintf, vasprintf	CWE789 Uncontrolled Mem Alloc

TABLE 2.6: Memory manipulation security issue example

By analyzing carefully the functions description, different security issues are found. Each security problem is attributed a CWE [21] matching it at best.

The table 2.6 gives examples of found security issues.
The full list of safety and security issues is detailed in the chapter 4. Methods to check and report such issues are also described there.

Summary

The analysis and study of the C language specification and semantics highlighted some security and safety issues in this language. It is shown how it can be difficult to know and remember all good usage rules of standard library functions. Automation of checking and reporting will greatly help developer by doing the hardest part of the task. The next chapter will introduce one category of these

used automatic processes: *static analysis*. And in the chapters 4 and 5 more details will be given on how to design, implement and evaluate tools implementing this technique.

Chapter 3

Static analysis tools

The complexity of programming languages and the growing cost of software vulnerabilities are major concerns of IT (Information Technology) users. These concerns are addressed in different ways using several strategies, methods and techniques. One of these prominent techniques is the code analysis and more precisely static analysis.

Static analysis is a technique used to analyze application code without actually executing it. It helps to find mechanical errors such as buffer overruns, unvalidated input, null dereference, uninitialized data access, code constructs leading to runtime errors, etc. These errors are hard to detect with testing or manual code inspection because they are "non-local" and involve for example uncommon execution paths or they are non deterministic such as race condition errors [52].

The OWASP (Open Web Application Security Project) [53] defines: *"Static Code Analysis (also known as Source Code Analysis) is usually performed as part of a Code Review (also known as white-box testing) and is carried out at the Implementation phase of a Security Development Life Cycle. Static Code Analysis commonly refers to the running of Static Code Analysis tools that attempt to highlight possible vulnerabilities within 'static' (non-running) source code by using techniques such as Taint Analysis and Data Flow Analysis"* [34].

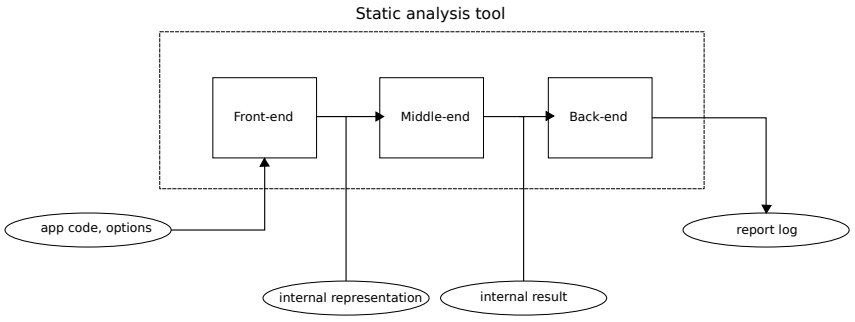

FIGURE 3.1: Static analysis tool architecture

A static analysis tool is a software that performs static analysis and returns a result report. Generally a static analysis tool is structured into three components: front-end, middle-end and the back-end as depicted in the figure 3.1. The front-end parses the application source code and produces an internal representation that will be fed to the next component. The middle-end will do the necessary computation and stores the obtained results that will be reported by the back-end part. This structure is similar to the structure of compilers. This similarity is due to the fact that a compiler is a special case of static analysis tool. A compiler first parses and checks the syntax of the given program and emits warnings if wrong constructions are encountered. After, it builds an internal representation that is translated into native machine code and outputs it in the form of an executable file. Some modern compilers even check for some basic security issues playing the role of security static analysis tool.

3.1 Front-end: parsing and translating

The front-end of a static analyzer is a kind of source code translator that will take an input language and outputs it in another language. The problem of source-to-source translation is widely explored and it is subject of many ongoing researches conducted either by academic and industrial entities. Source translator can be used to manipulate application code to produce a more efficient one as described

in a use case of LLVM (Low Level Virtual Machine) [54]. It can be simply used to produce the AST (Abstract Syntax Tree) which is a representation adapted for doing computations on code structure as in Cil (C intermediate Language) [55] or GCC (GNU Compiler Collection) [56]. In this chapter we will describe and classify the existing solutions for code translation.

The tools, frameworks and projects that exist can be classified into two categories:

- Proprietary:

  This means that, to use such tool a license must be purchased, and perhaps the license clauses forbid its modification and/or redistribution. For example we have: the widely used *EDG (Edison Design Group) framework of C++ and Java language* [57] or the *DMS Software Re-engineering Toolkit* [58]. The table 3.2 lists an extensive list of tools classified as proprietary.

- Open source:

  For this category the code source is open (publicly available). Even that the sources are accessible, reuse it and/or redistribute it could be protected by a specific license (GPL [59], APACHE License [60], LGPL [61], etc.). Cil framework [55] or LLVM suite [62] are examples of open source tools. The table 3.1 lists tools available on-line at the time of writing. We give in this table for each listed tool, its name, its source language, its destination language and a brief description.

In the table 3.1 some of the listed tools are not described as source code translator, but translates a given source code to a supposed more efficient language. For example we have: HPHPc (PHP to C++) [63] which is developed by Facebook Inc to make their servers running faster. LLVM [62] can also be used to produce a more efficient code. The idea of translating interpreted code into native code is very present such as py2c [64] (from Python to C) or Toba [65] (from Java to C).

Tool/Framework name	Source language	Target language	Brief description
CIL	C	C intermediate language	CIL (C Intermediate Language) is a high-level representation along with a set of tools that permit easy analysis and source-to-source transformation of C programs
LLVM	Ada, C, C++, Fortran, Java, Objective-C, or Objective-C++)	C, C++	Yes, you can use LLVM to convert code from any language LLVM supports to C. Note that the generated C code will be very low level (all loops are lowered to gotos, etc) and not very pretty (comments are stripped, original source formatting is totally lost, variables are renamed, expressions are regrouped), so this may not be what you're looking for.
HipHop for PHP (HPHPc)	PHP	C++	HipHop for PHP is a source code transformer for PHP script code. It automatically transforms your PHP source code into highly optimized C++ and then uses g++ to compile it. HipHop executes the source code in a semantically equivalent manner and sacrifices some rarely used features — such as eval() — in exchange for improved performance. HipHop includes a code transformer, a reimplementation of PHP's runtime system, and a rewrite of many common PHP Extensions to take advantage of these performance optimizations.
py2c	Python	C/C++	py2c is a Python to C/C++ translator (converter). Py2c translates Python to pure human-readable C/C++ like what you and me would write which does not have any Python API calls. The generated code can be run without Python installed and does not embed Python.
Gcc plug-ins	C, C++, Java , ADA, GO	C, C++	Compiler plugins (or loadable modules) make it possible for a developer to add new features to the compiler without having to modify the compiler itself.
J2C (eclipse plug-ins).	Java	C++	J2C will convert Java code into hopefully C++ code. It works on source level translating Java source code constructs into their rough equivalents in C++. The output will be reasonably valid C++ code that looks a lot like its Java counterpart and hopefully works mostly the same.
Toba	Java	C	Toba translates Java class files into C source code. This allows the construction of directly executable programs that avoid the overhead of interpretation. Toba deals with stand-alone applications, not applets.
p2c	Pascal	C	p2c converts the computer language Pascal to C which you can then compile with cc or gcc.
Cython	Python	C	This allows the compiler to generate very efficient C code from Cython code. The C code is generated once and then compiles with all major C/C++ compilers in CPython 2.6, 2.7

Continued on next page

Table 3.1 – *Continued from previous page*

Tool/Framework name	Source language	Target language	Brief description
php2cpp	PHP	C++	For good reason, PHP is a very popular language for publishing web sites with dynamic content. Its strong similarity to C makes the syntax easy to learn for many. The built-in equivalents of many standard C library file, date, string, and time functions make it easy to develop quite complex applications.
Perlcc	Perl.	C	The C back-end takes Perl source and generates C source code corresponding to the internal structures that Perl uses to run your program. When the generated C source is compiled and run, it cuts out the time which Perl would have taken to load and parse your program into its internal semi-compiled form.
JCGO	Java	C	JCGO, a Java source to C code (java-to-c) translator, is the software product originally developed by Ivan Maidanski. With JCGO you can compile your Java application to machine native code (a binary executable file) making it run faster, consume less system resources, harder to decompile and easier to deploy. Even more, with JCGO you could also make your Java applications run on a wider range of operating systems, computer systems, embedded devices and programmable controllers.
Jcvm	Java	C	JC is a Java virtual machine implementation that converts class files into C source files using the Soot Java byte-code analysis framework.
Stance java front end	Java	Ocaml structure	The STANCE Java front-end is a front-end (scanner, parser, type checker, and normalizer) for the Java programming language being developed by Gijs Vanspauwen and Bart Jacobs at the DistriNet Research Group at the Department of Computer Science of KU Leuven - University of Leuven
NestedVM	GCC supported languages	java bytecode	NestedVM provides binary translation for Java Bytecode. This is done by having GCC compile to a MIPS binary which is then translated to a Java class file. Hence any application written in C, C++, Fortran, or any other language supported by GCC can be run in 100% pure Java with no source changes.
Analysing the interpreter with the script as input	Perl, Python, Php, Javascript ...	C	Analyze the source code of the interpreter with the script as input could give important results, mainly if the source code of the interpreter is available in C or C++.

TABLE 3.1: List of open source code translators.

The tools listed in the table 3.2 are front-ends (DMS [58], EDG [57]) that can be used to develop other tools. These tools cover a wide variety of languages, OS and architectures. They are generally well maintained and documented. This high

qualities make them expensive and generally not adapted for innovative research conducted with small human and financial resources.

Tool/Framework name	Source language	Target language	Brief description
DMS Software Re-engineering Toolkit	C, C++, Java, Cobol, Perl , PHP...	NA	Semantic Designs offers predefined language front ends (domains) to enable the construction of custom compilers, analysis tools, or source transformation tools, based on first-class infrastructure (DMS) for implementing those custom tools.
Comeau	C++	C	Proprietary tool
JFE	Java	Cil, C	There is, however, an alternative that can reduce development cost and time. The JFE includes a component "CIL Generation" that translates the Java intermediate language (JIL) to the C intermediate language (CIL) used by EDG's C++/C front end. The CIL can then be fed into a code generator (the CIL is a lower-level description than the JIL and thus is easier to generate code for) or, using the C-generating back end provided with the JFE, output as C source code. This code can then be compiled to object code using any C compiler.
C++ Front End	C++	C	Also included: a C-generating back end, which can be used to generate C code for C++ programs
Adatoccpp-translator	Ada	C	Adatoccpptranslator is a converter software which allows C/C++ source files to be generated from Ada83 and Ada95 source files. To run correctly, input files must be compilable and executable. They only can contain a package specification or a package body or a separate unit.

TABLE 3.2: List of proprietary code translators.

In some cases it can be interesting to develop a front-end from scratch as this is done with compilers. A list of compiler builders is provided in the table 3.3. In this table, the tools Bison [66] or BNFC [67] are good examples of parser generators. They can be used to build parsers for an arbitrary programming language by only giving them the language's grammar specification as input.

Front-end can also be used to parse low level code such as binary executable, byte-code or assembly code. An example of frameworks parsing these formats is the angr framework [68].

Tool/Framework name	Source language	Target language	Brief description
BNFC	C, C++, C#, Haskell, Java, Ocaml, XML	C, C++, C#, Haskell, Java, Ocaml, XML	The BNF Converter is a compiler construction tool generating a compiler front-end from a Labeled BNF grammar. It is currently able to generate C, C++, C#, Haskell, Java, and OCaml, as well as XML representations.
ROSE	NA	NA	Developed at Lawrence Livermore National Laboratory (LLNL), ROSE is an open source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale C(C89 and C98), C++(C++98 and C++11), UPC, Fortran (77/95/2003), OpenMP, Java, Python and PHP applications.
newspeak	C, ADA	Newspeak	Newspeak is a simplified programming language, well-suited for the purpose of static analysis. C2newspeak compiles C programs into Newspeak. Ada2newspeak compiles Ada programs into Newspeak.
Antlr	NA	NA	ANTLR (ANother Tool for Language Recognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. It is widely used to build languages, tools, and frameworks. From a grammar, ANTLR generates a parser that can build and walk parse trees.
Elkhound	NA	NA	Elkhound is a parser generator, similar to Bison. The parsers it generates use the Generalized LR (GLR) parsing algorithm. GLR works with any context-free grammar, whereas LR parsers (such as Bison) require grammars to be LALR.
Bison	NA	NA	Bison is a general-purpose parser generator that converts an annotated context-free grammar into a deterministic LR or generalized LR (GLR) parser employing LALR(1) parser tables. As an experimental feature, Bison can also generate IELR(1) or canonical LR(1) parser tables. Once you are proficient with Bison, you can use it to develop a wide range of language parsers, from those used in simple desk calculators to complex programming languages.
JavaCC	NA	NA	Java Compiler Compiler (JavaCC) is the most popular parser generator for use with Java tm applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program
Marpa	NA	NA	Marpa is a parsing algorithm. It is new, but very much based on the prior work of Jay Earley, Joop Leo, John Aycock and R. Nigel Horspool. Marpa is intended to replace, and to go well beyond,recursive descent and the yacc family of parsers.

TABLE 3.3: List of compiler builders.

3.2 Middle-end : computation and detection

The middle-end part of static analysis tool is the part that does the useful computations of the analysis. These computations make use of different techniques and algorithms. In this section we will introduce the major existing techniques.

Early static analyzers used simple *syntactic pattern matching* to locate potentially vulnerable code patterns such as the Linux utility *grep* or FlawFinder. FlawFinder [69] is fundamentally a naive program; it does not even know about the data types of function parameters, and it certainly does not do control flow or data flow analysis. Nevertheless, Flawfinder can be a very useful aid in finding and removing security vulnerabilities. These simple tools have a high rate of false positive and cannot detect complex vulnerability involving the semantics of the code.

Improved tools such as Lint [70], Cppcheck [71] or even some modern compilers search for more complex vulnerabilities such as: variables being used before being set, division by zero, conditions that are constant, and calculations whose result is likely to be outside the range of values representable in the type used. *Data and control flow analysis* [72] combined with heuristic algorithms is the main technique used in these tools. These tools can be more precise than naive tools but can miss complex semantics related vulnerable code.

More advanced static analyzers use *abstract interpretation* [73] to prove the absence of RunTime Errors (RTEs) [74]. We cite as examples: Polyspace [8, 9], Frama-C [10, 11] and Astrée [12]. The idea behind this technique is to compute a superset of the values of every variable of every reachable program point for every possible input.

Another technique used is *symbolic analysis* which is a static analysis method for reasoning about program values that may not be constant. It aims to derive a

precise mathematical characterization of the computations and can be seen as a compiler that translates a program into a different language whereas this language consists of symbolic expressions and symbolic recurrences. C language static analysis tool CBMC [75] is an example of a tool using such techniques.

Another analysis technique is the *concolic execution*. It is a software analysis technique that performs symbolic execution, a technique that treats program variables as symbolic variables, along a concrete execution (testing on particular inputs). It was introduced by Godefroid et al. [15] where it was used to assist random testing to cover a maximum numbers of execution paths. This method is also used by KLEE [76] for their unassisted high coverage testing. In their tool AEG (Automatic Exploit Generation), Avgerinos et al. [77] were interested in automatically generating an exploit by combining source code analysis to find the exploitable vulnerability and binary analysis to produce the exploit. A lot of tools and techniques have been developed by using the concolic execution. Shoshitaishvili et al. [78] describe almost all state of art of these techniques and especially those related to binary code analysis. They implemented the techniques proposed in the open source framework *angr* [68].

The table 3.4 summarizes the major differences between the presented techniques of static analysis middle-end part.

Technique	Speed	Scalability	False positive	True negative
Syntactic analysis	High	High	High	High
Data flow analysis	Medium	Medium	Medium	Medium
Abstract interpretation	Low	Medium	Medium	No
Symbolic execution	Low	Low	No	Low
Concolic execution	Low	Low	No	Low

TABLE 3.4: Static analysis techniques comparison

3.3 Back-end: results collection and reporting

The back-end role is first to collect the results. Then, the results are formatted in the format specified by the tool user. Finally the output results are given in the requested form (text files, web pages, Excel sheet, databases, etc.).
Reporting taxonomy is important and makes it easier for user to understand the reported vulnerability. Different efforts have been made by the IT communities (industrial, academic, governmental) to build homogeneous security vulnerability taxonomy. Notable examples are the CWE (Common Weakness Enumeration) [21] which is an extensive hierarchical collection of security vulnerability definitions. The CVE (Common Vulnerability Exposures) [20] is another widely used security vulnerabilities directory that lists real vulnerabilities discovered in software used worldwide. The CVSS (Common Vulnerability Scoring System) [79] is an open industrial standard for assessing the severity of computer system security vulnerabilities. This scoring standard can be used to sort reported vulnerabilities from the most critical to the least. This helps static analysis tool user to establish the correct priority in dealing with the different reported vulnerabilities.

This taxonomy will be extensively used along all our contributions given in chapters 4, 5 and 6.

3.4 List of tools

The table 3.5 lists some static analysis tool. This list is compiled from the SAMATE (Software Assurance Metrics and Tool Evaluation) static analysis tool list [80]. For each tool, we give the input language it accepts, the used technique and a comment on what it searches for.

Industrial grade static analyzers such as (Fortify SCA [13] or Coverity [14]) use a mix of the cited techniques and rely mainly on a knowledge base of already seen security vulnerability patterns. This knowledge base is updated regularly to keep

Tool name	Input language	Technique	Comment
Astrée	C	Abstract interpretation	checks for undefined code constructs and run-time errors, e.g. out-of-bounds array indexing or arithmetic overflow.
Clang Static Analyzer	C, Objective-C	Data flow analysis	Reports dead stores, memory leaks, null pointer dereference, and more. Uses source annotations.
CodeSonar	C, C++	Multiple techniques	analyzes and validates the code or binary to identify serious vulnerabilities or bugs that cause system failures, poor reliability, system breaches, or unsafe conditions.
Csur	C	Abstract interpretation and other heuristics	cryptographic protocol-related vulnerabilities
UNO	C	Symbolic execution	checks for uninitialized variables, null-pointers, out-of-bounds array indexing and user-defined properties

TABLE 3.5: Static analysis tools list

up with newly discovered patterns. These tools are effective but needs a large development and maintenance efforts and can miss dangerous vulnerability not yet included in their bases.

Currently many new tools are being developed and this makes choosing the right tool a difficult task. National Security Agency's (NSA) and Center for Assured Software (CAS) developed and published Juliet [16], a test base specifically designed for assessing the capabilities of static analysis tools. This base contains examples covering more than 100 CWEs [21]. To help an application auditor to choose the right technique a good insight on existing static source analyzers is given in [81]. Other researchers are trying to simplify tool choice by a security oriented test base [82] or directly giving evaluations of some well known tools [83].

3.5 Major problems of static analysis

The domain of static analysis has many open problems. The major ones are:

- *False positive*:

  When a static analysis tool emits a *false positive* this means that it detected a vulnerability when in fact there is none. When the rate of false positive is high the developer stops paying attention to the tool's output even though it may contains real vulnerabilities. So this problem is one of the common

problems that occur when using static analysis tools. For example when using Polyspace [8] among the detected flaws we observe a class called *orange warnings*. The orange warnings are potential flaws and need to be checked manually to confirm their status. This type of warnings is caused by the approximations done at the computation stage.

- *False negative*:

  A tool searching for some class of vulnerabilities may miss a vulnerability it must find and does not report it. This can be caused by a tool limitation or a . This undetected true vulnerability is labeled as *False negative*.

- *Scalability*:

  The scalability problem is related to the fact that some tools have difficulties to run on large and/or complex software. This can be caused by the combinatorial explosion [84] of possible reachable states, resource exhaustion, an implementation bug or a tool limitation [85].

- *Input language*:

  The front-end parts of static analysis tool is sometimes the bottleneck of code analysis process. Front-end limitations can make the tool unable to analyze a given application code. This is the case when an application is using a feature of programming language not supported by the analyzer or that is using libraries not available for the analysis tool.

- *Result interpretation*:

  Tools have different result format with a varying file type and vocabulary. The report format influences the manner a tool user can use it and correctly interpret the reported vulnerabilities.

These problems will be discussed in the chapters 4, 5 and 6 where we present our contributions.

Summary

Static analysis is an interesting method that can be used to detect security vulnerabilities. Static analysis tools are composed of three components: front-end, middle-end and back-end. Developing static analysis tool can be done by re-using existing components after a variable amount of tuning and adaptation. In the next chapters we will propose novel static analysis methods and implement them using existing frameworks and tools.

Chapter 4

Security vulnerabilities in C language applications

Security of computer systems is important in the modern cyber space. Security of businesses, persons and even governments is facing a growing threat from a wide variety of attackers. Eliminating vulnerabilities from application's code is necessary to prevent attacks. The first step towards eliminating security vulnerabilities is their detection, which can be an arduous task in large size programs. Static analysis of the code helps to automate this process, by guiding the programmer towards the potential vulnerabilities before they are discovered by an attacker.

We investigate in this research vulnerabilities that arise in C code through the calling of standard library functions. We define criteria to detect dangerous use of these functions, and show that the evaluation of a static analyzer implementing the proposed detection model yields a low False Positive rate.

The work presented in this chapter is published in the *Security and Privacy Wiley Magazine* [86].

Introduction

Many efforts have been made in the past years to prevent security errors from appearing in the released source code. The choice of the implementation language has a great impact on the security of the developed application as shown in language security studies such as: JavaSec [87] for Java [88] and LaFoSec [89] for functional languages such as OCaml [90]. Low level flexible languages such as C language are still widely used even though the security defects inherent to this language are well known.

Vulnerabilities can be prevented during software implementation by avoiding dangerous language constructs or using only a secure subset of the programming language. Many coding rules such as CERT C coding rules [32] have been defined to guide the developer in avoiding security defects.

In this chapter, we present security vulnerability property checks that will be used to enhance a static analysis tool. The covering of security issues represents a key difference with existing static analysis tools such as (Polyspace [8, 9], Frama-C [10, 11] and Astrée [12]). These tools all rely on abstract interpretation [73] to check for safety issues[1]. The second novelty is that the proposed vulnerability checks are constructed by analyzing the language specification and libraries documentation. This makes our work different and complementary to tools such as Fortify [13] and Coverity [14] that are built based on developer mistakes or bad habits seen in real life code. Thus, we focus on a set of vulnerabilities derived from the usage of C language library functions. We define properties that can be checked to locate these vulnerabilities. For each defined property, we provide the related attack scenario to show its effect on security. We also provide details on fulfilled implementation and conducted experimental evaluation using the available Software Assurance Metrics and Tool Evaluation (SAMATE) test base Juliet [16]. The use of such public test base provides a reference benchmark for comparison of our tool with others. First the tool is evaluated based on the number of test

[1]Safety issues concerns availability and resilience, i.e. there is no safety issues when the program does not crash or misbehave of its own. The security issues concerns resistance to an attacker pushing a program to behave in a manner he wants.

cases where the analysis finishes without errors. After, we measure the number of true positives (vulnerabilities correctly located) and the number of false positives (non-vulnerable code detected as vulnerable). At the end we have a precise evaluation and comparison of the developed tool based on quantitative attributes and not only qualitative ones.

