Detecting Variadic Functions in Stripped Binaries from C/C++

A Dissertation
Presented in Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy
with a
Major in Computer Science
in the
College of Graduate Studies
University of Idaho
by
Zeinab Ghafari

Major Professor: Jim Alves-Foss, Ph.D.
Committee Members:
Daniel Conte de Leon, Ph.D.;
Clint Jeffery, Ph.D.;
Jia Song, Ph.D.
Department Administrator: Terence Soule, Ph.D.

December 2020

Authorization to Submit Dissertation

This dissertation of Zeinab Ghafari, submitted for the degree of Doctor of Philosophy with a Major in Computer Science and titled "Detecting Variadic Functions in Stripped Binaries from C/C++," has been reviewed in final form. Permission, as indicated by the signatures and dates below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor:	Jim Alves-Foss, Ph.D.	Date
Committee Members:	Daniel Conte de Leon, Ph.D.	Date
	Clint Jeffery, Ph.D.	Date
	Jia Song, Ph.D.	Date
Department Administrator:	Terence Soule, Ph.D.	Date

Abstract

C and C++ are the most popular programming languages used to implement browsers, runtime libraries, internet of things devices and operating system kernels. Due to the important nature of these devices and their software, it is important to identify security vulnerabilities in the software before adversaries find them. One of the common low-level vulnerabilities in C/C++ programming languages involves the misuse of variadic functions. Variadic functions take a variable number of arguments and pass them to other functions. Misusing variadic functions can lead to memory safety violations, mismatching of function arguments or can enable execution of remote code. The most common attack vectors involve providing input that forces a function in the program to assume it has received more arguments than were actually passed. This allows the attacker to read and possibly write values on the control stack, and in effect dynamically patch the code while it is running.

The goal of this research is to develop a theory and proof of concept tool for the automated detection of variadic functions in stripped binaries and the actual numbers of arguments passed to that function. This technology will enable future automated patching of vulnerable variadic functions. We implemented an automated tool, called Detector for identifying variadic functions to assist software developers and security analysts in pinpointing and repairing vulnerable code. The approach presented in this dissertation focuses on analyzing stripped binaries, which are those with all debug and symbol data removed. These binaries represent the difficulty found in fixing security vulnerabilities in legacy code and third-party libraries, although they can also be used to represent newly developed software. The target binaries were compiled by three different compilers: GCC, Clang, and ICC in both popular Intel x86 and x64 architectures.

Our major contribution in this research is using syntactic and semantic analysis to detect the variadic functions based on their behavior in the stripped binary code. Our experimental results indicate that the Detector is more accurate than other

existing tools. The average of precision and recall for GCC x64 and x86 are more than 99%, Clang X64 is around 98%, and ICC X64 is around 94%.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Jim Alves-Foss, for his support, encouragement, and patient guidance throughout my graduate studies. The journey of my Ph.D. would not be possible without him. His critical thinking and insights always inspired me, especially during the time I faced difficulties and setbacks. I greatly appreciate his efforts and patience in my research.

Second, I would like to thank my committee members, Dr. Daniel Conte de Leon, Dr. Clinton Jeffery, and Dr. Jia Song, for their valuable comments on my dissertation. I greatly appreciate their time and support in this research. The University of Idaho is excellent because of professors like you.

I would like to thank all my instructors Dr. Jim Alves-Foss, Dr. Daniel Conte de Leon, Dr. Hassan Jamil, and Dr. Jia Song for providing me with knowledge about Computer Science and building my research skills.

I would like to thank the department chair, Dr. Terry Soule, and Ms. Arvilla Daffin, Ms. Sue Branting, Ms. Arleen Furedy, and other staff members in the Department of Computer Science for their help during my study in the department.

I was fortunate to work as a teaching assistant during my Ph.D. progress. It gave me opportunities to work on interesting problems and improve my technical skills. I would like to thank Dr. Terry Soule, Dr. Bruce Bolden, and Dr. Michael Wilder. I would like to thank all the staff of the College of Graduate Studies for their help and support throughout my study at UI.

I would like to thank my labmates for providing encouragement, entertainment, and sometimes just an attentive ear to whatever problem I was facing. Thank you Jocelyn Stadler, Shannon Hurley, Ananth Jillepalli, Baoying Lou, and Jonathan Buch.

I would like to thank many other friends who made my life at UI very enjoyable. Thank you Harika Vadapalli, Goz Olivier, Homaja Kumari, Seema Kamod, Madhumitha Reddy, Amruta Kale, Joel Oduro-Afriyie, Mohammad Baqer, Abhinav Prabhu, Oshan Singh Karki, Matt Yang, Amal Aljohani, and Nuzhat Yamin.

I would like to thank my parents Hassan Ghafari and Mahnaz Sharabiani, for all your encouragement and prayer. I would also like to thank my sister, Monica Ghafari. You inspire me so much! You constantly support me to pursue my goals and encourage me during tough times.

The real person that earned this dissertation is my husband, Ehsan Mohandesi. I would like to thank him for understanding, support, encouragement, and love, all of which have helped me make this dissertation a reality. He has been there through the tears, tantrums, and thwarted attempts to give up. Thanks for believing in me and always having my back.

Dedication

I am deeply thankful to my parents Hassan Ghafari and Mahnaz Sharabiani for bringing me into this world and supporting me throughout my pursuit of higher education. I would like to express my heartfelt thanks to Monica, my beloved sister, for her selfless love, sacrifice, and encouragement. Finally and the most importantly to my husband Ehsan Mohandesi who provided me love.

Table of Contents

List of Figures

Listings

List of Tables

CHAPTER 1

Introduction

C and C++ are the most popular programming languages used to write system software. Although programmers use these programming languages for a wide variety of usages, there are some common vulnerabilities within them. Some of the low-level vulnerabilities in C and C++ programming languages are memory safety violations [1], uninitialized variables [2], misusing arguments in variadic functions [3], control-flow hijacking [4], privilege escalation [5], and information leakage [6]. Attacks against these vulnerabilities can lead to security concerns and these attacks can change the program's behavior. Therefore, analyzing these programming languages plays a vital role in security. These kinds of security vulnerabilities can be tracked in both source code and binary code. The applications of source code and binary code are different. Indeed, source code demonstrates function declarations, types of the arguments, and return values while binary code does not show such information because it is in the format of byte streams. Therefore, the importance of tracking vulnerabilities in binary codes requires meticulous attention. In this way, many binary analysis tools have been designed using static, dynamic, or even hybrid approaches to detect low-level vulnerabilities in binary codes.

The goal of this research is to develop a theory and proof of concept tool for the automated detection of variadic functions in stripped binaries and the actual numbers of arguments passed to that function. This technology will