4.1 Static analysis for security

On the source code, static analysis can be and is being used [91] to detect vulnerabilities. Early static analyzers used simple syntactic rules to locate potentially vulnerable code patterns (e.g. FlawFinder [69], Linux utility *grep* with special regular expression). These simple tools have a high rate of false positive and can not detect complex vulnerability involving the semantics of the code. Other more advanced static analyzers use abstract interpretation [73] to prove the absence of runtime errors RTEs [74] (Polyspace [8, 9], Frama-C [10, 11], Astrée [12]). These tools are more fitted to detect safety issues and do not deal with security issues. Modern static analyzers such as (Fortify SCA [13], Coverity [14]) use a mix of different strategies that rely mainly on knowledge base of already seen security vulnerability patterns. This knowledge base is updated regularly to keep up with newly discovered patterns. These tools are effective but needs a large development and maintenance effort and can miss dangerous vulnerability not yet included in their bases. Currently many new tools are being developed and this makes choosing the right tool a difficult task. Authors in [81] gives a good insight on existing static source analyzers. Other researchers are trying to help developers to choose the right tool by giving them a security oriented test base [82] that can be used to qualify and test a tool. Other researchers are directly giving evaluations of some well known tools [83].

Based on related work listed above, a solution is proposed, implemented and tested to detect automatically security vulnerabilities in C applications using static analysis techniques. In the following sections, we define the scope i.e. which part of the C language is covered and what vulnerabilities we are tackling.

4.2 Covered C language parts

The C language is among the most widely used programming languages half century after its introduction [92]. This language is composed of a central part defined by the ANSI C standard [39]. This standard describes the syntax and semantics of all the standard constructs and gives a full description of the standard library (headers files, constant values, data types, function headers and function descriptions, etc.). Compilers supporting the C language (open source or proprietary) must implement the standard part but may also implement different extensions on both the language and the standard library parts. The C language to some extent is therefore the composition of the standard part and a large number of extensions specialized for different systems. This makes a complete and exhaustive study of all extensions a heavy and sometimes impossible task as in the case of discontinued commercial compilers [93]. In our research, we study a target language composed of ANSI C and two widely used extensions: POSIX C [41] and GNU extensions [42].

The ANSI C part has a medium size and can be studied in whole. On POSIX C [41] and GNU extensions [42] we studied a list of the widely used functions. This list is obtained by statistical means on existing software code. We have studied the usage of the library functions in 24 open source projects written in the C language depicted in table 4.1. In all of these projects, we have identified 48062 calls to the 4461 target library functions and made a statistical analysis.

The figure 4.1 is a graph showing in the x-axis the number of functions from the target libraries that are called and in the y-axis the percentage of calls to them and the cumulative percentage of calls to them. As a result, functions never called

Application	**Type**
bash-1.14.7	command line tool
bind-9.10.2-P3	DNS server
curl-7.40.0	web library
dovecot-1.2.0	mail server
emacs-24.5	text editor
exim-4.77	mail server
grep-2.21	word searching tool
grub-2.00	bootloader
gzip-1.2.4	compression library
httpd-2.4.12	web server
lighttpd1.x-lighttpd-1.4.x	web server
nginx-1.6.3	web server
openssh-7.1p1	ssh server and client
open-ssl-1.0.1f	crypto library
ossh-1.5.12	ssh server and client
postfix-2.7.16	mail server
sudo-1.8.7	Linux utility
thttpd-2.25b	web server
tnftp-20141031	tftp server
vsft-pd-2.3.5	secure ftp server
wget-1.16.3	web utlitiy
wu-ftpd-2.6.0	ftp server
yardradius1.0.21	Radius server
zlib-1.0.4	compression library

TABLE 4.1: Open source applications used to compute C library functions usage.

are omitted from the figure. As depicted by the figure 4.1, the usage of functions from the targeted library shows a long-tail pattern, meaning that a few functions gather most of the usage while many functions are hardly used. Among the 4461 functions from the target libraries, only 751 functions were called at least once and 3710 target functions were never called. The dashed curve shows that the 116 most called functions account for 90% of the calls, and the 447 most called account for 99% of the calls.

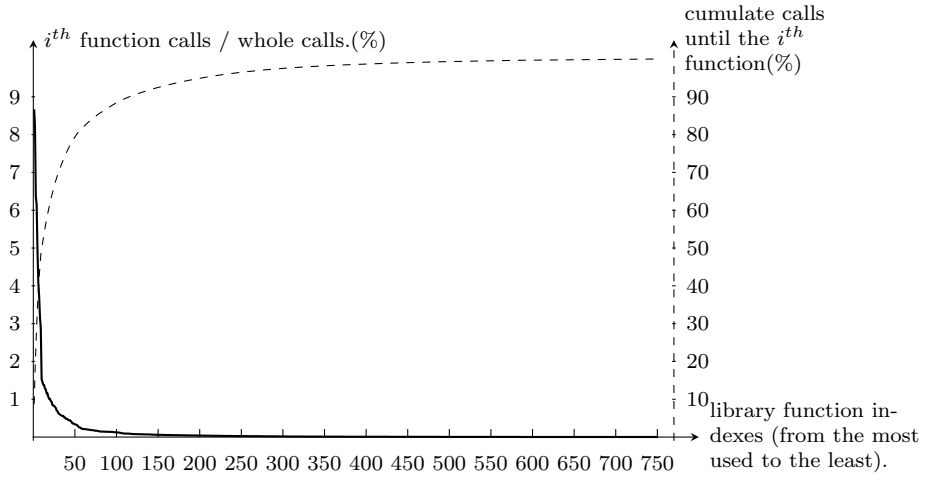

FIGURE 4.1: Usage rates of C language library functions in 24 open source projects.

The figure 4.2 depicts the top 30 most used functions. This figure shows that the nine first functions represent 50% of the calls.

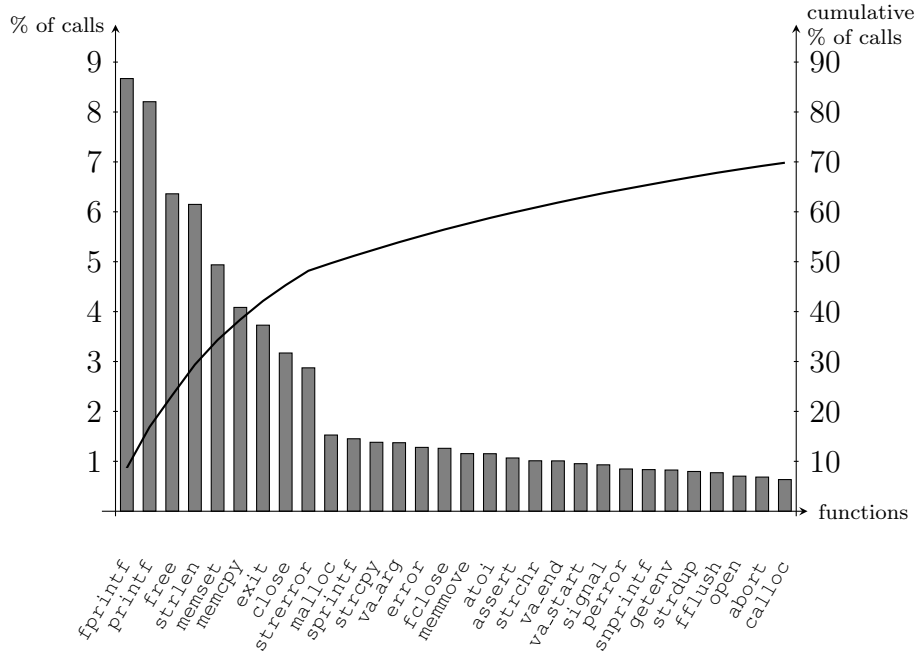

FIGURE 4.2: The top 30 most called C library functions in open source projects.

4.3 Covered security vulnerabilities

A security vulnerability can be defined in different ways. The IETF (Internet Engineering Task Force) RFC-2828 [94] defines it as: *"A flaw or weakness in*

a system's design, implementation, or operation and management that could be exploited to violate the system's security policy". The CVE Mitre Website [20] lists exploitable vulnerabilities found on real applications and the CWE Mitre Website [21] describes a large set of classes of vulnerabilities. Among these vulnerabilities, the CWE/SANS Top 25 ranks the 25 Most Dangerous Software Errors.

We focus on three target vulnerabilities:

- Format string vulnerabilities (CWE-134: Uncontrolled Format String, Top 25 rank 23);
- Command execution vulnerabilities (CWE-78: Improper Neutralization of Special Elements used in an OS Command ("OS Command Injection"), Top 25 rank 2);
- Buffer and memory vulnerabilities (CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'), Top 25 rank 3).

The "Format string vulnerabilities" and "Command execution vulnerabilities" can be directly associated to two CWEs, namely, CWE-134 and CWE-78 whereas "Buffer and memory vulnerabilities" can be associated with several CWEs amongst which CWE-120 is the most general. Note that in the Top 25 CWE-120 and CWE-78 are ranked 2 and 3, respectively, whereas CWE-134 is only ranked 23.

The choice of these target vulnerabilities is motivated, either by their presence in the Top 25 Most Dangerous Software Errors, which shows their extremely dangerous impact on the system security if they are exploited by an external attacker and the relative ease of their exploit, or by their wide usage in the studied open source projects of the target associated functions demonstrated below.

Table 4.2 lists the functions from the top 30 most used C library functions associated either with the three target vulnerabilities or none of them and gives the percentage of functions from the top 30 related to the vulnerabilities. This table shows that among the Top 30 functions used in open source projects, more than 46%, i.e. *14 functions* may lead to format string vulnerabilities or buffer and

memory vulnerabilities. This means that our targets are representative of actually used functions because they cover the first half of the functions used in open source software. The functions leading to "command execution vulnerabilities" are not represented in the Top30 but are ranked 208 in our statistical study of the most used functions in the Open Source Projects.

Vulnerability	**Related Top30 functions**	**Percentage**
Format string vulnerabilities	fprintf, printf, sprintf, snprintf.	13%
Command execution vulnerabilities	None	0%
Buffer and memory errors	strlen, memset, memcpy, malloc, strcpy, memmove, atoi, strdup, calloc, strchr	33%
Others	fclose, assert, va_end, va_start, signal, error, perror, getenv, fflush, open, abort, strerror, free, exit, close, va_arg	53%

TABLE 4.2: Mapping of the Top30 used functions to the target vulnerabilities.

As a conclusion, the chosen target vulnerabilities are representative of highly ranked vulnerabilities (rank 2 or 3) or vulnerabilities originating from the widely used functions. Target vulnerable functions are defined as the subset of the target functions that may lead to the three target vulnerabilities. All the target vulnerable functions belong to the Top 250 most used functions in open source projects.

For simplicity reasons, only a subset of the target vulnerable functions is considered (presented in table 4.3) to give demonstrative examples for the chosen property examples.

Exploiting a vulnerability usually requires interaction from the attacker via the application entry vectors. A data set is therefore *controllable by an external entity* if it is initialized via an application input vector. We call *tainted variables* all

Target vulnerability	**Target vulnerable functions**
Format string vulnerabilities	fprintf, fscanf, printf, scanf, sprintf, sscanf.
Command execution vulnerabilities	system, execl, execle, execlp, execv, execve, execvp, popen
Buffer and memory errors	calloc, malloc, realloc, fscanf,gets, scanf, sprintf, sscanf, strcat, strcpy, strncat, strncmp, strncpy, memchr, memcmp, memcpy, memmove, memset, scanf, gets, fwscan, sscanf.

TABLE 4.3: List of target vulnerable functions

variables present in the source code whose values are derived from *controllable data*.

4.4 Properties for security vulnerability detection and reporting

As stated in section 4.2, we focus on three categories of security vulnerabilities: format string, command execution, and buffer errors. For each vulnerability, we give an overview of the vulnerability, and present several properties that characterize ill-used of these functions leading to the vulnerability. If the property holds true, there is a vulnerability in the source code. For each property, we present a general description, an example of attack scenario applicable when this property holds, and a C code sample that shows one or more violations of the property.

4.4.1 Format string vulnerabilities

This vulnerability is caused by a misuse of a *formatted input/output function* described in chapter 2. Formatted functions take a variable number of arguments: a "format string" and values to be formatted. The "format string" is a mix of text that will be included as is in the formatted data and format specifiers that

start with the character '%'. Each specifier is replaced with the corresponding formated argument. For "output" functions, the format string is used to convert pure data in a human readable string representation that is written to the output stream (file, console, buffer). When used with input functions, the format guides the reading from the input stream to fill the given arguments. The formatting operation has a precise syntax and semantics defined in the language specification (or description). We present three properties that characterize ill-used format functions as follows:

FORMAT_WRONG_CALL:

- **Property description**: The format argument has an incorrect syntax, or the other arguments have not the right types and count according to the given format argument.
- **Attack scenario**: If the source code contains a wrong format call, the attacker has only to drive the software execution to execute the flawed call, to produce a Denial of Service (DoS [95]) due to an application crash.
- **C code sample**:

```
1 #include<stdio.h>
2  int main(){
3    int a;
4    char *s;
5    a = 0;
6    s = "hello"
7    printf("%b",a); // wrong format: unknown format specifier
       %b
8    printf("%d",s); // wrong format:  wrong argument
9                    // type 'char *' expected 'int'
10   printf("%d %d",a); // wrong format:  wrong argument count
11 }
```

In the piece of code above, the first call to *printf* causes a runtime error because the format specifier *%b* does not exist. The second call is wrong because the format specifier *%d* cannot be used with an argument of type *char **. For the third call the number of the format specifiers (2) is higher than the count of given arguments (1).

FORMAT_TAINTED_ARGUMENT:

- **Property description**: The format argument used to format input/output data is "controllable by an external entity".
- **Attack scenario**: There are numerous malicious formats the attacker can give to a vulnerable application. Depending on the way the wrong format is forged, the attacker can achieve different goals [96]. For example, if the attacker controls the format of an output function and concatenates `%s` to the actual format and whatever arguments passed, the attacker can leak arbitrary information. An arbitrary amount of stack data is printed on the output stream. If the attacker concatenates `%s%n` to an actual format, he can even achieve arbitrary code execution and completely control the application execution flow.
- **C code sample**:

```
1 #include<stdio.h>
2  int main(){
3    char str[60];
4    fgets(str,59,stdin); // input func: fgets, stream: stdin.
5    printf(str); // vuln : 'str' is controlled by
6                 // an external agent
7 }
```

In the piece of code above, the format argument *str* passed to the *printf* function call is read on the standard input via the *fgets* call. Therefore,

the property FORMAT_TAINTED_ARGUMENT holds on the *printf* call and a vulnerability must be reported.

FORMAT_BUFFER_OVERFLOW:

- **Property description**: A formatting function that writes data into or reads data from a buffer whose size of written (read) data is "controllable by an external entity" or the size of the formatted output may be bigger than the buffer size.
- **Attack scenario**:
  - If the format is not "controllable by an external entity" and the size of output is bigger than the buffer size, a buffer overflow arises leading to at least an application crash, and possibly an information leak or arbitrary code execution.
  - If the format is "controllable by an external entity" a buffer overflow will arise leading to an application crash then a DoS or an information leak or arbitrary code execution if other elements in the code can be controlled.
- **C code sample**:

```
1 #include<stdio.h>
2  int main(){
3    char str[10];
4    sprintf(str,"%s","AAAAAAAAAAAAA"); // buffer overflow !
5          // sizeof(str) < sizeof (output) : 10 < 14.
6 }
```

In the code above, the call to *sprintf* with the constant format *%s* causes a buffer overflow because the size of the output 14 is greater than the size of the receiving buffer *str* 10.

Property id	Related functions	CWE,CVE(s)
FORMAT _WRONG _CALL	fprintf, fscanf, printf, scanf, sprintf	CWE-628: Function Call with Incorrectly Specified Arguments
FORMAT _TAINTED _ARGUMENT	fprintf, fscanf, printf, scanf, sprintf	CWE-134: Use of Externally-Controlled Format String CVE-2015-8617 (PHP 7.x)
FORMAT _BUFFER _OVERFLOW	scanf, sprintf, sscanf, wscanf	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer CVE-2014-1545 (Mozilla Netscape)

TABLE 4.4: Format string properties.

The table 4.4 lists the target functions for each example of format property, and references CWE [21] or CVE [20] stemming from this property.

In this research, we choose to make the more precise CWE association but this is not always possible. Note that if the second property can be associated to a precise CWE-134, the two others can only be associated to CWE-628 that applies to any function call and CWE-119 that applies to a large number of buffer overflows.

Format string exploitation technical details

The generic class of a format string vulnerability is a "channeling problem". This type of vulnerability can appear if two different types of information channels are merged into one, and special escape characters or sequences are used to distinguish which channel is currently active. Most of the times one channel is a data channel, which is not parsed actively but just copied, while the other channel is a controlling channel [96].

This attack is publicly known since mid 2000, it was considered as a software bug. But in reality, its impact on security is worse than the well known buffer overflow issues.

A successfully exploited format string vulnerability could lead to the following problems (here the format string is entirely controlled by the attacker)

- **Crash of the program**: By utilizing format strings we can easily trigger some invalid pointer access by just supplying a format string like:

```
1 printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
```

Here for each %s a pointer is popped from the stack and dereferenced, this could potentially lead to an illegal memory access causing the program to abort.

- **Viewing the stack**: We can show some parts of the stack memory by using a format string like this:

```
1 printf ("%08x.%08x.%08x.%08x.%08x\n");
```

Here, each %08x conversion specification will retrieve a 4-bytes value from the stack and print it on 8 hexadecimal positions.

- **Viewing memory at any location**: the example:

```
1 printf ("\x10\x01\x48\x08_%08x.%08x.%08x.%08x.%08x|%s|");
```

Will dump memory from 0x08480110 until a NULL byte is reached. By increasing the memory address dynamically we can map out the entire process space. This because each %08x will consume a 4-byte value from the stack and the last %s will use the given address 0x08480110 as the start of string to print.

- **Overwriting of arbitrary memory**: is the most dangerous attack pattern, it uses basically the same principle of "viewing memory at any location" uses

the "%n" conversion specifier to write into memory an arbitrary value. This situation is described in the 'CWE-123: Write-what-where Condition', and can lead to an arbitrary code execution compromising the Confidentiality, Integrity, and Availability of the whole application or even the system.

- **Classic buffer overflow**: this special situation concerns the use of sprintf or even snprintf with wrongly computed size parameter, for example:

```
1    {
2    char buf[64];
3    sprintf (buf,'connected user:%s',username);
4   //username is an attacker controlled value.
5    }
```

4.4.2 Command execution vulnerabilities

This vulnerability may come from a misuse of a command or program execution function. An execution function takes as arguments a command and arguments if necessary and applies the command to all the arguments. Execution functions launch a command, given as a string argument, by using the system command interpreter. Although this feature is necessary to allow interactions of the program with the rest of the software environment, it can weaken the security of the application.

COMMAND_WRONG_CALL:

- **Property description**: The program and command execution function specification is not fulfilled or argument types and count are not correct.
- **Attack scenario**:

  The attacker must drive the execution to the flawed execution function call to achieve a DoS [95] due to an application crash.

- **C code sample**:

```
1 #include<unistd.h>
2  int main(){
3    execl("/bin/sh","/bin/sh","-c","ls -a",NULL);// correct
     call.
4    execl("/bin/sh","/bin/sh","-c","ls -a");     // wrong
     call.
5    execl("/bin/sh","-c","ls -a",NULL);          // wrong
     call.
6  }
```

In the piece of code above, the first *execl* call is correct because it fulfills the call conventions. The second call is wrong because there is no *NULL* argument at the end of the argument list. The third call is incorrect because the second function parameter is not set to the program name as stated in the function's specification.

COMMAND_TAINTED_ARGUMENT:

- **Property description**: The arguments of the execution function call (command string, program name or program arguments) are controlled by an external entity.
- **Attack scenario**: If an attacker controls one or more command execution function actual arguments, the attacker will be able to launch an arbitrary execution. This compromises not only the security of the application itself, but the entire system is at risk. For example the attacker can download and execute a malware on the target system via the command below:

```
1 wget http://malicious.server/malw.sh; \
2                         chmod +x malw.sh; ./malw.sh
```

- **C code sample**:

```
1 #include<stdio.h> // for fgets
2 #include<stdlib.h> // for system
3  int main(){
4    char str[60];
5    fgets(str, 59, stdin); // input: reading 'str' from stdin
     .
6    system(str); //  arbitrary command execution via 'str'.
7  }
```

In the example above, the variable *str* is initialized via the input function *fgets* passed as a command argument to the execution function *system*. The attacker can execute any command if he controls the standard input channel *stdin*.

COMMAND_TAINTED_ENV:

- **Property description**: Execution functions implicitly use environment variables for binding the command passed as argument and the executable file on the system. If the environment is *altered by the attacker*, the executed commands may be different from the expected ones.
- **Attack scenario**: If the attacker has access to the environment of the application, he will be able to launch arbitrary commands and programs and put at risk the entire system.
- **C code sample**:

```
1 #include<unistd.h> // for execlp
2  int main(){
3    execlp ("grep","grep","-h", NULL);
```

```
4       // "grep" is searched in "\$PATH", if "\$PATH" has
    been
5       // changed. The "grep" executed is not the real one.
6   }
```

In the example above, the *execlp* function searches for the program *"grep"* into the $PATH environment variable directories. If the attacker changes the value of the $PATH variable, he will be able to force the application to launch a program of his choice and so will be able to launch an arbitrary program.

Table 4.5 lists for each property, the target functions for which the property applies and some CWE and CVE references stemming from these properties.

Property id	Related functions	Related CWE, CVE
COMMAND_WRONG_CALL	system, execl, execle, execlp, execv, execve, execvp, popen	CWE-628: Function Call with Incorrectly Specified Arguments
COMMAND_TAINTED_ARGUMENT	system, execl, execle, execlp, execv, execve, execvp, popen	CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection'), CVE-2015-3306 (Proftpd)
COMMAND_TAINTED _ENV	system, execlp, execvp	CWE-114: Process Control, CVE-1999-0080 (wu-ftp)

TABLE 4.5: Command execution properties.

4.4.3 Buffer and memory vulnerabilities

This section focuses on buffer errors related to C library functions calls. For example the out of bound array accesses or pointer dereferencing errors are out of the scope of this section because they are not related to function calls. We consider the general *CWE-120: Buffer Copy without Checking Size of Input* as the envelop, but many CWE belong to this class: *CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer* or *CWE-121: Stack-based Buffer Overflow*.

A memory manipulation library function uses a set of unstructured data from a buffer as input or output argument. Functions taking as argument a pointer to an unstructured memory chunk (`char *`, `void *`, `int *`, `...`) are in the scope. For example, all of `malloc`, `calloc`, `memcpy`, `strcpy`, `memmove` are functions that manipulate the memory. We present properties that detect ill-use of memory manipulation functions.

MEMORY_TAINTED_SIZE_ALLOCATION:

- **Property description**: The size argument that contains the amount of allocated memory in memory allocation functions call is controlled by an external entity.
- **Attack scenario**: If the memory allocation argument is *controlled by an external entity*, an attacker can force the application to allocate huge chunks of memory causing a resource exhaustion leading to a DoS. The attacker can also give small values that will cause buffer overflows when the allocated buffers are read or written.
- **C code sample**:

```
1 #include<stdio.h>
2 #include<stdlib.h>
3  int main(){
4    unsigned size;
```

```
5    scanf("%d",&size); // input func : scanf, input stream:
     stdin.
6    void * buf = malloc(size);
7                        // vuln : allocation size in
     controlled
8                        //        by an external attacker.
9  }
```

The *malloc* function is called on the *size* argument that is initialized via an input operation *scanf*. This makes the amount of the allocated memory completely controllable by an external attacker.

MEMORY_TAINTED_NULL_TERMINATED_STRING:

- **Property description**: A string parameter that should be a null terminated is not or it is controlled by an external entity that can make it non null terminated.
- **Attack scenario**: If the attacker controls a string parameter in a string manipulation function, and drives the execution to this flawed call, he can achieve a DoS [95] due to an application crash.
- **C code sample**:

```
1 #include<string.h>
2  int main(){
3    char s[10];
4    strcpy(s,"AAAAAAAAAA"); //s string is not null terminated
      now.
5  }
```

The *strcpy* call copies the string *"AAAAAAAAAA"* into buffer *s*. But the size of the source string is set to 10 and no space is left to put the

terminating null character. The string *s* is now malformed and any further use can be a source of vulnerability.

MEMORY_TAINTED_SIZE_ARGUMENT:

- **Property description**: The size argument used in a memory manipulation function is controlled by an external entity.
- **Attack scenario**: When the attacker controls the size of the buffer, he can force a buffer overflow and achieve arbitrary code execution, information leak or application crash.
- **C code sample**:

```
1 #include<string.h>
2 #include<stdio.h>
3  int main(){
4    unsigned size;
5    char a[10];
6    char b[10];
7    scanf("%d",&size); // input func : scanf, input stream:
     stdin.
8    memcpy(a,b,size);  // TAINTED_SIZE_ARGUMENT = true.
9  }
```

The *size* argument of the *memcpy* call is read from input via *scanf*. So the copy operation is under the control of the attacker.

MEMORY_UNBOUNDED_INPUT:

- **Property description**: Data is read from an external source via an input function, and written to a fixed size buffer without a verification of the data size.
- **Attack scenario**: The attacker is able to trigger a buffer overflow by giving large input chunks to the application. Once a buffer overflow occurs he can exploit it to achieve arbitrary code execution, information leak or application crash.
- **C code sample**:

```
1 #include<stdio.h>
2  int main(){
3    char str[10];
4    gets(str); // unbounded input causing an overflow.
5  }
```

The *gets* call reads data on standard input (stdin) until a terminating newline is read. An attacker can cause an overflow by giving on the standard input channel a string bigger than the allocated buffer (10).

Table 4.6 lists for each property, the target functions and references CWE [21] and CVE [20] stemming from these properties.

4.5 Test and evaluation of Carto-C

An implementation of the properties qualifying security vulnerability was developed in the SafeRiver tool Carto-C [97] based on Frama-C [10]. The implementation consisted of adding the newly found properties to enhance the tool capabilities to cover more security vulnerabilities. The tool has been tested on the Juliet test base and also on real applications. More details on the implementation are given in the chapter 6, section 6.1.

Property id	Related functions	Related CWE, CVE(s)
MEMORY_ TAINTED_ SIZE_ ALLOCATION	calloc, malloc, realloc	CWE-789: Uncontrolled Memory Allocation
MEMORY_ TAINTED_ NULL_ TERMINATED_STRING	fscanf, gets, scanf, sprintf, sscanf, strcat, strcpy, strncat, strncmp, strncpy	CWE-170: Improper Null Termination CVE-2015-3200 (lighttpd).
MEMORY_ TAINTED_ SIZE_ ARGUMENT	memchr, memcmp, memcpy, memmove, memset, strncat	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer CVE-2014-0160, (heartbleed) OpenSSL.
MEMORY_ UNBOUNDED_ INPUT	scanf, gets, fscanf, fwscan, sscanf	CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer. CVE-2003-0595, WiTango Application Server

TABLE 4.6: Buffer and memory properties.

Testing efficiency and accuracy of a static analysis tool is a challenging task as stated by the SAMATE SATE IV report [98]. Our tool has been tested using synthetic test cases and using real application code.

4.5.1 Test with synthetic test cases

The tool has been tested using the synthetic code samples base Juliet from NIST [16] developed for the NSA CAS [99] and the NIST SAMATE SATE (Static Analysis Tool Exposition) [100] projects. This test suite contains more than 40000 code samples targeting more than 100 CWEs. Each Juliet sample contains a code snippet triggering the vulnerability called a *flaw* test case, and another code where the vulnerability has been patched called the *fix* test case. Each test case have

different data and control flow variations. The simplest case of control flow is called *a baseline test case*.

Based on the implemented properties we tested the tool on all test cases in the C Juliet test suite for the CWEs related to the target vulnerabilities. Table 4.7 gives for each property (column 1), the related CWE Id (column 2), states if the CWE is tested in the Juliet test suite (column 3) and if the property is tested by the CWE test cases.

Property	CWE	Present	Tested
FORMAT_ WRONG_ CALL	628	No	No
FORMAT_ TAINTED_ ARGUMENT	134	Yes	Yes
FORMAT_ BUFFER_ OVERFLOW	119	No	No
COMMAND_ WRONG_ CALL	628	No	No
COMMAND_ TAINTED_ ARGUMENT	78	Yes	Yes
COMMAND_ TAINTED_ ENV	114	Yes	No
MEMORY_ TAINTED_ SIZE_ ALLOCATION	789	Yes	Yes
MEMORY_ TAINTED_ NULL_ TERMINATED_STRING	170	No	No
MEMORY_ TAINTED_ SIZE_ ARGUMENT	119	No	No
MEMORY_ UNBOUNDED_ INPUT	119	No	No

TABLE 4.7: Mapping of vulnerability properties on Juliet test cases

Table 4.7 shows that four property related CWEs are present in the Juliet test suite (CWE-134, CWE-78, CWE-114 and CWE-789) and that only three of them truly test the target properties (CWE-134, CWE-78 and CWE-789). The CWE-114 can not be tested because it contains only Windows C code examples. As a result, the Juliet test suite allows for the verification of only three properties.

Table 4.8 gives for each property tested in the Juliet test suite and for each type of test case *flaw* and *fix*, the number of test cases (column #), the percentage of test cases truly analyzed (column %P) the percentage of *flaw* test cases for which the

tool detects the expected vulnerability (column %TP) and the percentage of *fix* test cases for which the tool does not detect the vulnerability as expected (column %TN). The false positive (fix case detected as vulnerable) ratio is not directly mentioned in the table and is equal to: $100 - \%TN$.

Tested property	CWE	Flaw			Fix		
		#	%P	%TP	#	%P	%TN
FORMAT_ TAINTED_ ARGUMENT	134	2000	58%	93%	5460	59%	100%
COMMAND_ TAINTED_ ARGUMENT	78	1560	95%	90%	2160	100%	100%
MEMORY_ TAINTED_ SIZE_ ALLOCATION	789	380	95%	100%	1100	90%	49%

#: number of analyzed test cases; %P: ratio of test cases passing compilation step; %TP: ratio of flaws that passes compilation and identified as vulnerable among all flawed test cases; %TN: ratio of fixes that passes compilation and not identified as vulnerable among all fixed test cases.

TABLE 4.8: Result on part of Juliet using Carto-C.

Table 4.8 shows that the subset of the Juliet suite that tests the target properties contains 12660 test cases. The percentage of test cases for which the tool truly performs the analysis is between 90% and 100% for CWE-78 and CWE-789 and 58% for CWE-134. This difference comes from the fact that the CWE-134 makes use of wide character functions not yet supported by Carto-C.

The percentage of test cases for which the tool gives the correct answer is between 90% and 100% except for *Fix* test cases of CWE-789 for which the tool produces a vulnerability while it should not (False Positive). We have analyzed the corresponding test cases and it appears that the fix in these test cases is a tainted value sanitization added in the code. This sanitization transforms the tainted data into non-tainted, but the current version of the tool is unable to recognize this behavior.

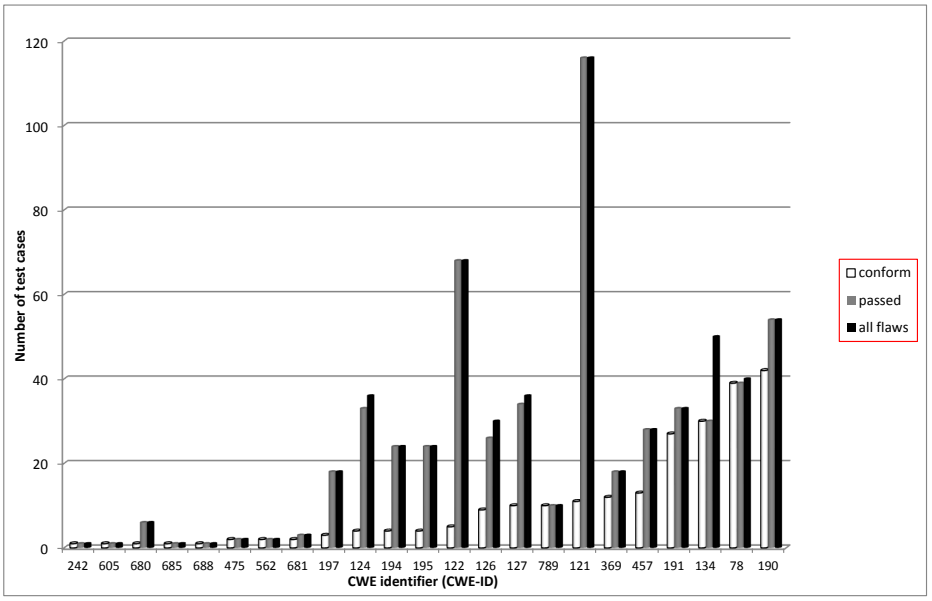

FIGURE 4.3: Result of Carto-C on Juliet baseline flaw test cases (After new properties added)

In the figure 4.3 it is depicted the result of running Carto-C on the entire Juliet baseline test cases. The term *conform* in this figure means that the vulnerability present in the flaw test case was correctly identified and reported with the right corresponding CWE identifier. We see a good coverage of safety related vulnerabilities (computed by Frama-C [10], collected and reported by Carto-C) such as: 190, 191, 457, 369, 121, 126, 127, 122, 195 and 194. To be able to measure the added value of this contribution, in the figure 4.4 it is presented the result of running ancient version of Carto-C on the same subset of Juliet base before any development. The major improvements are:

- The front-end is better and it is able to parse more test cases.
- The developed security property checks allows to correctly detect more security related vulnerabilities such as the CWE: 78, 134, 789.

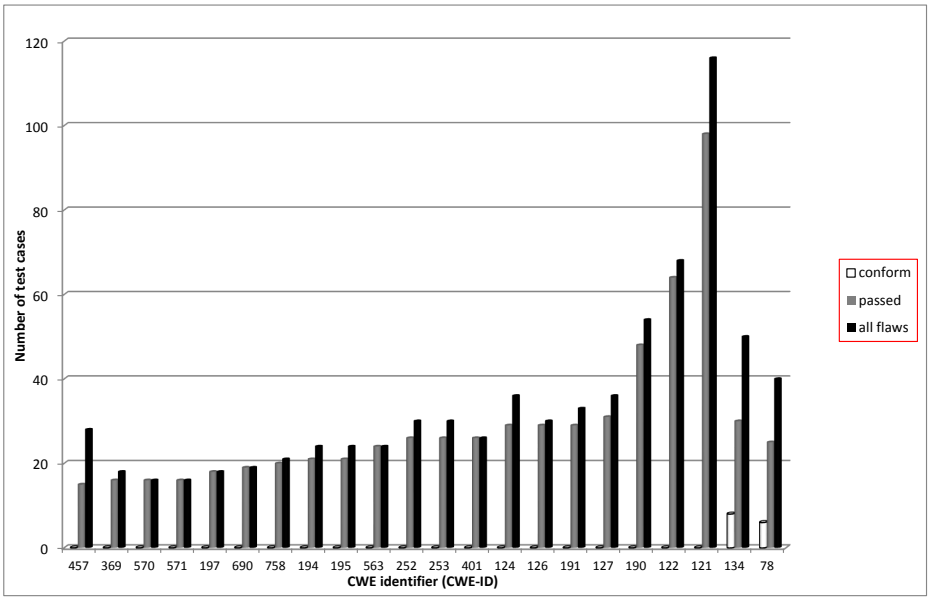

FIGURE 4.4: Result of old Carto-C version on Juliet baseline flaw test cases (Before adding new properties)

- Carto-C is able to report more safety vulnerability thanks to the development of functions to collect and return the alerts emitted by the Frama-C kernel.

The figure 4.5 depicts the result of running Carto-C on the entire Juliet test base. We see that in the two cases (running on baseline or on all) the results are practically the same. This shows the ability of Carto-C to detect vulnerabilities on complex programs with the same accuracy as it does for simple ones.

4.5.2 Test on real applications

We have tested Carto-C on real open source applications such as:

- *tar 1.13.19*: a widely used GNU/Linux utility, with a code size of 56 kloc (kilo lines of code). The analysis passed the compilation step after adding different header files missing in the C library implementation. The analysis

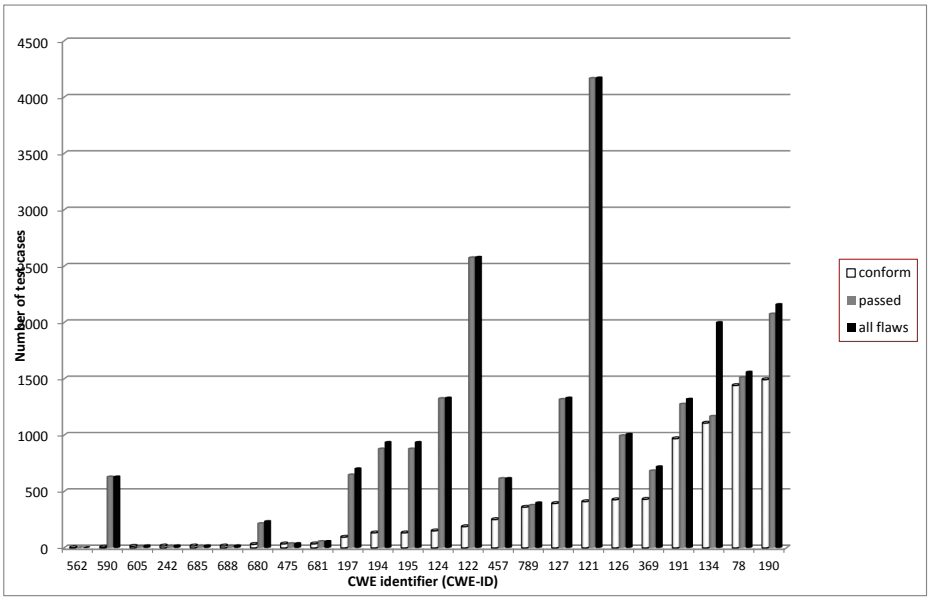

FIGURE 4.5: Result of Carto-C on all Juliet flaw test cases

does not finish because of the complexity of the application and the presence of unsupported constructs (recursive functions).

- *libpng 1.2.40*: a widely used multi-media library of a medium size (28 kloc). The analysis of all of the entry points of this library reported different errors that must be investigated manually to check if they are a true or false positive.

- *drivers/net/wireless/broadcom/brcm80211*: a Broadcom wireless driver of Linux kernel 4.x This type of application makes use of the Linux kernel driver libraries. The compilation step failed because, for instance, the tool does not support Linux kernel libraries.

On the open source applications, two classes of difficulties where encountered:

- Real source code makes extensive use of GNU C and POSIX C libraries not yet supported by Carto-C.

- The *Frama-C* framework does not scale very well on real life programs and does not support yet constructs often used in real applications such as recursive calls or while loops with complex stop conditions.

An ongoing developing work consists of extending the Carto-C supported libraries and analyzing the scaling problems.

4.6 Discussion and conclusions

In this work, we showed through experiments that many of dangerous security vulnerabilities are caused by the misuse of functions from the C libraries. This misuse can correspond to calling a library function in a wrong way, or exposing the call to library functions to data controlled from external sources. We have shown that detecting such errors with an automated tool is possible. On synthetic test cases we had a limited false negative rate (7%) and a low rate of false positive (between 10% and 51% in the worst case). Finally we have made an inventory of library functions and header files to implement to be able to analyze more Juliet tests [16] not yet supported and also real applications. These results were promising and resulted in the development of new functionalities in the Carto-C vulnerability knowledge base and also new test cases in the Carto-C test base.

Future work includes elaborating properties for new vulnerabilities and more complex properties to cover larger sets of vulnerabilities. We will also work on giving the tool user a better understanding of the reason of the property violations through execution traces. Another research axis, is to tackle the security issues stemming from sequences of function calls such as the famous malloc/free problems causing a use-after-free or double-free vulnerabilities.

Chapter 5

Vulnerability detection by behavioral pattern recognition

In this chapter we present our method for exploitable vulnerabilities detection in binary code with almost no false positives. It is based on the concolic (a mix of concrete and symbolic) execution of software binary code and the annotation of sensitive memory zones of the program traces. Three major families of vulnerabilities are considered (taint related, stack overflow and heap overflow). Based on the *angr* framework as a supporting software the tool VYPER was developed to demonstrate the viability of the method. Several test cases using custom code, Juliet test base [16] and widely used software libraries were performed showing a high detection potential for exploitable vulnerabilities with a very low rate of false positives.

Introduction

Building robust and secure software free from vulnerabilities is becoming a major concern of the IT community. Programming errors committed at the development stage are the main source of vulnerabilities and security holes. Different strategies are used to build more secure software such as the use of strict coding rules like

the CERT-C [32] or the choice of a less insecure programming languages. The automatic testing or *fuzzing* [101] can also be used to uncover security vulnerabilities. For the software industry, static analysis tools such as Coverity [14], Fortify [13] or Polyspace [8] are being adopted by developers to detect these programming errors and prevent them from happening on deployed applications. The source code static analysis has many drawbacks. We cite the problem of *false positives* where the analysis tool detects vulnerabilities, which in fact are not existing or are existing but not exploitable. Software security analysts will have a harsh task to sort true and false positives especially for large and complex software, and they may miss severe true exploitable vulnerabilities. To improve the security of an application using a static analysis tool all reported vulnerabilities must be dealt with using clear priority criteria. The exploitability can be a good criteria for establishing a priority to deal with reported vulnerabilities.

To resolve this problematic situation we propose a method that describes how to detect exploitable vulnerabilities with almost no false positive. The given solution helps software analyst to easily confirm a reported vulnerability by providing him an input sample that can trigger it. The proposed solution can also be used to automatically sort true and false positive vulnerabilities obtained from other software analysis tools.

The proposed method is a 3-stages process that first, computes program traces by a concolic [15] (a mix of concrete and symbolic) execution of the software binary code. Next, for each state of the computed trace(s) an annotation of sensitive memory zones is computed. Finally, for each state, a predicate based on this annotation and the result of the symbolic execution allows to verify if a vulnerability can be triggered in the corresponding state (giving the necessary input as well). The use of binary code instead of source code is motivated by the fact that it contains all the necessary details [78] to accurately find exploitable vulnerabilities. The other benefits of analyzing binary code is to be able to analyze the real code that will be executed. The real code is sometimes different than the source code because of compiler added optimizations or details related to the hardware such as process memory alignment and padding. Such approach best

responds to the objective of generating no false positive. We concentrated on three classes of vulnerabilities: taint related, stack overflow and heap overflow and we gave corresponding descriptions of the annotation and detection functions.

5.1 Concolic execution and binary code analysis

Concolic execution is a software analysis technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution (testing on particular inputs). It was introduced by Godefroid et al. [15] where it was used to assist random testing to cover a maximum numbers of execution paths. This method is also used by KLEE [76] for their unassisted high coverage testing. In their tool AEG (Automatic Exploit Generation) Avgerinos et al. [77] were interested in automatically generating an exploit by combining source analysis to find the exploitable vulnerability and binary analysis to produce the exploit. Mayhem [102] is also a tool that automatically find a vulnerability and generate an exploit based only on binary analysis. In our work, we focus more on modeling and detecting different classes of exploitable vulnerabilities, and less on generating a completely working exploit. This marks the difference that exits between our approach and the other cited above. Different tools and methods were developed since the introduction of concolic execution. Shoshitaishvili et al. [78] describe almost all major state of art techniques used in binary analysis based on concolic execution and other techniques that they implemented in the open source *angr* framework [68].

To demonstrate the viability of our approach we developed a real testable tool Vyper (VulnerabilitY detection based on dynamic behavioral PattErn Recognition) that uses *angr* framework. The tool can be used in *search mode* to search for vulnerabilities. In what we call *refine mode* the tool is used to check if a given vulnerability is exploitable or not and to produce the input that allows to confirm it. We tested Vyper on several custom test cases, Juliet test suites [16] and

some widely used open source libraries (openssl, libpng and libtiff). The results are very promising: in the first two cases most exploitable vulnerabilities were detected with almost no false positive (2% at most), while in the last case several previously unknown vulnerabilities were detected. We recall that the problem of analyzing program behavior is undecidable such as the case with the *halting problem*. Although, we can develop methods and tools that do interesting computations by taking trade-offs. The proposed method aims to minimize the false positive rate at the price of not being able to detect some vulnerabilities at all (false negatives) and a longer analysis time.

We define what is meant by *exploitable vulnerabilities* in section 5.2. We give the formalization of the proposed solution in section 5.3. Where in section 5.4 we show how this formal model can be applied on examples of exploitable vulnerabilities. The section 5.5 details the implementation and evaluation of Vyper. We conclude and discuss the future work in section 5.6.

5.2 Exploitable vulnerabilities

We consider exploitable vulnerabilities, *i.e.* vulnerabilities that allow an attacker to execute an arbitrary code on the target. Executing an arbitrary code means that an external application user is able only via input vectors (command line arguments, environment variables, file system and network sockets) to hijack the application control flow and execute code fed as input to the application. Arbitrary code execution is still actively exploited in the wild because of efficient application protection mechanisms such as the ASLR (Address Space Layout Randomization) [103], DEP (Data Execution Protection) [104] or stack canaries can be bypassed. Generally these protections are bypassed by exploiting an information leak vulnerability along with the one leading to the code execution exploit. Other vulnerabilities such as application crashing or interference with application logic will not be covered. This kind of vulnerability compromises the three security aspects

(confidentiality, integrity, availability) of a successfully attacked system. For this reason it is generally attributed the higher CVSS [79] scores especially when it can be exploited remotely by non-authenticated users. We focused our study on the most commonly exploited and documented vulnerabilities within the white and gray hat hacker communities such as Metasploit [105], Exploit-DB[106], or the pen-testing Linux distribution Kali [107]. We considered the following classes of vulnerabilities:

1. *Taint related vulnerabilities*:

   Format string, or command injection vulnerabilities, that are caused by calling some dangerous functions (printf, syslog, system, execlp) with a tainted attacker supplied argument. These vulnerabilities are exploitable if the tainted argument has not been sanitized at all or has been incorrectly sanitized. Exploiting a command injection vulnerability is trivial when an attacker controls all or parts of the executed command. Format string vulnerabilities are bit more complex to exploit. Worse, the format string exploiting techniques where unknown until 2000s. In the section 4.4.1 it is given how a vulnerable call to a format string function can be exploited. To detect these vulnerabilities one can monitor all calls to these dangerous functions and report a vulnerability when the given arguments are not sanitized (can contains prohibited characters) and are derived from the application input.

2. *Stack overflow*:

   A stack overflow occurs when data is written or read at a location that is beyond the stack buffer maximum size. A stack overflow will be exploitable if the return value stored on the stack is erased with an attacker supplied value allowing him to hijack the control flow into a desired location. Note that not all stack overflows are exploitable, for example if we erase only few stack memory locations and we do not reach the stored return address [108], the caused stack overflow is not exploitable. To detect this vulnerability we monitor all stored return values and report a vulnerability if one of these addresses is erased with an attacker supplied value derived from application

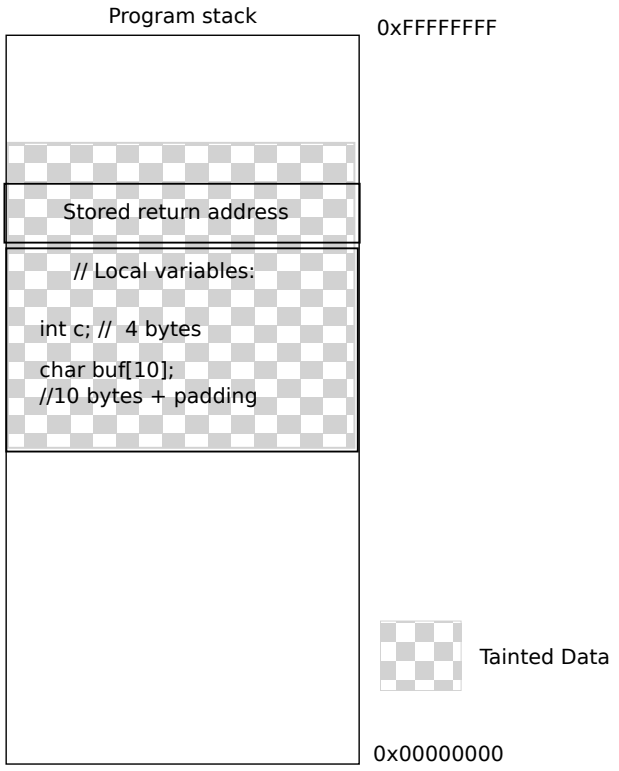

FIGURE 5.1: Exploitable stack overflow
28 bytes of tainted data written to the buffer erasing the return address.

input. The Figure 5.1 depicts a case of an exploitable stack overflow on an x86_64 machine running on Unix based OS.

3. *Heap overflow*:

   When a heap buffer is allocated, the dynamic memory allocator adds different control data before and after the buffer. Exploitable heap buffer overflow occurs when a heap is written beyond its size with a user supplied value (value derived form the input). Because the erased control data will confuse the memory allocator, this can potentially allow the attacker to execute arbitrary code. This vulnerability can be detected by keeping track of all sensitive heap memory addresses (allocated buffer start, allocated buffer end, freed buffer start) and a vulnerability is signaled when these memory places are accessed beyond the allocated size. Exploiting a heap buffer overflow is less easy than stack overflow. It needs a deep knowledge of heap meta-data and heap allocator algorithms. In practice, heap overflow exploits are probabilistic and use some techniques such as heap spraying [109] and heap layout

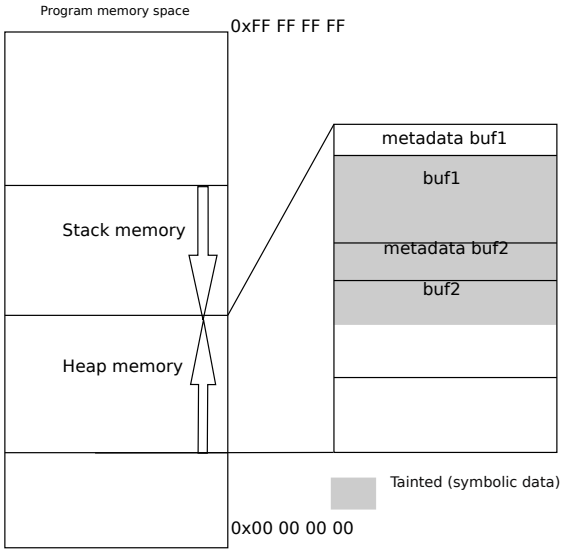

FIGURE 5.2: Exploitable heap overflow
Overflowing buf1 will erase "meta data buf2" and allow an arbitrary code execution

information leak to make the exploits more reliable. The Figure 5.2 depicts a case of an exploitable heap overflow where tainted data is written to *buf1* and the write operation overflows and erases the meta-data of the following buffer *buf2* leading to an exploitable heap overflow situation.

4. *Use-after-free*:

   A *use-after-free vulnerability* occurs when a freed heap buffer is accessed and can be exploited in the same manner as a heap overflow can be.

5. *Double free*:

   Double freeing a heap buffer can also allow an attacker to obtain a control hijack and execute an arbitrary code by confusing the memory allocator routines.

From the examples above we observe a general pattern shown in the Figure 5.3: a vulnerability is exploitable into a control hijack, if some *sensitive* memory zones (function argument, stored return address, heap buffer start and end) are accessed (read, write) in some special situations (with data derived from input or with data not correctly sanitized). By searching for this pattern in analyzed applications we

FIGURE 5.3: Exploitable vulnerability pattern.
Sensitive memory erased with tainted data.

will be able to locate *exploitable* vulnerabilities. The detection of these exploitable vulnerabilities is done in 3 steps:

- *Program traces construction:* computed using concolic execution, that guarantees the propagation of initial symbolic inputs along the generated traces with path formulas expressed within these inputs. We have constructed all traces of a maximum fixed length to guarantee the termination of this computation phase.

- *Sensitive memory annotation:* a sensitive memory zones set is constructed in this step. This set will be used by the next phase.

- *Vulnerability detection:* in this step we will check if the operation we are executing and the reached program state causes a sensitive memory zone to be written with data coming from input vectors (symbolic data) and report the corresponding vulnerability information. When possible the input data that will trigger the vulnerability is also reported.

5.3 Formalization of exploitable vulnerability detection

Below, we present a formalization of the detection and reporting of exploitable vulnerabilities based on their general behavioral pattern.

To detect an exploitable vulnerability in a given program, a method based on the annotation of program traces states' space is proposed. As in model checking method, reachable program states are constructed and the execution traces are kept. After that, for each trace, each state is annotated by a list of sensitive memory zones, based on the executed instructions. Finally, for each state, vulnerabilities are reported based on checks and constraints resolution done on the program states, execution traces and the constructed annotation containing sensitive memory zones.

We notice that for our goal we do not need to distinguish between memory locations and registers, so we can assume that the starting addresses of the memory correspond to the registers.

We will abstract the real computer program and the concrete machine model by random-access machine (RAM) model [110] enriched with additional instructions.

In order to be closer to the reality we consider two restrictions on the RAM model: first we assume that the number of registers (the memory) is finite and of size M; second we consider that each register (memory cell) can hold a bounded integer value – at most 2^n. Hence, the memory can be defined as a vector of size M: $Memory \in INT(n)^M$, where $INT(n) = \{0, \ldots, 2^n - 1\}$ be the set of numbers that can be represented with at most n bits in the binary notation.

Next, we augment the RAM model with additional instructions. An instruction is defined by the operation and the list of its arguments that are indexes of memory locations on which the operation is executed. We remark, that we consider instructions using constant values as new types of instructions. So, let OP be the set of all possible operation codes, then an instruction is a tuple $I = (op, a_1, \ldots, a_k)$,

where $op \in OP$ and k depends on op: $k = nb\_operands(op)$. The set of all instructions will be denoted as $INS \subseteq OP \times \mathbb{N}^K$, where K is the maximal number of arguments for any instruction.

A program $P = I_1, \ldots, I_n$ is an ordered list of instructions: $P \in INS^*$.

A state of the machine is given by the contents of the memory and by the current instruction index: $S_k = (i_k, M_k)$, $i_k \in \mathbb{N}$, $M_k \subseteq INT(n)^M$. An execution of an instruction I_{i_k} allows to pass from state S_k to S_{k+1}: $S_k \overset{I_{i_k}}{\Longrightarrow} S_{k+1}$.

A trace (an execution) of length n is the sequence

$$\pi : S_0 \overset{I_{i_0}}{\Longrightarrow} S_1 \overset{I_{i_1}}{\Longrightarrow} \ldots \overset{I_{i_{n-1}}}{\Longrightarrow} S_n.$$

We annotate each state S_i of trace π. We denote by A_i^π the corresponding annotation. An annotation is a list of triples $(memLocation, size, attribute)$, where $memLocation, size$ are respectively the address and the size of the annotated memory. The *attribute* value is used to keep the type of the corresponding memory zone ($RETURN\_ADDRESS, HEAP\_METADATA$, $CALL\_ARGUMENT, etc.$). Hence $A_i^\pi \in (\mathbb{N} \times \mathbb{N} \times attribute)^*$. We will omit the superscript π if π can be deduced from the context.

For each state S_i of π we consider the detected vulnerabilities information denoted V_i^π, which is a list of couples $(codeLocation, VULN\_INFO)$, where $VULN\_INFO$ is a structure that contains the name of the vulnerability, the corresponding CWE [21] identifier and the context (call stack, input values, etc). Hence, $V_i^\pi \in$ (N×VULN_INFO)*. As above, we will omit the superscript π if it can be deduced from the context. These definitions will be used in the following to show how the computation will be done.

We use a *concolic* execution of the program. One of the major advantages of such method is that it keeps track of the input data called symbolic data. Thus, it is possible to distinguish if a value of a memory location is computed using

external data. As we could see above, the input data plays a major role to make a vulnerability exploitable.

Another important point is that we limit the execution traces to a certain length (L). The idea behind this limitation is that we would like to detect vulnerabilities in a reasonable time, implying that the execution of the program will be stopped after some number of steps. The drawback of this approach is that the vulnerabilities requiring a higher number of steps for the detection will not be found. From the other point of view, the tool we use for the concolic execution already has similar limitations.

Hence, we will consider execution traces of length at most L and we will denote the corresponding set by π_L. We would like to remark that below we will consider that π_L is already computed (by making all corresponding runs). However, in the implementation we have chosen another approach where at each step we keep track of all traces and we evolve them in parallel. While this leads to the same final result, conceptually, it is easier to suppose that the set of traces is already computed.

For every program trace $\pi : S_0 \Longrightarrow \dots S_n$ of length n we will populate the list A_i, $0 \leq i \leq n$ as follows:

$$A_0 = \emptyset$$
$$A_{k+1} = Annotate(S_k, A_k, I_{i_k}), \quad 1 \leq k \leq n.$$

The function *Annotate* will allow to store the access to sensitive memory zones and this information will be further used for the detection of different kind of vulnerabilities. Based on the general behavioral pattern of a vulnerability some special memory locations have an important role, for example the return addresses stored in the stack frame or the buffer meta-data stored on the heap. When these memory locations are written, corresponding states need to be annotated for a future search for a vulnerability.

Next, we compute all V_k, $0 \leq k \leq n$ as follows:

$$V_0 = \emptyset$$
$$V_{k+1} = Detect(S_k, A_k, I_{i_k}, V_k), \quad 1 \leq k \leq n.$$

The function *Detect* checks for different vulnerabilities. The check is based on the annotated list for each state. Given the behavioral pattern of a vulnerability and the previous annotations on the execution path the *Detect* function will check if the conditions to have an exploitable vulnerability are met. In the positive case, it stores the information about the found vulnerability in V_{k+1}. For example if a memory location is annotated as being a stored return address in a function stack frame and it is written with tainted (symbolic) data, then we signal this as exploitable stack overflow.

The introduced model allows to describe a general detection framework based on the annotation of used memory locations. In the next section we describe how this framework can be applied for the classes of vulnerabilities we are interested by.

5.4 Formal model application on exploitable vulnerabilities

For practical reasons we will group program instructions into functional groups (i.e. memory access, subroutine call), as for many instructions the *Annotate* and *Detect* functions are almost identical. These groups are closely related to the functioning of *angr* framework [68] and especially to event based breakpoints that can be fired, e.g. on a memory/register access or a function call. The used part of [68] will be detailed in section 6.2.3. However, it should be clear that it makes no particular difficulty to unroll the corresponding definitions and give them for concrete instructions.

So, below we present the functions *Detect* and *Annotate* used for the detection of the five types of vulnerabilities.

- Taint vulnerabilities (printf, system):
  - Function *Annotate*:
    It is the identity function so the annotation list A_i will be always empty. We have no need to annotate any memory locations for this type of vulnerabilities. The sensitive memory to check is the given argument when calling these dangerous functions.
  - Function *Detect*:
    If the current operation is a call to taint related function, and the format argument is pointing a symbolic area then report a vulnerability.
- Stack overflow:
  - Function *Annotate*:
    If the current operation is a function call, then add the triplet ($stack\_pointer$, $reg\_size, STACK$) to the list A_i. If it is a `ret` operation subtract the corresponding stack pointer element from the set A_i.
  - Function *Detect*:
    If the current operation is a memory write, and the destination argument is within A_i and marked as $STACK$ and the source is symbolic then report an exploitable stack overflow vulnerability.
- Heap overflow:
  - Function *Annotate*:
    If the current operation is a call to a memory allocation function then add the triplet ($malloc\_ret\_value, mallo\_arg, HEAP$) to A_i
  - Function *Detect*:
    If the current operation is a memory write, and the data is written beyond the size of a buffer annotated as a $HEAP$ element and the source

buffer is symbolic then report an exploitable heap overflow vulnerability.

- Double free:
  - Function *Annotate*:
    When *free* is called, the passed argument is annotated as being already freed buffer.
  - Function *Detect*:
    When *free* is called, we check if the passed argument is in our sensitive memory zones and marked as freed. In the positive case we report an exploitable vulnerability.
- Use after free:
  - Function *Annotate*:
    The same as for detecting double free.
  - Function *Detect*:
    When a heap buffer is accessed we check if it is in the set of sensitive memory zones marked as already freed. If so, we report an exploitable vulnerability.

These are only examples of the application of the proposed vulnerability detection method. Other vulnerabilities (Information leak, DoS, Authorization bypass, etc) can be covered by figuring out what *Annotate*, *Detect* functions are needed.

5.5 Test and evaluation of Vyper

The detection of exploitable vulnerabilities presented above is implemented and evaluated in the Vyper tool. Vyper is a tool that allows to analyze a binary file and report any exploitable vulnerability it finds. It is implemented using the *angr* [68] framework. The tool has been tested on the Juliet [16] test base and gave satisfying results. It has also been tested on some real applications with good

results. The whole details of Vyper implementation are given in the chapter 6, section 6.2.

To evaluate our implementation of the proposed method we used three types of test cases:

- **Testing on custom test cases**: During the development, we built a small test base that was used to test the detection of some simple cases of vulnerable code. This base was also used for debugging and non-regression purposes.
- **Testing on synthetic test cases**: This testing was done using publicly available Juliet test base [16] that contains thousands of test cases especially written to test static analysis software.
- **Testing on real applications**: We used some well known applications and libraries that are available in open-source to test our tool on real life code.

5.5.1 Testing on custom test cases

The table 5.1 below summarizes the different test objectives. All tests were run in detection mode and all of the introduced vulnerabilities were detected and reported correctly. For example, one of these custom tests makes several calls to *printf* function. All calls except one use valid constant format specifiers. The remaining call uses a tainted format string and is only called if the input from *argv* has a special value. The vulnerable call was correctly detected and the needed input value was precisely reported. The source codes of these tests are given in the annex C.

5.5.2 Testing on Juliet test base

Juliet [16] is a collection of test cases written in C language. It contains examples for 118 different CWEs [21]. A test case contains at least a vulnerable code

TABLE 5.1: Results on custom tests using Vyper.

Vulnerability	CWE-ID	Comment
Tainted format	134	The code contains a vulnerable call to *printf* conditioned with input values.
Tainted format with concrete value	134	The code contains a vulnerable call to printf conditioned with input value. When the input value is constrained with a concrete value, no vulnerability is detected.
Stack overflow	121	The code contains 2 stack overflows: one with a loop and an index the other with a call to strcpy.
Double free	415	The test calls 2 times "free" in 2 different functions on the same buffer pointer
Use-after-free	416	The test code tries to access a memory location of a freed buffer. The test is reported as a heap overflow.
Heap overflow	122	A program that allocates heap memory and causes a write overflow with tainted data.

(flaw) and the same code with the vulnerability fixed (fix). Flawed or fixed code can be activated using compiler macros. For each CWE, test cases are created using the simplest form of the flaw as well as other cases testing the same flaw with added control or data flow complexity. For example *CWE134_Uncontrolled_-Format_String__char_connect_socket_snprintf_01.c* test case will test the CWE-134 with tainted data source from "connect_socket" and a sink vulnerable function "snprintf". The suffix "01.c" means that this test is the simplest case of this flaw. So, the *CWE134_Uncontrolled_Format_String__char_connect_socket_snprintf_11.c* is a test case testing the same vulnerability but with more complex control or data flow, i.e. the use of intermediate variables or function calls that cross multiple files.

TABLE 5.2: Result of Vyper on parts of Juliet test suite.

Type	CWE	Time	#	%TP in Flaw	%TN in Fix
taint related	134	1h	100	100%	100%
taint related	78	36min	80	73%	100%
stack overflow	121	2h	232	0%	99%
heap overflow	122	1h	136	14%	98%
double free	415	4m	12	100%	100%

time: total analysis time; #: the number of analyzed test cases;
%TP in Flaw: the ratio of flaws correctly identified as vulnerable among all flawed test cases;
%TN in Fix: the ratio of fixes correctly not identified as vulnerable among all fixed test cases.

For our testing we have developed scripts that will compile test cases into binary code, launch VYPER with all checkers activated. These scripts also collect, normalize and synthesize the analysis results. The results are summarized in table 5.2. We can notice that we generally have a very satisfying rate of false positives ($FP = 100 - TN$): from 0 to 2%, which is in correspondence with our objective to build a method with a very low false positive rate. Also, we remark that for the CWE-121 (stack overflow) we detected 0% of exploitable vulnerabilities. This is due to the fact that the flawed code does not allow to completely erase the return address that is necessary for an arbitrary code execution. In other words the CWE 121 test cases present in the Juliet test base are not exploitable to gain arbitrary code execution. For the CWE 122, the low rate of detected vulnerability is caused by the fact that some heap overflows occur when reading heap buffers. The read access overflow is not considered exploitable in our definition because no heap meta-data can be erased with a read access. We equally tested several commercial and proprietary tools on the same test cases and we found that they have a relatively higher mean value of false positive (FP). Hence, our tool performs better on the corresponding vulnerability classes. One explanation of such performance is that tested commercial tools are not tuned for exploitable vulnerabilities and also they usually rely on techniques inherently generating a high rate of false positive alerts.

5.5.3 Testing on real applications

Testing our tool on real applications is an important step to show its effectiveness. Thousands of applications are now available in open-source. However, the intrinsic limitations of the *angr* framework allow us to perform the tests only on small or medium size applications. Another problem when dealing with open source code is how to state if the found vulnerability is correct or not automatically. We tested VYPER on a variety of open source software:

- *Udhcp server*: udhcp-0.9.8 is a program running on a variety of devices (routers, modems, set-top boxes, IP cameras, etc.). We inserted into the source code a vulnerable *printf* call (call to *printf* with a tainted format) at the code line: udhcp-0.9.8/dhcpd.c:102. VYPER was able to correctly detect this vulnerability in about 3 minutes. This result shows the effectiveness of the tool for vulnerability detection on this type of (embedded) programs.

- *Widely used libraries*: We tested VYPER on OpenSSL-1.1.0f (libssl.so), libpng-1.5.20 and tiff-3.8.1. To test these libraries we launched the tool directly on the ".so" file and changed the entry point to each of the exported functions. We fixed the timeout to 300 seconds to be able to analyze a maximum number of functions in a reasonable time. The results are shown in table 5.3. In this table we see that, by fixing the analysis time to only 5 minutes per function we were able to analyze the majority of these library functions (only 33 time-outed among 786 function for OpenSSL). The detected vulnerabilities are not confirmed to be true positives. Analyzing a library function by function may lead to false positive independently from the performance of the analysis method. This is because some functions are never called directly in a normal use case of the library. This experimentation allows at least to confirm that for the functions where the analysis terminated and no vulnerability were detected that they are free from exploitable vulnerabilities. For the cases where vulnerabilities were detected we know that if a real exploitable vulnerability exists it will be among the found ones specially if the analysis did not timeout.

TABLE 5.3: Detection of vulnerabilities in widely used libraries using Vyper.

Library name	Source size	Binary size	# of functions	Timeouted functions	Entry function	Detected CWE id
OpenSSL-1.1.0f	246 kloc	1.2 MB	786	33	SSL_add_dir_cert_subjects_to_stack	CWE-122
					SSL_check_private_key	CWE-121
					SSL_CTX_set_ct_validation_callback	CWE-121
					SSL_CTX_use_PrivateKey_ASN1	CWE-121
libpng-1.5.20	33 kloc	452 kB	235	50	png_destroy_struct	CWE-415
					png_do_unpack	CWE-122
					png_free_default	CWE-415
					png_info_init_3	CWE-415
					png_push_process_row	CWE-121
					png_safecat	CWE-121
tiff-3.8.1	44 kloc	1.1 MB	215	13	TIFFCreateDirectory	CWE-415
					TIFFCreateDirectory	CWE-122
					TIFFGetConfiguredCODECs	CWE-122
					TIFFInitCCITTFax3	CWE-122
					TIFFReadEXIFDirectory	CWE-415
					TIFFReadEXIFDirectory	CWE-121

5.5.4 Testing refinement mode

To demonstrate the usability of Vyper *refine mode*, we analyzed the example in figure 5.4 using Carto-C. Carto-C detected a stack overflow vulnerabilities on the lines 23 and 30. By analyzing the the compiled application binary in refine mode to search for exploitable vulnerabilities and by giving as input the already found vulnerabilities locations. Vyper reported that the first is not exploitable and the second is. This experimentation shows the effectiveness of the refine mode to sort vulnerabilities into exploitable and non-exploitable ones and helping an application auditor to tackle the found vulnerabilities in the right order in order to augment the analyzed application security.

5.5.5 Other Vyper use cases

The capacity of Vyper to analyze binary code gives it the ability to accomplish different tasks such as:

```
1  #include<string.h>
2  int i;
3
4
5  void function1(char *str) {
6
7     char buffer[16];
8
9
10   for( i = 0; i < 16; i++)
11     buffer[i] = str[i]; // Non-vunerable call
12
13 }
14
15
16
17 void function2(char *str) {
18
19    char buffer[16];
20
21
22   for( i = 0; i < 24; i++)
23     buffer[i] = str[i]; // Non exploitable stack overflow
       vulnerability detected by Carto-C, and marked as non exploitable
       by Vyper.
24
25 }
26 void function3(char *str) {
27
28    char buffer[16];
29
30    strcpy(buffer,str); // Exploitable stack overflow vulnerability
       detected by Carto-C, and confirmed as exploitable by Vyper.
31  }
32
33 int main(int argc,char** argv) {
34
35     function1(argv[1]);
36     function2(argv[1]);
37     function3(argv[1]);
38 }
```

FIGURE 5.4: Code used to test refinement mode

- Helps understanding and reverse engineering binary application such as device drivers and other important operating system components. For example we have done an experimentation on the USB driver for Windows XP SP3 and got a detailed control flow graph representation.

- Tracks newly introduced vulnerability in application from version to version

by run it in refine mode with the undesired vulnerabilities located in the newly added code lines. If any vulnerability is detected we know the newly added code is the responsible.

- Binary code complexity measuring by computing some metrics on the computed control flow graph.

5.6 Discussion and future work

We have shown the effectiveness of using state-of-art binary-code analysis frameworks to detect exploitable vulnerabilities. The method we propose tries to recognize the common patterns present in application behavior allowing successful exploits. These patterns are searched using concolic execution engine provided by the *angr* framework. The implemented tool VYPER performs well on synthetic test cases as well as on real life applications.

One of the major drawbacks of the developed method is its low speed with respect to other methods. Moreover, at the time of the writing it does not scale very well for large applications.

The refine mode of VYPER can be very useful to check newly introduced vulnerabilities when application code is modified by aiming the new added lines of code and specifying the vulnerability we do not want to have.

Hence, in the future we will try to enhance the refine mode and try to demonstrate its use to detect newly added vulnerabilities in large software. We will also try to make the tool faster and more scalable for large programs. Another direction that we plan to explore is the addition of more vulnerability patterns, in particular those related to race conditions and parallel executions.

Chapter 6

Tools implementation details

In this chapter it is presented all the developments done along this thesis. The first tool Carto-C [97] was extended with the new security vulnerability checks for C language applications presented in the chapter 4 and tested with the Juliet test base [16] and on real application. Vyper, the second tool we developed to detect exploitable vulnerabilities using the method described in chapter 5 is also presented.

6.1 Carto-C

6.1.1 Presentation of Carto-C

Carto-C is a tool for establishing the cartography of a C source code by computing the attack surface (input and output streams). The figure 6.1 shows a use case of the Carto-C tool. It is also able to find some safety and security vulnerabilities such as:

- Finding potential run-time errors with static analysis, thanks to its underlying Frama-C stem.

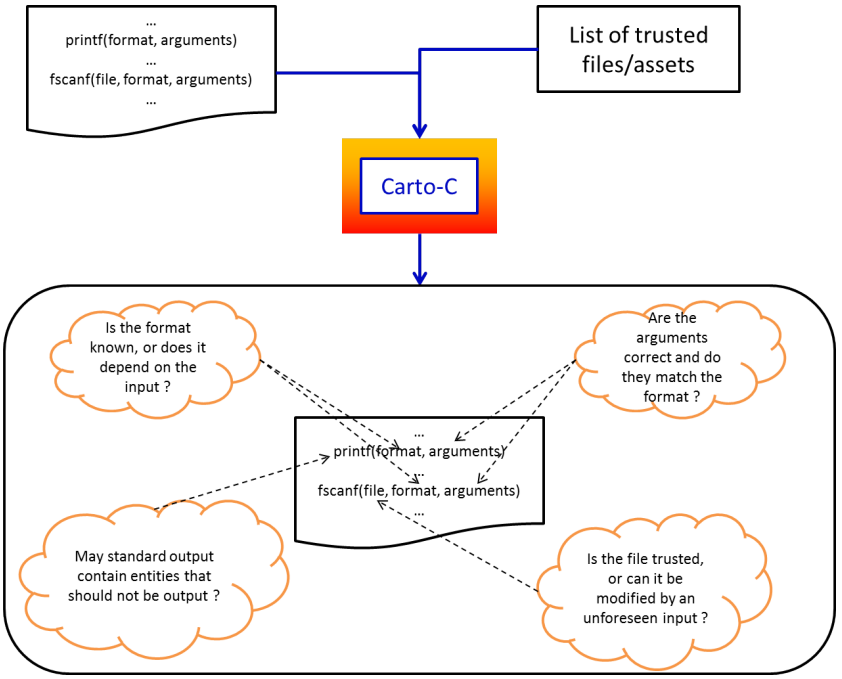

FIGURE 6.1: Carto-C use case

- Finding input and output points in the code, even those the designer/developer is not aware of. This includes files, standard input and output, environment variables, the network, the localization, the current time, etc.
- Finding critical points that depend on these input or that have an influence on the output. This allows e.g. finding whether a password can potentially be output on the standard error output.
- Finding vulnerabilities linked with formatting and execution functions. These vulnerabilities correspond to the common weaknesses enumeration items CWE 134 and CWE 78

6.1.2 Presentation of Frama-C

Frama-C stands for Framework for Modular Analysis of C programs. Frama-C is a collection of inter-operable static program analyzers for C programs. Frama-C has been developed by Commissariat à Énergie Atomique et aux Énergies Alternatives (CEA-List) and INRIA.

Frama-C relies on CIL (C Intermediate Language) [55] to generate an abstract syntax tree. Several plugins can manipulate this abstract syntax tree. The frequently used plugins are:

- *Value analysis*: computes a value or a set of possible values for each variable in a program. This plugin uses abstract interpretation [73] techniques and many other plugins make use of its results.
- *Jessie*: verifies properties in a deductive manner. Jessie relies on the Why or Why3 back-end to enable proof obligations to be sent to automatic theorem provers like Z3, Simplify, Alt-Ergo or interactive theorem provers like Coq or Why. Using Jessie, an implementation of bubble-sort or a toy e-voting system can be proved to satisfy their respective specifications. It uses a separation memory model inspired by separation logic.
- *WP (Weakest Precondition)*: similar to Jessie, verifies properties in a deductive manner. Unlike Jessie, it focuses on parameterization with regards to the memory model. WP is designed to cooperate with other Frama-C plugins such as the value analysis plug-in, unlike Jessie that compiles the C program directly into the Why language. WP can optionally use the Why3 platform to invoke many other automated and interactive provers.
- *Impact analysis*: highlights the impacts of a modification in the C source code.
- *Slicing*: enables slicing of a program. It enables generation of a smaller new C program that preserves some given properties.
- *Spare code*: removes useless code from a C program.

Carto-C makes use of the *value analysis plugin* to locate safety vulnerabilities in the analyzed program. The computation done by this plugin is also used to find security vulnerabilities by checking for dependence between data coming from the attack surface and dangerous sinks (dangerous function calls).

6.1.3 Implementation details

The development work related to this thesis was to extend Carto-C to cover vulnerabilities we described in chapter 4. Since Frama-C (internal engine of Carto-C) only supports a subset of the standard C library, the unsupported target library functions (GNU C, POSIX C...) have been added: the header files have been implemented or completed with missing functions and the ACSL (Annotated C Specification Language) specifications [111] of these functions have been developed. The properties checking security vulnerabilities were added to Carto-C knowledge base that was specially designed to be easily extended without heavy changes on core components. Hundreds of entries were added to this base for every library function that we identified in our properties. Small testing suites were developed and tested continuously to ensure that the added entries produced the desired result at least on basic test cases.

A *security property check* entry in Carto-C knowledge base is composed of three parts as shown in the example depicted in figure 6.2. The *property type* is related to vulnerability class and it acts as a switch that will choose what detection algorithm will be triggered. The *function name* is the concerned function and the *argument list* is list of argument indexes related to the security check. The Carto-C knowledge base contains also the list of input functions with information on the streams these functions read the data from and in what argument the input data is stored. Data initialized with such an input function will be considered as a *tainted external data*. This listing of input vectors allows Carto-C to compute the application attack surface and report it in a comprehensible manner.

In numbers, the developments added to Carto-C consisted of:

- 300 header files of ANSI C, POSIX or LINUX were added.
- 500 ACSL specifications of C library functions were added or modified.
- 200 lines Ocaml code to check for format string correctness.
- 300 input functions added in the Carto-C knowledge base.

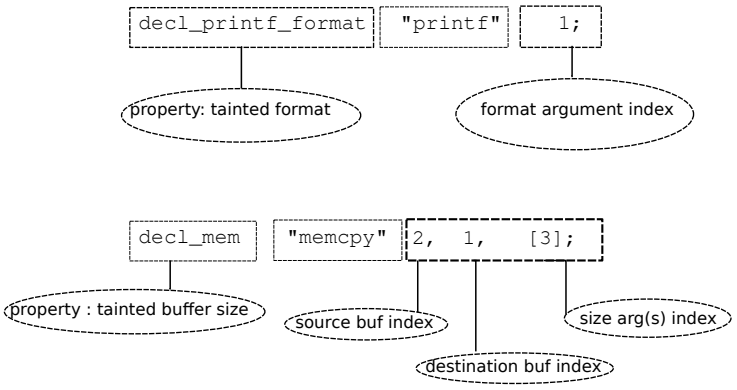

FIGURE 6.2: Carto-C security properties examples

- 200 output functions added in the Carto-C knowledge base.
- More than 120 security check properties concerning: format string, command execution and memory manipulation functions were added to Carto-C knowledge base.
- 40 custom test cases were written for the purpose of unit testing all the added developments.
- Practically all of the 700 most used functions obtained using statistical means on open-source projects were studied. Related security checks were added to Carto-C.

6.2 Vyper

As stated before we built a tool called VYPER that implements the analysis method described in chapter 5. Its implementation needs a good concolic execution engine. Implementing a concolic engine from scratch is hard task, which is not in our research scope. For this reason VYPER uses the *angr* framework [68] for program loading and states exploration. This allows to accelerate the development and helps us to focus more on vulnerability detection tasks. Another important point

is that *angr* is actively developed and continuously improved, so our tool will benefit from further improvements of the *angr* framework.

6.2.1 A brief description of *angr* framework

angr is an open-source framework available to the security community. It is a binary analysis framework that implements a number of analysis techniques that have been proposed in the past. This allows researchers to use them without wasting their effort reinventing the wheels. We cite some of the techniques implemented and documented in *angr* framework: binary loader for different OSes and architectures, control flow graph (CFG) computation, data flow graph (DFG) computation, value-set analysis (VSA), concolic execution using execution path explorers and code inspection. Code Inspection provides a powerful debugging interface, allowing breakpoints to be set on complex conditions, exact expression makeup, and symbolic conditions. This interface can also be used to change the behavior of the concolic execution engine by firing callback functions when a special breakpoint is hit.

6.2.2 Vyper specification

Vyper can be specified as follows:

- Input: the binary program, the entry point function, analysis mode (optional, default: detection), vulnerability class (optional, default: all), environment model file (optional).
- Output: Vulnerability report containing for each reported vulnerability: vulnerability CWE identifier, location information, input values, call stack.
- Requirements:
  - Load the binary file and initialize the initial state as specified with a special environment model file if any.

- Activate the requested checkers by setting the corresponding breakpoints that are used for the annotation and detection purposes.
- Launch and control the concolic execution.
- If an annotation breakpoint is hit, trigger callback function that will store the annotation information.
- If a detection breakpoint is hit, trigger callback function that will check if the executed code is vulnerable, and store vulnerability information if a vulnerability is present.
- If the symbolic execution is stopped (after a timeout) or finished, collect all the reported vulnerabilities and output it in the requested format.

6.2.3 Vyper implementation details

Vyper is an application developed in Python. The choice of this language is motivated by the fact that *angr* is shipped as a python module. So, it was easier to use the same language. Vyper is composed of different vulnerability checkers. Each vulnerability checker is implemented in separate functions and can be activated via command line arguments. This modularity allows Vyper to be easily extended to cover more vulnerabilities. Functions to pretty print the control graph flow of the analyzed application for debugging purposes are also implemented. Due to some missing C library functions stubs in the framework, about 15 functions were added to the existing *angr* implementation of C library functions. These functions were implemented or modified to make the results more precise. We note that the missing functions stubs were not a blocking issue such as with Carto-C or more generally with static source analysis tools. During the development of Vyper different bugs of *angr* were discovered. One discovered bug concerning a special type of breakpoint was reported to the developer community, and our proposed work around was accepted [112]. The figure 6.3 shows the general algorithm followed by Vyper. In the following paragraphs we will explain the internals of each box.

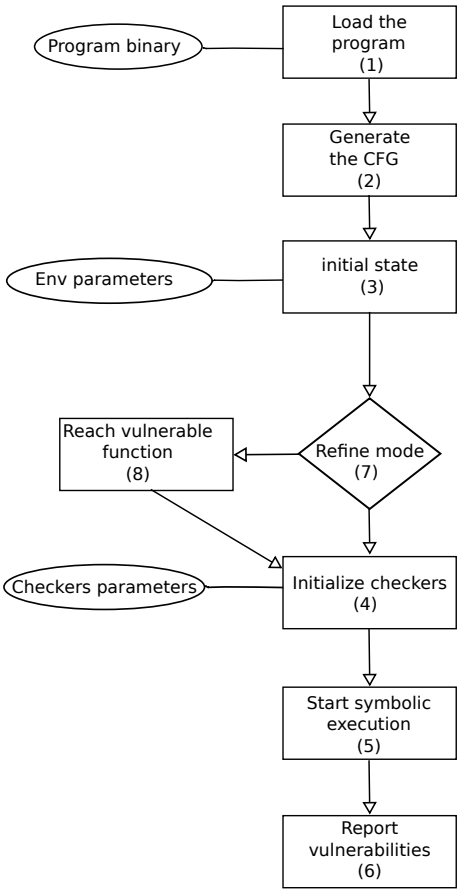

FIGURE 6.3: Vyper general algorithm

1. *Load the program.* The binary of the analyzed program which is given as an argument is loaded by calling the *angr.Project* class. The option *auto_load_libs* is set to false. This will disable loading system libraries. All called system functions must be stubbed to have a correct analysis.

2. *Generate the CFG (Control Flow Graph).* CFG is generated based on the previously loaded program. We make use of *CFGFast* class of *angr*.

3. *Prepare program initial state.* Before launching the analysis we initialize analyzed program state by giving the following information:

   - Analyzed program parameters (argvs) and environment variables: they can be concrete values, symbolic values or a mix of both. All this information is specified in a file describing the environment given as an argument to VYPER, or set to default values if no file is given.

- Standard input, files, network sockets: to be initialized as specified by the user in the file describing the environment given as input to Vyper, or set to default symbolic values. The *angr* framework gives the possibility to use a concrete file system and directly interact with real files.
- The program entry point: if we analyze a whole program and not only a part, this argument must not be set, letting *angr* guessing automatically the entry point directly from the binary headers.

4. *Initialize checkers.*

   Initializing checkers is done by inserting *breakpoints* that, when hit, will trigger the convenient check routine, thus detecting the vulnerability and collecting necessary reporting information. For example, to detect a *tainted argument* we will insert a *breakpoint* that fires when a call to a vulnerable function (printf, fprintf, etc.) is performed. This breakpoint will call *check_tainted_arg* that will report a format string vulnerability. Checkers are all deactivated by default and activated only via the corresponding VYPER argument. This structure allows VYPER to be easily extended by new vulnerabilities checks by adding a breakpoint that will launch checks on program state and report the vulnerability if these checks pass. Since the checks can be activated or deactivated, the analysis can be faster if we are interested only in special category of vulnerabilities checked by VYPER.

5. *Start concolic execution.* In this step, the concolic execution is launched and continued until all the CFG is covered. This is done via the *PathGroup* class of *angr* framework.

6. *Report vulnerabilities.* When vulnerabilities are detected they are not directly reported (only a small notification is emitted in the execution log). The found vulnerabilities and their related data (location, call stack, input values, etc.) are stored in memory and reported in the requested format (txt, xml, html, etc.) at the end of the analysis. The location of the reported vulnerability can be a precise location related to the analyzed application source

when debug information is present in the binary file[1]. When this is not the case the reported location is an address in the binary code.

7. *Refine mode.*

   VYPER can be used in *check mode* to find exploitable vulnerabilities or in *refine mode* to refine vulnerability reports obtained by some other tools in order to eliminate false positives. The corresponding reports are fed as input along with the program to analyze. This feature can allow an application auditor to specify vulnerabilities that should not occur at some code location, e.g. searching for exploitable stack overflows in authentication related code can be very useful to grant the security of the whole application.

8. *Reach vulnerable function*

   This part is executed when VYPER is running in *refine mode*. Using the *PathGroup* class of *angr* framework, the analyzed program is explored without any checker activated until the execution reaches the vulnerable function where the necessary checkers will be activated.

6.3 Synthesis

The implementation and evaluation of the static analysis methods proposed in chapters 4 and 5 show their effectiveness and their added value to the static analysis and application security fields. Carto-C was extended to cover new vulnerabilities and this extension was confirmed by experimental results. Vyper also was able to correctly locate exploitable vulnerabilities on application binary code of several custom tests, Juliet test cases and widely used applications and libraries.

The two tools developed along this thesis show the effectiveness of static analysis framework use. The use of Frama-C [10] and angr [68] frameworks helps to optimize the development effort. These frameworks deal with the tasks of loading

[1]Debug information is present in executable files compiled with the flag *-g* using GCC

programs, building internal representation and even doing some basic computations. Thus, the main development was directed on the proposed static analysis method and did not deal a lot with low level technical details.

The experimentation and developments done along this research were performed on a medium size machine with an x86_64 Intel(R) Core(TM)2 processor, 8 GB RAM and standard Gentoo distribution. Ocaml is needed to compile Carto-C and Python 2.7 is needed to run Vyper. The *angr* framework is installed with the *pip* installer.

Chapter 7

Conclusion

"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."

Antoine de Saint-Exupéry

To improve the security of an information system we were interested in detecting security vulnerabilities and especially those affecting C language applications. We showed through experiments that many of dangerous security vulnerabilities are caused by the misuse of functions of the C libraries. This misuse can correspond to calling a library function in wrong way, or exposing the call to library functions to data controlled from external sources. We have shown that detecting such errors with an automated tool is possible by implementing checks for the proposed security properties. On synthetic test cases results were satisfying with with a limited false negative ratio (maximum 7%) and a small ratio of false positives (between 10% and 51% in the worst case). This contribution helped the development of new functionalities in the Carto-C [97] vulnerability knowledge base. The experimentation on real application sources showed Carto-C ability to analyze real life code and highlighted its limitations. These limitations are inherited from Frama-C [10] or related to features not implemented yet in Carto-C.

We continued the started work but on binary code and searching for exploitable vulnerabilities. We have shown the effectiveness of using state-of-art [68] binary-code analysis framework to detect exploitable vulnerabilities. The method we propose tries to recognize the common patterns present in application behavior allowing successful exploits. These patterns are searched using concolic execution engine provided by the *angr* framework. The implemented tool Vyper performs well on synthetic test cases as well as on real life applications.

7.1 Lessons learned

The full list of lessons learned is long to enumerate. We tried to focus on the most important lessons, which we present below.

- *Analyze a language specification*
  Reading and analyzing the C99 [39] specification in order to locate security vulnerabilities was a rich experience. This task allowed to develop the critic sense needed by a security researcher. In fact, the process of discovering inconsistencies and flaws in requirement specification is widely used by critical software designers. The functional hazard analysis [113] is an example of a such process. In this thesis we shown the effectiveness of a such process on a programming language specification in order to locate security flaws causes.

- *x86_64 binary programs details and structure*
  The development of Vyper tool was very constructive and allowed to dive in x86_64 assembly code, program memory layout and compiler behaviors and their effects on the generated binary code. The understanding of vulnerable code samples permitted to deduce their general patterns from local observations. While developing Vyper, x86_64 architecture documentation was used several times to tune the code we develop to this architecture. Documents on ARM and MIPS architectures were read to see the differences and future developments are planned.

- *Unit testing to minimize regression*

  Unit testing the tools developed was very useful to locate bugs as early as possible. It permitted to confirm that new development are doing what they should do and not interfering with past developments. For each developed tool, Carto-C and Vyper, a test benchmark was developed and extended regularly. This benchmark served as an oracle to confirm the correctness of newly developed functionalities and non-regression for old ones.

- *Importance to have good framework documentation*

  The use of *Frama-C* and *angr* frameworks permitted to produce working tools rapidly. The documentation of these frameworks was necessary to efficiently use them. The documentation explaining the frameworks code and showing clearly their interfaces (inputs, outputs) was helpful to be able to reuse these frameworks functions and classes in order to develop the new static analysis methods described in chapters 4 and 5.

7.2 Open questions and major problems

The domain of static analysis to detect security vulnerabilities has a lot of gaps to fill and areas to explore such as:

- How to specify precisely what a tool detect, and how this may enhance the security of an application.
- Methods and techniques to deal with the problem of “False positive”.
- The problem of analysis speed and how it can scale for huge and complex modern applications.
- How to report the detected vulnerability in an unambiguous way.
- How to be able to analyze a specific language (the front end problems).

7.3 Future work

A lot of ideas for future work and possible improvements come into mind after a good understanding of static analysis problematics. At the time of this writing the most important ideas for future work are:

- **Finalization and deployment of developed tools**: it is figured out that we will continue the tunning of the developed tools in order to be deployed in commercial projects. Tool input channels must be rebuilt to facilitate tool launching via command line or via a GUI (Graphic User Interface). The tool output also need some development effort. The returned vulnerability report format and content must have an easy integration with application development environment.

- **Deliberately inserted vulnerability detection**: we will try to apply the general methodology developed in the chapter 5 to discover other challenging hard to detect security vulnerabilities. We have special interest in detecting *back-doors* or deliberately inserted vulnerabilities. In the cyber security taxonomy this is called the *Insider threat*. This threat is very challenging and it is gaining a growing share in the threats which corporations must deal with. Ninety percent of organizations feel vulnerable to insider attacks [114]. To detect such vulnerability we may use tricks used by insiders [115] to build general back-door behavioral patterns and search for these patterns among the computed application behavior traces.

- **Static analysis for malware detection**: the manipulation of binary code in the second contribution leads to imagine how static analysis can be used to recognize malware by statically fingerprinting its behavior and matching it with known malware behavior signature. So, this technique uses the old technique of malware signature matching. Not on files content and hashes, but on the behavioral signature. This technique may easily detect new malware variants if they keep using of the same behavioral tactics. To go further, the malware behavior patterns knowledge base can be filled automatically

by running a behavior extraction engine on large base of known malware codes such as this repository [116] that contains hundreds of real life malware samples.

- **Detection of security vulnerability on ARM and MIPS architectures**: Vyper can be extended to cover different architectures. The main components of Vyper will remain the same. The implementation details may be extended to cover the differences between newly added architectures. The result will be a tool with ability to detect exploitable vulnerabilities on many architectures. We are interested in ARM architecture which is used by billions of smart phones and IoT (Internet of Things) connected devices. The MIPS architecture running on a wide variety of appliances such as: modems, routers, set-top boxes or even network firewalls is also a prominent architecture to study.

Appendix A

Résumé en Français

Introduction

Dans un monde de plus en plus numérisé, la sécurité des systèmes informatiques devient primordiale. La sécurité d'un système informatique repose sur trois critères : la confidentialité, l'intégrité et la disponibilité. La sécurité est une propriété globale du système qui doit être assurée sur différents niveaux. C'est comme une chaîne qui est autant faible que son maillon le plus faible. Une des stratégies qui peut être utilisée pour garantir la sécurité d'un système est la défense en profondeur. Ceci veut dire que chaque composant du système essaie de garantir sa propre sécurité. Des logiciels qui ne contiennent pas de failles de sécurité sont nécessaires pour avoir un système sécurisé. Mais malheureusement il n'est pas simple d'avoir un logiciel sans faille. La taille et la complexité des logiciels modernes rendent la recherche et la correction de faille de sécurité dans un logiciel une tâche très fastidieuse. Cette situation nécessite d'avoir des outils et des techniques permettant la détection et la prévention de vulnérabilités dangereuses le plus tôt dans le cycle de développement logiciel. L'analyse du code et plus spécialement l'analyse statique est une des réponses à cette problématique. Dans cette thèse nous explorons cette technique. Nous proposons de nouvelles méthodes et nous

développons et évaluons de nouveaux outils d'analyse statique.

L'analyse statique pour la sécurité est le sujet central de cette thèse. Trois différents sujets ont été abordés :

- **La détection de vulnérabilités de sécurité dans les applications développées en langage C.** Dans cette contribution nous utilisons l'interprétation abstraite étendue avec des propriétés vérifiant les vulnérabilités de sécurité. Cette extension couvrant les vulnérabilités de sécurité présente un apport comparée aux outils existants tels que Polyspace [8] ou Astrée [12]. Le deuxième apport est que ces propriétés ont été construite à partir de la spécification du langage C et non pas à partir de motifs de vulnérabilités déjà connues comme c'est le cas avec Fortify [13] ou Coverity [14]. Cette méthode a été implémentée dans l'outil Carto-C et évaluée sur des cas de tests synthétiques (base Juliet [16]) et des applications réelles.

- **La détection de vulnérabilité exploitable sur le code binaire.** Dans cette contribution, nous présentons une méthode pour la détection de vulnérabilité exploitable avec le minimum de faux positifs. La solution proposée utilise *l'exécution concolique* pour explorer les chemins d'exécution de l'application analysée et calculer les états atteignables. Sur ces états est effectuée une recherche de motifs de vulnérabilités exploitables. Une fois une vulnérabilité détectée, les données d'entrée qui permettent de la déclencher sont retournées pour simplifier sa vérification manuelle. Cette méthode peut être aussi utilisée pour trier automatiquement les vulnérabilités détectées par un autre outil d'analyse. Cette méthode a été implémentée et évaluée dans l'outil Vyper.

- **Implémentations et évaluations d'outils d'analyse statique.** Dans cette partie nous présentons tous les développements et expérimentations qui ont été effectués durant cette thèse. Les détails de conception et d'implémentation de Carto-C et de Vyper sont dévoilés. La validité de ces implémentations est testée en utilisant la base de tests orientée sécurité Juliet.

L'efficacité des outils développés est évaluée en analysant des logiciels réels très utilisés par les utilisateurs des technologies d'information.

Organisation de la thèse

Dans le chapitre 1, les domaines de la cyber sécurité sont introduits, en montrant la place de l'analyse statique parmi ces domaines. Dans le chapitre 2 des exemples montrant la complexité du langage C et leurs effets sur la sécurité des application écrites dans ce langage sont présentés. Le chapitre 3 est consacré aux techniques et outils d'analyses statiques existants. Dans le chapitre 4, nous détaillons notre contribution sur la détection de vulnérabilité de sécurité dans les applications développées en C. Dans le chapitre 5, nous présentons la solution proposée pour la détection de vulnérabilités exploitables au niveau du code binaire. Les outils développés et les expérimentations menées sont présentés dans le chapitre 6. Une conclusion ainsi que les perspectives sont dressées dans le chapitre 7.

Introduction à la cyber sécurité

«La sécurité est un processus, ce n'est pas un produit. »

Bruce Schneier.

Le nombre croissant de systèmes informatiques interconnectés et la forte dépendance envers ces systèmes font surgir de nouveaux risques sur l'activité économique, politique, sociale et même sécuritaire. La cyber sécurité est le domaine qui s'occupe de la protection de ces systèmes. Cette protection est à la fois au niveau physique, au niveau logiciel et au niveau des données traitées ou stockées. Ceci rend ce domaine d'une grande importance pour tous les acteurs des technologies d'information. La cyber sécurité englobe des domaines techniques tels que : les opérations et l'architecture de sécurité, des domaines réglementaires tels que la gouvernance et

la conformité au standards et aux normes, et des domaines relevant de l'humain tels que le développement de carrière et la formation d'utilisateurs. Sous le domaine de *l'évaluation de risques*, il existe des sous-domaines tel que le *scan des vulnérabilités* qui s'intéresse à la recherche de vulnérabilités connues en utilisant des outils spécialisés [19] tels que Nmap, Nessus, Qualys, etc. À l'inverse, le *scan du logiciel* a pour but de rechercher les nouvelles vulnérabilités en boite blanche (avec accès au code source) ou en boite noire (sans accès au source). Le *scan du code source* peut intervenir très tôt dans le cycle de développement logiciel garantissant ainsi un logiciel déployé sécurisé. Les techniques et outils qui peuvent être utilisés pour détecter les vulnérabilités sont décrits en détail dans le chapitre 3. Une vulnérabilité est une faille dans un système permettant à un attaquant de compromettre une ou plusieurs propriétés basiques de sécurité (Confidentialité, Intégrité et Disponibilité). Une vulnérabilité exploitable est l'intersection de trois éléments : (1) une vulnérabilité de sécurité, (2) un attaquant qui puisse accéder et (3) la capacité à l'exploiter. Ainsi, l'attaquant doit avoir au moins une technique ou un outil qui puisse atteindre la faiblesse d'un système.

Les causes et les sources des vulnérabilités sont diverses et variées et affectent toutes les phases du cycle du développement d'un logiciel. Les plus importantes sont :

- La négligence des aspects de sécurité pendant la conception d'un logiciel.
- L'utilisation des langages de programmation intrinsèquement dangereux et source de vulnérabilités tel que le langage C/C++ et l'assembleur.
- Les erreurs d'origines humaines telles que les erreurs de frappes ou la confusion des noms de variables par exemple.
- Tests unitaires et tests d'intégration insuffisants et non orientés vers les aspects de sécurité.
- L'insertion de vulnérabilité de manière intentionnelle par un acteur malveillant. Ce type est communément appelé porte dérobée ou *back-door*.

Pour éradiquer les vulnérabilités logicielles, plusieurs actions et décisions peuvent être prises par la communauté du logiciel. Ces actions et décisions peuvent être d'ordre organisationnel, humain ou technique. Parmi ces actions, nous pouvons citer :

- L'inclusion des aspects de sécurité dès les premières étapes de conception d'un logiciel.
- Le choix adéquat d'un langage de programmation sécurisé quand c'est possible.
- L'utilisation de règles de codage tels que le CERT-C [32] qui rendent l'usage du langage C moins risqué.
- Tests unitaires et d'intégration rigoureux couvrant les aspects de sécurité conduits manuellement ou assistés avec des outils.
- L'analyse du code source sans exécution (l'analyse statique) ou avec une exécution contrôlée (analyse dynamique). C'est cette technique, et plus spécialement l'analyse statique qui a été utilisée durant cette thèse.

Les sources de vulnérabilités dans le langage C

Les développeurs en langage C peuvent produire un code correct et fonctionnel en connaissant juste une partie de la spécification de ce langage. Les détails de la sémantique de ce langage permettent de découvrir sa complexité et comment commettre facilement une erreur qui aura de grands effets sur la sécurité.

Le langage C est un langage de programmation impérative à usage général. Ce langage a été standardisé en 1989 par ANSI (American National Standards Institute) devenu le ANSI C. Il a été aussi normalisé par ISO (International Organization for Standardization) appelé ISO C ou le C99. Le langage C a plusieurs extensions appelées par les développeurs *flavors* ou dialectes à savoir : ANSI C, Posix C, Gnu C et Windows C. Pour montrer la complexité de la sémantique du langage

C, nous nous sommes intéressés à trois catégories de fonctions fournies par sa bibliothèque standard. Ces catégories sont : les fonctions d'E/S formatées, les fonctions d'exécution de commandes et les fonctions de manipulation de mémoire.

Les fonctions d'E/S formatées sont un type spécial de fonctions permettant d'afficher ou de lire des données dans un format lisible par les humains. Les informations utilisées sont extraites du standard ANSI ISO/IEC 9899:TC2 (nommé aussi C99) [39]. L'argument du format a une syntaxe et sémantique très précise.
Certaines mauvaises utilisations de fonctions de format conduisent à un comportement indéfini [45] de l'application. Les problèmes de sécurité sont liés à un argument de format avec un contenu contrôlé de l'extérieur de l'application par un potentiel attaquant. La même procédure d'analyse aussi conduite sur Posix, Gnu et Windows du langage C révèle des différences et des sources de vulnérabilités à prendre en considération.

Nous avons effectué une étude sur les fonctions du langage C permettant l'exécution de commandes systèmes. Cette fonctionnalité existe dans la majorité des langages de programmation. Cette fonction consiste à exécuter une commande via l'invite de commande par défaut du système. Les fonctions de lancement de programmes sont aussi considérées dans cette catégorie. Différentes fonctions d'exécution de commandes sont implémentées dans les différents dialectes du C. ANSI C (C99) [39], POSIX C [41], GNU C [42]. À partir de la spécification de ces fonctions, les problèmes suivants peuvent se poser :

- Le comportement de la fonction *system* est défini par l'implémentation et peut être source d'instabilité pour le code appelant cette dernière.
- Un appel à la fonction *system* est intrinsèquement dangereux.
- Un appel à une fonction d'exécution de commande avec un argument provenant des données d'entrées peut conduire à une exécution arbitraire de commande sur la machine cible.
- Même quand la commande n'est pas contrôlée par l'attaquant, il peut toujours interférer avec l'application attaquée en modifiant son environnement.

Nous nous sommes également intéressés aux fonction de manipulation de mémoire. Les fonctions qui prennent comme argument un pointeur vers des zones mémoires du type (char *, void *, int *, etc.) sont concernées. Par exemple : *malloc, calloc, memcpy, strcpy, memmove* sont des fonctions de manipulation de mémoire. Le langage C et ces différents dialectes : C99 [39], Posix [41], Linux [43] contiennent des dizaines de fonctions de manipulation de mémoire. Les problèmes de sûreté sont directement extraits de la description des fonctions étudiées. Ce sont les situations qui conduisent à des comportements indéfinis, non-spécifiés ou définis par l'implémentation. Les problèmes de sécurité ne sont pas directement décrits dans les spécifications et doivent être déduits en utilisant une analyse critique sur comment en tirer profit d'un point de vue attaquant des erreurs possible d'utilisation des ces fonctions. Plus de détails sur la qualification et la détection de ces vulnérabilités seront donnés dans le chapitre 4.

Synthèse

L'analyse du langage C et de ces différents dialectes fait apparaître plusieurs sources de vulnérabilités de sécurité. Se rappeler de toutes les règles de bonne utilisation et les mettre en pratique est une tâche difficile. La vérification automatique de ces règles peut largement améliorer la sécurité d'une application écrite en C. L'une des techniques d'analyse automatique de code est *l'analyse statique* qui est introduite dans le chapitre 3. Sont présentées dans les chapitres 4, 5 et 6 la conception, l'implémentation et l'évaluation d'outils d'analyse statique.

Les outils d'analyse statique

La problématique de détection de vulnérabilité logicielle peut être abordée en utilisant plusieurs stratégies, techniques et outils. L'une des techniques utilisées est l'analyse du code et plus précisément l'analyse statique.

L'analyse statique est une technique qui consiste à analyser un code sans l'exécuter. Elle peut être utilisée pour trouver des erreurs de programmation tels que les dépassement de tampons, l'utilisation de données non initialisées, le débordement d'entier, etc. Suivant la méthode et les algorithmes utilisés elle peut permettre de détecter des vulnérabilités qui ne sont pas simples à trouver avec la revue manuelle de code.

L'outil d'analyse statique est un outil prenant en entrée un code source et retournant en sortie un rapport d'analyse. Généralement un outil d'analyse statique est composé de trois parties : le front-end, le middle-end et le back-end. Le front-end permet de lire les codes source donnés en entrée et produit une représentation interne qui est passée au middle-end, là où tous les calculs sont effectués. Le back-end reprend les calculs effectués par le middle-end et génère un rapport de résultats à retourner à l'utilisateur. Cette structure est très similaire à la structure d'un compilateur, car un compilateur est un cas spécial d'outil d'analyse statique.

Différentes techniques et algorithmes sont utilisés par les outils d'analyse statique. Les premiers outils d'analyse statique utilisent une simple recherche de motifs syntaxiques afin de retrouver des motifs connus sources de vulnérabilités. Par exemple l'outil Flawfinder [69] qui est un outil naïf cherchant certaines chaînes de caractères dans le code à analyser sans se soucier de sa structure ou sa sémantique. Malgré sa simplicité, il est capable de trouver certaines vulnérabilités mais avec un très grand taux de fausses alertes appelées *faux positifs*.

Des outils améliorés tels que Lint [70] ou CppChek [71] génèrent une représentation abstraite du code analysé. Sur cette représentation, différents algorithmes et heuristiques d'analyse du flot de contrôle et de flot de données permettent de détecter des vulnérabilités avec un taux de faux positifs inférieur à la méthode précédente. Cette méthode a aussi ses limites et ne peut pas détecter des vulnérabilités non-triviales dans un code complexe.

Une des techniques modernes d'analyse est *l'interprétation abstraite* qui crée une nouvelle sémantique du code analysé. Cette nouvelle sémantique associe à chaque variable du programme son intervalle de valeurs possibles. Cette technique sert

pour prouver l'absence de certaines catégories d'erreurs. Elle est intéressante pour les problématiques de sûreté de programmes effectuant une tâche critique.

L'exécution symbolique qui consiste à transformer le programme en une formule logique en fonction des variables d'entrée est une autre technique moderne. La résolution de cette formule permet de statuer sur les prédicats de sécurité recherchés. L'outil CBMC [75] utilise cette technique sur des programmes en langage C.

Une autre technique est l'exécution concolique (concrète + symbolique) qui effectue une exécution symbolique en même temps avec une exécution concrète quand cela est possible [15].

Des outils d'analyse statique utilisés par l'industrie du logiciel tels que Fortify [13] ou Coverity [14] utilisent plusieurs techniques parmi celles citées précédemment combinées avec des algorithmes heuristiques pour atteindre les meilleures performances sur les codes analysés.

Le choix d'outil d'analyse statique peut s'avérer parfois compliqué. Pour cela le CAS (Center for Assured Software) de la NSA (National Security Agency) a développé une base de tests Juliet [16] spécialement conçue pour évaluer les capacités d'outils d'analyse statique. Plusieurs recherches ont été aussi effectuées pour évaluer et comparer des outils d'analyse statique [83] ou présentant des méthodologies d'évaluation [81].

Contribution 1. La détection de vulnérabilités de sécurité dans les applications développées en C

La sécurité des systèmes informatiques revêt une importance considérable dans un monde toujours plus dépendant du numérique. Les données des citoyens, entreprises, et même des gouvernements sont sous la menace croissante d'attaques de plus en plus sophistiquées. L'éradication des vulnérabilités dans le code source

est une brique de base de la sécurité des systèmes informatiques. La première étape de cette élimination est la détection de ces vulnérabilités, avant qu'elles ne puissent être découvertes par un adversaire malveillant. Cette détection est toutefois une tâche ardue, plus particulièrement dans des programmes de grande taille. Elle peut être automatisée avec l'aide de l'analyse statique du code source.

L'analyse statique et la détection de vulnérabilité de sécurité

De nombreuses solutions ont été proposées pour détecter ou empêcher des erreurs de programmation qui entraînent les vulnérabilités. Tout d'abord, le choix du langage de développement a un très grand impact sur la sécurité de l'application implémentée. Ceci a été montré dans les études de sécurité des langages telles que: JavaSec [87] pour Java [88] et LaFoSec [89] pour les langages fonctionnels (OCaml) [90]. Un langage flexible permettant des opérations de bas niveau comme le langage C reste largement utilisé malgré les problèmes de sécurité bien connus intrinsèques à ce langage. Pendant le développement d'une application, des règles de codage comme le *CERT C coding rules* [32] peuvent être utilisées pour aider le développeur à éviter certaines failles de sécurité. Sur le code source, l'analyse statique peut être utilisée pour détecter les vulnérabilités. Cette analyse peut être basée sur l'interprétation abstraite [73] qui permet de prouver l'absence d'erreurs d'exécution (*runtime errors*) [74] qui sont les prémisses de nombreuses failles de sécurité (Polyspace [8, 9], Frama-C [10, 11] ou Astrée [12]) ou sur d'autres techniques qui permettent de détecter des faiblesses (Fortify [13] ou Coverity [14]). Nous nous sommes intéressés aux vulnérabilités du code C provenant de l'utilisation des fonctions de la bibliothèque C. Nous avons défini des propriétés permettant la détection d'appels incorrects ou potentiellement dangereux à de telles fonctions. Ces propriétés se basent sur une analyse de la spécification de ces fonctions. L'outil Carto-C de détection des vulnérabilités qui intègre ces propriétés a été testé sur une base de tests de référence pour évaluer l'efficacité de notre approche.

Propriétés de détection des vulnérabilités

Cette contribution est focalisée sur trois vulnérabilités : les problèmes de chaîne de format, exécution de commande, et problèmes de manipulation de tampons. Pour chaque vulnérabilité, une description générale ainsi que la liste des propriétés qui caractérisent les utilisations des fonctions causant la vulnérabilité sont données. Quand cette propriété est vraie, le code analysé contient une vulnérabilité. Chaque propriété à vérifier est nommée, décrite succinctement, associée à un scénario d'attaque applicable quand la propriété est vraie et à un exemple de code C illustrant la présence de la vulnérabilité.

Une donnée est dite *contrôlée par une entité extérieure* si elle est initialisée via les vecteurs d'entrée de l'application. De même, une variable est dite "tainted" si sa valeur est potentiellement calculée à partir de données *contrôlées par une entité extérieure*.

Discussion et conclusion

Dans cette étude, nous avons montré que certaines vulnérabilités sont causées par la mauvaise utilisation des fonctions de la bibliothèque C. Cette mauvaise utilisation consiste à les appeler incorrectement, ou les exposer à des données *contrôlées via l'extérieur* de l'application. Nous avons démontré que la détection, sur une base de tests synthétiques, de telles vulnérabilités est possible avec un outil automatisé avec très peu de faux négatifs (maximum 7%) et peu de faux positifs (entre 10% et 51% dans le pire cas). Nous avons aussi identifié l'absence de cas de tests pour certaines de nos propriétés dans la base de tests de référence et nous avons développé des cas de tests nécessaires. Nous avons aussi fait l'inventaire des fonctions et fichiers d'entêtes manquants pour pouvoir analyser les cas de tests de la base de référence qui ne sont pas encore analysés et aussi pour pouvoir analyser des applications réelles. Les résultats obtenus sont prometteurs et vont nous aider à rajouter de nouvelles fonctionnalités dans l'outil Carto-C, des fichiers de la bibliothèque C et de nouveaux cas de tests. Parmi les futurs travaux, nous souhaitons

définir de nouvelles propriétés pour détecter de nouvelles vulnérabilités mais aussi donner à l'utilisateur une meilleure explication de la vulnérabilité trouvée (la trace d'exécution par exemple). Un autre axe de recherche est d'étudier les vulnérabilités causées par des séquences d'appels de fonctions comme la fameuse séquence malloc/free qui est la cause de plusieurs vulnérabilités tel que : use-afte-free [117] ou double-free [118].

Contribution 2. La détection de vulnérabilités exploitables au niveau du code binaire

Dans cette contribution nous présentons notre méthode pour la détection de vulnérabilités exploitables avec un taux très faible de faux positif. Cette solution utilise l'exécution concolique pour rechercher le motif des comportements de codes vulnérables. Nous avons étudié trois catégories de vulnérabilités : celles liées à des données "tainted", le débordement de pile et le débordement du tas. La méthode proposée a été implémentée en utilisant le framework *angr* [68]. L'outil développé a été testé sur les cas de tests de la base Juliet et sur les binaires d'application réelles permettant une détection de vulnérabilités exploitables avec un taux de faux positifs assez bas.

L'exécution concolique et l'analyse du code binaire

L'exécution concolique décrite précédemment dans le chapitre 3 a été introduite par Godefroid et al. [15] pour assister le test aléatoire à atteindre une couverture maximale. Elle a été aussi utilisée par les développeurs de KLEE [76] qui est un outil pour la génération automatique de cas de tests. L'outil AEG (Automatic Exploit Generation) de Avgerinos et al. [77] utilisent aussi l'exécution concolique pour la génération automatique d'attaques contre des vulnérabilités trouvées

dans les codes sources. Shoshitaishvili et al. [78] dans leur recensement des techniques d'analyse du code binaire présentent un panorama des techniques sur la détection de vulnérabilités exploitables [78] et la majorité de ces techniques sont implémentées dans le framework open-source *angr* [68].

Vulnérabilités exploitables

Dans le cadre de cette contribution nous avons considéré comme une vulnérabilité exploitable, toute vulnérabilité qui permet à un attaquant d'exécuter du code arbitraire. Une exécution de code arbitraire signifie qu'un attaquant peut via les vecteurs d'entrée d'une application (arguments de lignes de commandes, variable d'environnement, système de fichiers ou communications réseaux) prendre le contrôle du flot d'exécution et le réorienter vers du code machine qu'il a lui même fourni via ces vecteurs d'entrées. La capacité d'un attaquant à avoir le contenu de la mémoire à une adresse arbitraire via les vecteurs de sortie va lui permettre d'outrepasser des mécanismes de protection d'application tels que ASLR (Address Space Layout Randomization) [103] ou DEP (Data Execution Protection) [104]. Les attaques causant uniquement le crash de l'application attaquée ou celles qui interfèrent avec la logique de haut niveau implémentée dans l'application ne sont pas considérées dans le cadre de cette contribution. Nous nous sommes focalisés sur les vulnérabilités les plus exploitées et largement documentées par les différentes communautés de pirates à chapeaux blancs ou gris comme Metasploit [105], Exploit-DB [106], ou la distribution Pentesting de Linux Kali [107]. Les vulnérabilité suivantes sont considérées :

1. *Les vulnérabilités liées à des données "tainted".*

   Les vulnérabilités de format et d'exécution de commandes sont causées par les appels de fonction avec des arguments "tainted". Ces vulnérabilités seront exploitables si l'argument en question n'a pas été "sanitisé" ou a été "mal-sanitisé". Pour détecter ces vulnérabilités, il suffit de surveiller le contenu de l'argument sensible et reporter une vulnérabilité si sa valeur au moment d'un appel est "tainted".

2. *Débordement de pile.*

   Le débordement de pile apparaît quand un accès hors borne sur un tableau alloué sur la pile est effectué. Un accès hors borne est exploitable quand il permet de modifier l'adresse de retour stockée sur la pile avec une valeur "tainted" fournie via les entrées. Pour détecter ces vulnérabilités, on peut annoter l'emplacement de l'adresse de retour et reporter une vulnérabilité si cet emplacement est écrit avec une valeur "tainted".

3. *Débordement de tas.*

   Quand une application alloue de la mémoire dynamique. Le système d'allocation garde à coté du tampon alloué des métadonnées qui permettent une gestion optimisée de l'espace mémoire. Quand un débordement sur ce type de mémoire corrompt ces métadonnées, un attaquant peut prendre le contrôle du système d'allocation de mémoire et l'utiliser pour exécuter un code arbitraire. Pour détecter ce type de vulnérabilité on peut annoter ces métadonnées et rapporter une vulnérabilité quand ces données sont écrasées par des données "tainted".

À partir des exemples précédents, est dressé un motif général de vulnérabilité exploitable. Ce motif est le fait que certaines zones mémoires sensibles sont remplacées par des données "tainted". Ces zones mémoires ne sont pas fixées au départ de l'application mais leur apparition dépendent du comportement de l'application. La détection de ce motif peut s'effectuer en trois étapes :

- Le calcul des traces d'exécution du programme analysé via l'exécution concolique.
- L'annotation de zones mémoires sensibles pour chaque trace d'exécution.
- La vérification pour que chaque opération exécutée de l'application, les données sensibles annotées ne sont pas corrompues via des données 'tainted'. À chaque fois que cette propriété est fausse, une vulnérabilité est rapportée.

Le type de cette vulnérabilité dépend du type de la mémoire sensible corrompue (arguments de fonctions dangereuses, adresse de retour, métadonnées du tas).

Implémentation et expérimentation

Notre méthode a été implémentée dans notre outil Vyper (VulnerabilitY detection based on dynamic behavioral PattErn Recognition). Vyper est un outil qui permet d'analyser le code binaire afin de détecter certaines catégories de vulnérabilités exploitables. Il est implémenté en utilisant le framework *angr* [68]. Les détails de la conception, l'implémentation et l'expérimentation de Vyper sont donnés dans le chapitre 6.

Conclusion et discussion

Cette contribution a montré l'efficacité de l'utilisation des techniques d'analyse du code binaire pour détecter des vulnérabilités exploitables. La méthode proposée cherche les motifs de vulnérabilités exploitables dans le comportement du programme analysé. Cette méthode a été implémentée en utilisant le framework *angr* et testée avec succès sur les cas de test de la base Juliet [16] et des applications réelles.

L'une des limitations de la méthode développée est sa lenteur comparée aux autres méthodes d'analyse. Aussi dans l'état actuel de l'implémentation, Vyper ne peut pas analyser efficacement des codes binaires de grande taille (plus de 5 Mo) ou des codes complexes.

Contribution 3. Implémentation et expérimentation d'outils

Implémentation et expérimentation de Carto-C

Carto-C est un outil développé par Saferiver [119] basé sur Frama-C [10]. Il est destiné à détecter différentes vulnérabilités de sécurité ou de sûreté. Carto-C permet d'identifier les appels de fonctions de la bibliothèque pouvant être à l'origine d'une des vulnérabilités étudiées. Cette identification est basée sur les propriétés proposées dans le chapitre 4.

Tester l'efficacité et la précision d'un outil d'analyse statique est toujours une tâche ardue, comme l'indique le rapport du projet SAMATE SATE IV [98]. Notre implémentation a été testée à la fois sur des cas de tests synthétiques de référence, et sur des exemples réels.

Cas de tests synthétiques. Notre outil a été testé sur la base de cas de tests Juliet [16], initialement créée pour le projet CAS [99] de la NSA et utilisée depuis par le projet SAMATE SATE [100] du NIST. Cette base de tests contient plus de 40000 exemples de code C, couvrant plus de 100 CWEs. De plus, chaque exemple de code apparaît en deux versions : l'une faisant apparaître la vulnérabilité (*flaw*, erronée), l'autre dans laquelle la vulnérabilité n'apparaît pas (*fix*, réparée).

Les tests utilisés pour Carto-C sont un sous-ensemble de la base Juliet couvrant les CWEs traitées dans notre thèse.

Cas de test réels. Des tests sur des applications open-source ont été effectués, telles que : tar 1.13.19, libpng 1.2.40 et drivers/net/wireless/broadcom/brcm80211.

Les difficultés rencontrées dans le test d'applications open-source ont été de deux natures. Tout d'abord, ces programmes utilisent beaucoup les bibliothèques GNU C

et POSIX C, qui ne sont pas encore supportées par Carto-C. Par ailleurs, Frama-C ne fonctionne pas toujours sur les programmes de grande taille et n'implémente pas certaines structures de programmation souvent utilisées dans le code source, telle que la récursivité. Nos développements futurs portent donc sur l'extension de Carto-C à ces bibliothèques ainsi que sur les problèmes de passage à l'échelle.

Implémentation et expérimentation de Vyper

Nous avons développé l'outil *Vyper* en utilisant le framework *angr*. Nous nous sommes servis de la fonctionnalité d'exécution concolique offerte par ce framework. Ceci a permis de se focaliser sur la partie analyse et ne pas se préoccuper de tous les détails d'un moteur d'exécution concolique.

angr est un framework disponible en open-source pour l'analyse du code binaire. Il implémente la majorité des techniques d'analyse connues par ailleurs. Ceci permet de réutiliser ces techniques sans avoir à les re-développer à nouveau. Parmi les techniques proposées : un loader de code binaire pour différentes architecture, le calcul du graphe du flot de contrôle (CFG) , le calcul du graphe du flot de données (DFG), l'analyse de l'ensemble de valeur (VSA), l'exécution concolique en utilisant les explorateurs de chemins et aussi les "break-points" conditionnels.

L'outil est développé en Python. Le code est structuré en différentes fonctions. Par exemple, pour chaque classe de vulnérabilités nous avons au moins une fonction qui fait l'annotation et une qui fait la détection. Ces fonctions sont activées en utilisant des break-points conditionnels déclenchés pendant l'exécution concolique qui est lancée et contrôlée à partir de la fonction principale. C'est cette fonction aussi qui parse les arguments d'entrée, prépare l'état initiale et collecte les résultats d'analyse.

Les spécifications de Vyper sont :

- Entrée : le code binaire du programme, la fonction de début d'analyse, le mode d'analyse (optionel, par defaut : détection), classe de vulnérabilité recherchée (optionel, par défaut : tout).
- Sortie : rapport de vulnérabilités trouvées.
- Exigences :
  - Charger le code binaire et initialiser l'état de départ d'analyse.
  - Activer les vérifications et annotations nécessaires suivant les classes de vulnérabilités demandées.
  - Lancer, suivre et contrôler l'exécution concolique.
  - Si une opération d'annotation est déclenchée, appeler la routine correspondante et stocker les informations annotées.
  - Si une opération de vérification est déclenchée, appeler la routine correspondante et récupérer les informations liée à la vulnérabilité s'il y en une.
  - Quand l'exécution concolique se termine ou dépasse le temps maximal d'exécution, collecter et rapporter les vulnérabilités dans le format demandé.

Pour évaluer Vyper nous avons utilisé trois types de tests :

- Tests spécifiques pour les vulnérabilités recherchées.
- Test en utilisant la base Juliet [16].
- Tests avec des applications réelles.

Tests spécifiques Durant le développement de Vyper nous avons développées quelques cas de tests contenant des vulnérabilités exploitables. Ces tests ont permis de raffiner notre développement et de garantir la non-régression entre les différentes version de Vyper.

Tests sur Juliet Juliet [16] est une collection de codes écrits en C/C++ pour tester les capacités d'outils d'analyse statique. Elle contient des tests pour 118 CWEs. Chaque test contient le code vulnérable "flaw" et le même code corrigé (sans vulnérabilité) "fix" activable via des macros. Aussi les cas de tests sont parfois repris en complexifiant les flots de contrôle et/ou les flots de données pour évaluer la capacité des outils à analyser la sémantique du code analysé.

Pour nos tests, nous avons écrit des scripts spéciaux pour lancer en automatique les tests dans les deux modes "flaw" et "fix", normaliser les résultats d'analyse et synthétisé les résultats sous une forme statistique. Cette procédure de test a été relancée avec d'autres outils d'analyse statique et a permis d'observer la différence concernant ces outils d'analyse et Vyper.

Test sur les applications réelles Pour évaluer l'outil Vyper et connaître ses limitations, nous l'avons testé sur le binaire d'applications réelles :

- *Udhcp server* : udhcp-0.9.8 est un mini-utilitaire qui s'exécute sur une variété d'appareils (routeurs, modems, décodeur TV, cameras IP, etc). Nous avons inséré un appel vulnérable à la fonction *printf* à l'endroit *udhcp-0.9.8/dhcpd.c:102*. Vyper a pu localiser la vulnérabilité avec précision en trois minutes d'analyse. Cette expérimentation confirme l'efficacité de Vyper à tester des logiciels destinés à tourner sur des appareils embarqués.
- *Bibliothèques logicielles largement utilisées* : Vyper a été testé sur: OpenSSL-1.1.0f (libssl.so), libpng-1.5.20 et tiff-3.8.1. Le test consiste à lancer l'outil sur le binaire de la bibliothèque analysée pour chaque fonction exportée et en fixant le temps d'exécution à chaque lancement à 5 minutes.

Cette expérimentation confirme également l'efficacité de Vyper.

Conclusion

Dans la première contribution de cette thèse, nous avons confirmé que certaines vulnérabilités sont causées par la mauvaise utilisation des fonctions de la bibliothèque C. Cette mauvaise utilisation consiste à les utiliser incorrectement par rapport à leurs spécifications, ou les exposer à des données *contrôlées via l'extérieur* de l'application. Nous avons démontré que la détection sur une base de tests synthétiques de telles vulnérabilités est possible avec un outil automatisé avec peu de faux négatifs (maximum 7%) et peu de faux positifs (entre 10% et 51% dans le pire cas). Cette contribution a permis d'étendre l'outil Carto-C pour couvrir plus de vulnérabilités de sécurité. Elle a aussi servi pour connaître les limitations de l'outil développé tels que les problèmes de scalabilité et les problèmes de front-end.

La contribution décrite dans le chapitre 5 montre l'efficacité de l'utilisation de techniques d'analyse du code binaire afin de repérer les vulnérabilités exploitables. La solution proposée était de construire les traces du programme analysé et rechercher les motifs de vulnérabilités exploitables. Cette méthode a été implémentée dans l'outil Vyper qui a réalisé de bonnes performances sur les cas de tests synthétiques et codes binaires d'applications réelles.

Appendix B

Demo of Carto-C and Vyper on Juliet test base

- Vyper launch command.

```
1 #!/bin/sh
2 "juliet/launch-vyper.py" "-tool" "vyper.py" "-out" "juliet/
    cwe134_4/tests/OnPurpose/Juliet/Juliet_baseline/testcases/
    CWE134_Uncontrolled_Format_String/s01/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /BAD/std_results" "-main" "main" "-src" "juliet/cwe134_4/
    tests/OnPurpose/Juliet/Juliet_baseline/testcases/
    CWE134_Uncontrolled_Format_String/s01/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src" "-I" "juliet/cwe134_4/tests/OnPurpose/Juliet/
    Juliet_baseline/testcases/CWE134_Uncontrolled_Format_String/
    s01/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src" "-I" "juliet/src/OnPurpose/Juliet/testcasesupport" "-D"
     "INCLUDEMAIN" "-D" "OMITGOOD" "-compil" "gcc" "-OS" "Linux"
    "-proc" "x86"
```

- Carto-C launch command.

```
1 #!/bin/sh
```

```
2 "Carto-C/toolsScripts/launch-carto-c.py" "-tool" "/home/boudjema
    /install/Carto-C/bin/Carto-C" "-out" "test/res_Carto_C2/tests
    /OnPurpose/Juliet/Juliet_baseline/testcases/
    CWE134_Uncontrolled_Format_String/s01/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /BAD/std_results" "-main" "main" "-src" "test/res_Carto_C2/
    tests/OnPurpose/Juliet/Juliet_baseline/testcases/
    CWE134_Uncontrolled_Format_String/s01/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src" "-I" "test/res_Carto_C2/tests/OnPurpose/Juliet/
    Juliet_baseline/testcases/CWE134_Uncontrolled_Format_String/
    s01/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src" "-I" "extract_juliet/src/OnPurpose/Juliet/
    testcasesupport" "-defaultlibc" "/home/boudjema/labossec" "-
    defaultpreprocessor" "gcc" "-D" "INCLUDEMAIN" "-D" "OMITGOOD"
     "-compil" "gcc" "-OS" "Linux" "-proc" "x86"
```

- Juliet test source code: Juliet/testcases/CWE134_Uncontrolled_Format _String/s01 /CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01.c.

```
1 /* TEMPLATE GENERATED TESTCASE FILE
2 Filename:
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    .c
3 Label Definition File: CWE134_Uncontrolled_Format_String.label.
    xml
4 Template File: sources-sinks-01.tmpl.c
5 */
6 /*
7  * @description
8  * CWE: 134 Uncontrolled Format String
9  * BadSource: connect_socket Read data using a connect socket (
    client side)
10 * GoodSource: Copy a fixed string into data
11 * Sinks: fprintf
12 *    GoodSink: fprintf with "%s" as the second argument and
    data as the third
```

```
13  *    BadSink : fprintf with data as the second argument
14  * Flow Variant: 01 Baseline
15  *
16  * */
17
18 #include "std_testcase.h"
19
20 #ifndef _WIN32
21 #include <wchar.h>
22 #endif
23
24 #ifdef _WIN32
25 #include <winsock2.h>
26 #include <windows.h>
27 #include <direct.h>
28 #pragma comment(lib, "ws2_32") /* include ws2_32.lib when
       linking */
29 #define CLOSE_SOCKET closesocket
30 #else /* NOT _WIN32 */
31 #include <sys/types.h>
32 #include <sys/socket.h>
33 #include <netinet/in.h>
34 #include <arpa/inet.h>
35 #include <unistd.h>
36 #define INVALID_SOCKET -1
37 #define SOCKET_ERROR -1
38 #define CLOSE_SOCKET close
39 #define SOCKET int
40 #endif
41
42 #define TCP_PORT 27015
43 #define IP_ADDRESS "127.0.0.1"
44
45 #ifndef OMITBAD
46
47 void
       CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
       ()
```

```
48 {
49     char * data;
50     char dataBuffer[100] = "";
51     data = dataBuffer;
52     {
53 #ifdef _WIN32
54         WSADATA wsaData;
55         int wsaDataInit = 0;
56 #endif
57         int recvResult;
58         struct sockaddr_in service;
59         char *replace;
60         SOCKET connectSocket = INVALID_SOCKET;
61         size_t dataLen = strlen(data);
62         do
63         {
64 #ifdef _WIN32
65             if (WSAStartup(MAKEWORD(2,2), &wsaData) != NO_ERROR)
66             {
67                 break;
68             }
69             wsaDataInit = 1;
70 #endif
71             /* POTENTIAL FLAW: Read data using a connect socket
      */
72             connectSocket = socket(AF_INET, SOCK_STREAM,
      IPPROTO_TCP);
73             if (connectSocket == INVALID_SOCKET)
74             {
75                 break;
76             }
77             memset(&service, 0, sizeof(service));
78             service.sin_family = AF_INET;
79             service.sin_addr.s_addr = inet_addr(IP_ADDRESS);
80             service.sin_port = htons(TCP_PORT);
81             if (connect(connectSocket, (struct sockaddr*)&
      service, sizeof(service)) == SOCKET_ERROR)
82             {
```

```
83                  break;
84              }
85              /* Abort on error or the connection was closed, make
     sure to recv one
86               * less char than is in the recv_buf in order to
    append a terminator */
87              /* Abort on error or the connection was closed */
88              recvResult = recv(connectSocket, (char *)(data +
    dataLen), sizeof(char) * (100 - dataLen - 1), 0);
89              if (recvResult == SOCKET_ERROR || recvResult == 0)
90              {
91                  break;
92              }
93              /* Append null terminator */
94              data[dataLen + recvResult / sizeof(char)] = '\0';
95              /* Eliminate CRLF */
96              replace = strchr(data, '\r');
97              if (replace)
98              {
99                  *replace = '\0';
100             }
101             replace = strchr(data, '\n');
102             if (replace)
103             {
104                 *replace = '\0';
105             }
106         }
107         while (0);
108         if (connectSocket != INVALID_SOCKET)
109         {
110             CLOSE_SOCKET(connectSocket);
111         }
112 #ifdef _WIN32
113         if (wsaDataInit)
114         {
115             WSACleanup();
116         }
117 #endif
```

```
118     }
119     /* POTENTIAL FLAW: Do not specify the format allowing a
        possible format string vulnerability */
120     fprintf(stdout, data);
121 }
122
123 #endif /* OMITBAD */
124
125 #ifndef OMITGOOD
126
127 /* goodG2B uses the GoodSource with the BadSink */
128 static void goodG2B()
129 {
130     char * data;
131     char dataBuffer[100] = "";
132     data = dataBuffer;
133     /* FIX: Use a fixed string that does not contain a format
        specifier */
134     strcpy(data, "fixedstringtest");
135     /* POTENTIAL FLAW: Do not specify the format allowing a
        possible format string vulnerability */
136     fprintf(stdout, data);
137 }
138
139 /* goodB2G uses the BadSource with the GoodSink */
140 static void goodB2G()
141 {
142     char * data;
143     char dataBuffer[100] = "";
144     data = dataBuffer;
145     {
146 #ifdef _WIN32
147         WSADATA wsaData;
148         int wsaDataInit = 0;
149 #endif
150         int recvResult;
151         struct sockaddr_in service;
152         char *replace;
```

```
153         SOCKET connectSocket = INVALID_SOCKET;
154         size_t dataLen = strlen(data);
155         do
156         {
157 #ifdef _WIN32
158             if (WSAStartup(MAKEWORD(2,2), &wsaData) != NO_ERROR)
159             {
160                 break;
161             }
162             wsaDataInit = 1;
163 #endif
164             /* POTENTIAL FLAW: Read data using a connect socket
        */
165             connectSocket = socket(AF_INET, SOCK_STREAM,
      IPPROTO_TCP);
166             if (connectSocket == INVALID_SOCKET)
167             {
168                 break;
169             }
170             memset(&service, 0, sizeof(service));
171             service.sin_family = AF_INET;
172             service.sin_addr.s_addr = inet_addr(IP_ADDRESS);
173             service.sin_port = htons(TCP_PORT);
174             if (connect(connectSocket, (struct sockaddr*)&
      service, sizeof(service)) == SOCKET_ERROR)
175             {
176                 break;
177             }
178             /* Abort on error or the connection was closed, make
       sure to recv one
179              * less char than is in the recv_buf in order to
      append a terminator */
180             /* Abort on error or the connection was closed */
181             recvResult = recv(connectSocket, (char *)(data +
      dataLen), sizeof(char) * (100 - dataLen - 1), 0);
182             if (recvResult == SOCKET_ERROR || recvResult == 0)
183             {
184                 break;
```

```
185             }
186             /* Append null terminator */
187             data[dataLen + recvResult / sizeof(char)] = '\0';
188             /* Eliminate CRLF */
189             replace = strchr(data, '\r');
190             if (replace)
191             {
192                 *replace = '\0';
193             }
194             replace = strchr(data, '\n');
195             if (replace)
196             {
197                 *replace = '\0';
198             }
199         }
200         while (0);
201         if (connectSocket != INVALID_SOCKET)
202         {
203             CLOSE_SOCKET(connectSocket);
204         }
205 #ifdef _WIN32
206         if (wsaDataInit)
207         {
208             WSACleanup();
209         }
210 #endif
211     }
212     /* FIX: Specify the format disallowing a format string
        vulnerability */
213     fprintf(stdout, "%s\n", data);
214 }
215
216 void
        CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_good
        ()
217 {
218     goodG2B();
219     goodB2G();
```

```
220 }
221
222 #endif /* OMITGOOD */
223
224 /* Below is the main(). It is only used when building this
      testcase on
225    its own for testing or for building a binary to use in
      testing binary
226    analysis tools. It is not used when compiling all the
      testcases as one
227    application, which is how source code analysis tools are
      tested. */
228
229 #ifdef INCLUDEMAIN
230
231 int main(int argc, char * argv[])
232 {
233     /* seed randomness */
234     srand( (unsigned)time(NULL) );
235 #ifndef OMITGOOD
236     printLine("Calling good()...");
237
      CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_good
      ();
238     printLine("Finished good()");
239 #endif /* OMITGOOD */
240 #ifndef OMITBAD
241     printLine("Calling bad()...");
242
      CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
      ();
243     printLine("Finished bad()");
244 #endif /* OMITBAD */
245     return 0;
246 }
247
248 #endif
```

- Vyper result.

```
1 format vuln found at: /home/boudjema/angr/poc/juliet/cwe134_4/
    tests/OnPurpose/Juliet/Juliet_baseline/testcases/
    CWE134_Uncontrolled_Format_String/s01/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    .c:120
2
3
4 Backtrace:
5 Func 0x400a10, sp=0x7fffffffefec0, ret=0x400d7a
6 Func 0x400b8d, sp=0x7fffffffeff70, ret=0x400dc4
7 Func 0x400d90, sp=0x7fffffffeff90, ret=-0x1/n
```

- Carto-C result.

```
1 -- Compute exported from Carto-C version 1.3.4-20161011T135925
2 Loc File,Loc Line,Caller,CWE ID, CWE Description,Type,Status,
    Message
3 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    .c,110,
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
    ,676,Use of Potentially Dangerous Function,CloseFun,KO,Call
    to close
4 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    .c,120,
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
    ,676,Use of Potentially Dangerous Function,FormatFun,KO,Call
    to fprintf
```

```
5 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    .c,120,
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
    ,685,Function Call With Incorrect Number of Arguments,Format
    Match (number of arguments),OK,unknown match
6 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    .c,120,
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
    ,686,Function Call With Incorrect Argument Type,Format Match
    (type of arguments),OK,unknown match
7 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    .c,120,
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
    ,134,Uncontrolled Format String,Control,KO,See 'controls.csv'
     file
8 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    .c,120,
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
    ,676,Use of Potentially Dangerous Function,OutputFun,KO,Call
    to fprintf
9 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    /src/
    CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
    .c,234,main,676,Use of Potentially Dangerous Function,
    InputFun,KO,Call to time
```

```
10 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,50,
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
     ,676,Use of Potentially Dangerous Function,UnknownFun,KO,Call
      to Frama_C_bzero
11 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,72,
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
     ,676,Use of Potentially Dangerous Function,InputFun,KO,Call
     to socket
12 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,72,
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
     ,676,Use of Potentially Dangerous Function,OutputFun,KO,Call
     to socket
13 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,77,
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
     ,676,Use of Potentially Dangerous Function,MemFun,KO,Call to
     memset
14 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,81,
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
     ,676,Use of Potentially Dangerous Function,OpenFun,KO,Call to
      connect
```

```
15 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,88,
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01_bad
     ,676,Use of Potentially Dangerous Function,InputFun,KO,Call
     to recv
16 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/io.c,13,printLine,676,Use of Potentially Dangerous
     Function,FormatFun,KO,Call to printf
17 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/io.c,13,printLine,685,Function Call With Incorrect
     Number of Arguments,Format Match (number of arguments),OK,
     Format '%s\n' match
18 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/io.c,13,printLine,686,Function Call With Incorrect
     Argument Type,Format Match (type of arguments),OK, Format '%s
     \n' match
19 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/io.c,13,printLine,676,Use of Potentially Dangerous
     Function,OutputFun,KO,Call to printf
20 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,234,NA,9999,Unknown message,RTE,KO,Completely invalid
     destination for assigns clause *timer. Ignoring.
21 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,61,NA,9999,Unknown message,RTE,KO,cannot evaluate ACSL
     term; unsupported ACSL construct: logic functions or
     predicates
22 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,94,NA,787,Out-of-bounds Write,RTE,KO,out of bounds write.
     assert \valid(data+(unsigned int)(dataLen+(unsigned int)((
     unsigned int)recvResult/sizeof(char))));
```

```
23 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,96,NA,9999,Unknown message,RTE,KO,cannot evaluate ACSL
     term; unsupported ACSL construct: \base_addr function
24 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,96,NA,9999,Unknown message,RTE,KO,cannot evaluate ACSL
     term; unsupported ACSL construct: logic functions or
     predicates
25 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,99,NA,787,Out-of-bounds Write,RTE,KO,out of bounds write.
     assert \valid(replace);
26 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,101,NA,9999,Unknown message,RTE,KO,cannot evaluate ACSL
     term; unsupported ACSL construct: \base_addr function
27 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,101,NA,9999,Unknown message,RTE,KO,cannot evaluate ACSL
     term; unsupported ACSL construct: logic functions or
     predicates
28 CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     /src/
     CWE134_Uncontrolled_Format_String__char_connect_socket_fprintf_01
     .c,104,NA,787,Out-of-bounds Write,RTE,KO,out of bounds write.
      assert \valid(replace);
```

Appendix C

Demo of Vyper on custom tests

- Tainted related format string vulnerability.

```
1
2 #include<stdio.h>
3 #include<string.h>
4 #define SIZE 10
5
6
7
8 int f(int a){
9  if (a==0) return 1; else return a * f(a-1);
10
11 }
12
13 int wrong_call(char * fmt){
14      printf(fmt); //exploitable vulnerability when input i == 10
15         // The input was correctly reported.
16 }
17
18
19
20
21
22
23
```

```
24 int main( int argc , char** argv){
25   char fmt[SIZE];
26   char fmt2[SIZE];
27   char fmt3 [SIZE];
28   int i;
29         fmt2[0] = 0;
30   fmt3[0] = '0';
31   fmt3[1]  = 0 ;
32
33         fgets(fmt,SIZE-1,stdin);
34   printf ("give i:");
35   scanf("%d",&i);
36   scanf("%s",fmt);
37
38   if (f(i)==3628800)
39     wrong_call(argv[1]);
40   else
41       printf ("good");
42
43
44  return 0;
45 }
```

- Use of concrete argument values.

```
1 int wrong_call(char * fmt){
2      printf(fmt); //Exploitable vulnerability detected
3         // when argv is given the concrete value "good" in the
    input spec file
4         // no vulnerabitlity is detected.
5
6
7 }
8
9
10 int main( int argc , char** argv){
11
12   if (!strcmp(argv[1],"bad"))
```

```
13      wrong_call(argv[1]);
14    else
15        printf ("good");
16
17
18   return 0;
19 }
```

- Stack overflow.

```
1
2  #include<string.h>
3  int i;
4
5  void function(char *str) {
6
7     char buffer[16];
8
9
10   for( i = 0; i < 16; i++)
11      buffer[i] = str[i];
12
13 }
14 void function1(char *str) {
15
16    char buffer[16];
17
18    strcpy(buffer,str);
19  }
20
21 int main(int argc,char** argv) {
22
23
24
25
26   if (strlen(argv[1])<50)
27      {
28      function(argv[1]); // Non-vunerable call
```

```
29      function1(argv[1]); // Exploitable vulnerability
30      }
31    return 0;
32 }
```

- Double free.

```
1  #include<stdio.h>
2  #include<string.h>
3  #include<stdlib.h>
4  #define SIZE 100
5
6
7
8  int handle(char *buf){
9
10   if (buf[0] == 1)
11     free(buf); // Double free vulnerability detected.
12   else
13     return buf[0];
14   }
15 int dumb_func(int a){
16
17 char *fmt;
18 char *fmt2;
19   fmt = malloc(SIZE);
20   fmt2 = malloc(SIZE);
21   memset(fmt,1,SIZE);
22   memcpy(fmt2,fmt,SIZE);
23   free(fmt);
24   handle(fmt);
25   return fmt[SIZE-1];
26
27 }
28
29 int main( int argc , char** argv){
30   char num[10];
31   int i ;
```

```
32   scanf("%d",&i);
33   dumb_func(i);
34 }
```

- Use after free.

```
1 #include<stdio.h>
2 #include<string.h>
3 #include<stdlib.h>
4 #define SIZE 100
5
6
7
8 int handle(char *buf){
9
10   if (buf[0] == 1)
11     buf[0] = 2; // Use after free vulnerability detected.
12   else
13     return buf[0];
14   }
15 int dumb_func(int a){
16
17 char *fmt;
18 char *fmt2;
19   fmt = malloc(SIZE);
20   fmt2 = malloc(SIZE);
21   memset(fmt,1,SIZE);
22   memcpy(fmt2,fmt,SIZE);
23   free(fmt);
24   handle(fmt);
25   return fmt[SIZE-1];
26
27 }
28
29 int main( int argc , char** argv){
30   char num[10];
31   int i ;
32   scanf("%d",&i);
```

```
33   dumb_func(i);
34 }
```

- Heap overflow.

```
1
2 #include<stdio.h>
3 #include<string.h>
4 #define SIZE 10
5
6
7 int dumb_func(char* fmt){
8
9 char *fmt2;
10   fmt2 = malloc(SIZE);
11   memcpy(fmt2,fmt, 2* SIZE); // Exploitabe heap oveflow
       vulnerability
12   return fmt[SIZE-1];
13
14 }
15
16 int main( int argc , char** argv){
17   char num[10];
18   int i ;
19   i = atoi();
20
21   dumb_func(argv[1]);
22 }
```

Bibliography

[1] The cia principle, 2016. URL http://www.doc.ic.ac.uk/~ajs300/security/CIA.htm.

[2] Burcu Bulgurcu, Hasan Cavusoglu, and Izak Benbasat. Information security policy compliance: an empirical study of rationality-based beliefs and information security awareness. *MIS quarterly*, 34(3):523–548, 2010.

[3] Ariane. the ariane catastrophe., . URL http://www.around.com/ariane.html.

[4] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy Magazine*, page 49–51, 2011.

[5] Cnn. toyota recall costs: $2 billion., . URL http://money.cnn.com/2010/02/04/news/companies/toyota_earnings.cnnw/index.htm.

[6] It is 100 times more expensive to fix security bug at production than design., . URL https://www.owasp.org/images/f/f2/Education_Module_Embed_within_SDLC.ppt.

[7] Chris F Kemerer. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering*, 1(1):1–22, 1995.

[8] Polyspace static analysis, 2016. URL https://fr.mathworks.com/products/polyspace/.

[9] Jean-Louis Boulanger. Polyspace. In *Static Analysis of Software*, chapter 3, pages 113–142. John Wiley & Sons, Inc, 2013. ISBN 9781118602867.

doi: 10.1002/9781118602867.ch3. URL http://dx.doi.org/10.1002/9781118602867.ch3.

[10] Frama-c, 2015. URL https://frama-c.com/.

[11] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015. ISSN 1433-299X. doi: 10.1007/s00165-014-0326-7. URL http://dx.doi.org/10.1007/s00165-014-0326-7.

[12] The astree static analyzer, 2016. URL http://www.astree.ens.fr/.

[13] Hp fortify static code analyzer, 2016. URL http://www8.hp.com/us/en/software-solutions/static-code-analysis-sast/.

[14] Coverity static analyzers, 2016. URL https://www.coverity.com/.

[15] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065036. URL http://doi.acm.org/10.1145/1064978.1065036.

[16] Juliet test suites, 2016. URL https://samate.nist.gov/SRD/testsuite.php.

[17] 2016 cost of data breach study: Global analysis, 2016. URL https://www.cloudmask.com/hubfs/IBMstudy.pdf?t=1511525326416.

[18] Henry Jiang (CISSP). The world of cybersecurity map version 2.0, 2017. URL https://www.linkedin.com/pulse/map-cybersecurity-domains-version-20-henry-jiang-ciso-cissp/.

[19] Top 125 network security tools, 2017. URL http://sectools.org/tag/vuln-scanners/.

[20] Common vulnerabilities and exposures cve, 2015. URL http://cve.mitre.org/.

[21] Common weakness enumeration cwe, 2015. URL https://cwe.mitre.org/.

[22] Mario Heiderich, Gareth Heyes, and Abraham Aranguren-Aznarez. Systems and methods for client-side vulnerability scanning and detection, June 10 2014. US Patent 8,752,183.

[23] S Balaji and M Sundararajan Murugaiyan. Waterfall vs. v-model vs. agile: A comparative study on sdlc. *International Journal of Information Technology and Business Management*, 2(1):26–30, 2012.

[24] Definition of a security vulnerability, 2017. URL https://msdn.microsoft.com/en-us/library/cc751383.aspx.

[25] Peter Mell, Karen Scarfone, and Sasha Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6), 2006.

[26] Openssl 'heartbleed' vulnerability (cve-2014-0160), 2017. URL https://www.us-cert.gov/ncas/alerts/TA14-098A.

[27] Craig Timberg. Net of insecurity a flaw in the design, 2015. URL http://www.washingtonpost.com/sf/business/2015/05/30/net-of-insecurity-part-1/.

[28] John Viega Michael Howard, David LeBlanc. *19 Deadly Sins of Software Security: Programming Flaws and How to Fix Them (Security One-off)*. McGraw-Hill Osborne Media.

[29] Charles Arthur. Apple's ssl iphone vulnerability: how did it happen, and what next?, 2014. URL https://www.theguardian.com/technology/2014/feb/25/apples-ssl-iphone-vulnerability-how-did-it-happen-and-what-next

[30] Gary McGraw. Software security. *IEEE Security & Privacy*, 2(2):80–83, 2004.

[31] Marisa Reddy Randazzo, Michelle Keeney, Eileen Kowalski, Dawn M Cappelli, and Andrew P Moore. Insider threat study: Illicit cyber activity in the banking and finance sector. 2005.

[32] Cert coding standards, 2016. URL https://www.securecoding.cert.org.

[33] Misra c, 2017. URL https://misra.org.uk/.

[34] Static code analysis, 2015. URL https://www.owasp.org/index.php/Static_Code_Analysis.

[35] Teun Koetsier. *On the prehistory of programmable machines; musical automata, looms, calculators.* PERGAMON, 2001.

[36] James Ettinger. *Jacquards Web.* Oxford University Press, 2004.

[37] Dennis M. Ritchie. The development of the c language. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 201–208, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. doi: 10.1145/154766.155580. URL http://doi.acm.org/10.1145/154766.155580.

[38] Tiobe index for january 2018, 2016. URL https://www.tiobe.com/tiobe-index/.

[39] ISO. Iso/iec 9899:tc2 (c99 standard). 2005. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[40] List of compilers, 2017. URL https://www.codechef.com/wiki/list-compilers.

[41] IEEE. The open group base specifications issue 6 ieee std 1003.1, 2004 edition, 2004. URL http://pubs.opengroup.org/onlinepubs/009695399/.

[42] The GNU C library (glibc), 2015. URL http://www.gnu.org/software/libc/.

[43] Linux standard base core specification 4.1, 2010. URL https://refspecs.linuxfoundation.org/LSB_4.1.0/LSB-Core-generic/LSB-Core-generic.txt.

[44] Standard C++ library reference, 2015. URL https://msdn.microsoft.com/en-us/library/aa248416%28v=vs.60%29.aspx.

[45] ISO. ISO/IEC 9899:tc2, appendix J, paragraph 2. 2005. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf.

[46] Cwe-134: Uncontrolled format string. 2015. URL https://cwe.mitre.org/data/definitions/134.html.

[47] fprintf, printf, snprintf, sprintf - print formatted output, 2015. URL http://pubs.opengroup.org/onlinepubs/009695399/functions/fprintf.html.

[48] closelog, openlog, setlogmask, syslog - control system log, 2015. URL http://pubs.opengroup.org/onlinepubs/009695399/functions/syslog.html.

[49] Linux manual help page, 2015. URL http://linux.die.net/man/3/.

[50] Format specification syntax: printf and wprintf functions, 2015. URL https://msdn.microsoft.com/en-us/library/56e442dc.aspx.

[51] CWE-78: Improper neutralization of special elements used in an OS command ('OS command injection'), 2014. URL https://cwe.mitre.org/data/definitions/78.html.

[52] BA Wichmann, AA Canning, DL Clutterbuck, LA Winsborrow, NJ Ward, and DWR Marsh. Industrial perspective on static analysis. *Software Engineering Journal*, 10(2):69–75, 1995.

[53] Owasp - open web application security project. 2017. URL https://www.owasp.org/.

[54] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[55] C intermediate language, 2017. URL https://github.com/cil-project/cil.

[56] Gcc front ends, 2016. URL https://gcc.gnu.org/frontends.html.

[57] Edison design group, 2016. URL http://www.edg.com.

[58] Dms® software reengineering toolkit™, 2016. URL http://www.semanticdesigns.com/Products/DMS/DMSToolkit.html.

[59] Gnu general public license, 2016. URL https://www.gnu.org/copyleft/gpl.html.

[60] Apache software foundation, 2016. URL https://www.apache.org/licenses/.

[61] Gnu lgpl, 2016. URL https://opensource.org/licenses/lgpl-license.

[62] The llvm compiler infrastructure, 2016. URL http://llvm.org.

[63] The hiphop virtual machine, 2016. URL https://www.facebook.com/note.php?note_id=10150415177928920.

[64] a python to c++ converter, 2017. URL https://github.com/pradyun/Py2C.

[65] Toba:a java-to-c translator, 2017. URL https://www2.cs.arizona.edu/projects/sumatra/toba/.

[66] Gnu bison, 2017. URL https://www.gnu.org/software/bison/.

[67] The bnf converter, 2016. URL https://github.com/BNFC/bnfc.

[68] angr framework, 2016. URL https://github.com/angr/.

[69] Flawfinder, 2017. URL https://www.dwheeler.com/flawfinder/.

[70] How to use lint for static code analysis, 2017. URL https://barrgroup.com/Embedded-Systems/How-To/Lint-Static-Analysis-Tool.

[71] Cppcheck, 2017. URL http://cppcheck.sourceforge.net/.

[72] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[73] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[74] Runtime error definition, 2012. URL http://techterms.com/definition/runtime_error.

[75] Daniel Kroening and Michael Tautschnig. Cbmc–c bounded model checker. In *TACAS*, pages 389–391, 2014.

[76] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855756.

[77] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Commun. ACM*, 57(2):74–84, February 2014. ISSN 0001-0782. doi: 10.1145/2560217.2560219. URL http://doi.acm.org/10.1145/2560217.2560219.

[78] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[79] Common vulnerability scoring system, 2016. URL https://www.first.org/cvss/.

[80] Source code security analyzers, 2016. URL https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html.

[81] Gregory Larsen, EK Fong, David A Wheeler, and Rama S Moorthy. State-of-the-art resources (soar) for software vulnerability detection, test, and evaluation. Technical report, INSTITUTE FOR DEFENSE ANALYSES ALEXANDRIA VA, 2014.

[82] Alexander M. Hoole, Issa Traore, Aurelien Delaitre, and Charles de Oliveira. Improving vulnerability detection measurement: [test suites and software security assurance]. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, EASE '16, pages 27:1–27:10, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3691-8. doi: 10.1145/2915970.2915994. URL http://doi.acm.org/10.1145/2915970.2915994.

[83] Gabriel Díaz and Juan Ramón Bermejo. Static analysis of source code security: Assessment of tools against samate tests. *Information and software technology*, 55(8):1462–1476, 2013.

[84] Combinatorial explosion, . URL http://pespmc1.vub.ac.be/ASC/COMBIN_EXPLO.html.

[85] Arnaud J Venet and Michael R Lowry. Static analysis for software assurance: soundness, scalability and adaptiveness. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 393–396. ACM, 2010.

[86] El Habib Boudjema, Christèle Faure, Mathieu Sassolas, and Lynda Mokdad. Detection of security vulnerabilities in c language applications. *Security and Privacy*, pages e8–n/a. ISSN 2475-6725. doi: 10.1002/spy2.8. URL http://dx.doi.org/10.1002/spy2.8. e8.

[87] Guide de règles et de recommandations relatives au développement d'applications de sécurité en java, 2009. URL http://www.ssi.gouv.fr/uploads/IMG/pdf/JavaSec-Recommandations.pdf.

[88] Java SE, 2016. URL https://docs.oracle.com/javase/specs/.

[89] Éde de la sécurité intrinsèque des langages fonctionnels, 2011. URL http://www.ssi.gouv.fr/uploads/IMG/pdf/LaFoSec_-_Analyse_des_langages_OCaml_F_et_Scala.pdf.

[90] OCaml, 2016. URL https://ocaml.org/docs/.

[91] Coverity scan - static analysis, 2017. URL https://scan.coverity.com/.

[92] A brief history of c, 2016. URL https://www.le.ac.uk/users/rjm1/cotter/page_06.htm.

[93] Borland turbo c, 2016. URL https://en.wikipedia.org/wiki/Borland_Turbo_C.

[94] Internet security glossary, 2016. URL https://tools.ietf.org/html/rfc2828.

[95] Denial of service attack, 2016. URL https://en.wikipedia.org/wiki/Denial-of-service_attack.

[96] scut/Team TESO. Exploiting format string vulnerabilities. 2001.

[97] Saferiver , carto-c, 2017. URL http://saferiver.fr/index.php?id=carto-c.

[98] Static analysis tool exposition (sate) iv, 2016. URL https://samate.nist.gov/SATE4.html.

[99] Cas static analysis tool study - methodology, 2011. URL https://samate.nist.gov/docs/CAS_2011_SA_Tool_Method.pdf.

[100] Static analysis tool exposition, 2016. URL https://samate.nist.gov/SATE.html.

[101] Ari Takanen, Jared D. Demott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2008.

[102] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 380–394, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.31. URL http://dx.doi.org/10.1109/SP.2012.31.

[103] PaX Team. Pax address space layout randomization (aslr). 2003.

[104] Starr Andersen and Vincent Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[105] The world's most used penetration testing framework. URL https://www.metasploit.com/.

[106] Offensive security's exploit database archive, . URL https://www.exploit-db.com/.

[107] Our most advanced penetration testing distribution, ever. URL https://www.kali.org/.

[108] Smashing the stack for fun and profit, 2016. URL http://insecure.org/stf/smashstack.html.

[109] Exploit writing tutorial part 11 : Heap spraying demystified, 2016. URL https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/.

[110] Stephen A. Cook and Robert A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7(4):354–375, 1973. doi: 10.1016/S0022-0000(73)80029-7. URL https://doi.org/10.1016/S0022-0000(73)80029-7.

[111] Ansi/iso c specification language, 2016. URL https://frama-c.com/acsl.html.

[112] Problems while using inpsector breakpoints #424. URL https://github.com/angr/angr/issues/424.

[113] Clifton A Ericson. Functional hazard analysis. *Hazard Analysis Techniques for System Safety*, pages 271–289, 2005.

[114] Insider threat report. URL http://crowdresearchpartners.com/portfolio/insider-threat-report/.

[115] Chris Wysopal, Chris Eng, and Tyler Shields. Static detection of application backdoors. *Black Hat*, 2007.

[116] A repository of live malwares for your own joy and pleasure. URL https://github.com/ytisf/theZoo.

[117] Cwe-416: Use after free, 2016. URL https://cwe.mitre.org/data/definitions/416.html.

[118] Cwe-415: Double free, 2016. URL https://cwe.mitre.org/data/definitions/415.html.

[119] Saferiver, 2016. URL http://safe-river.com.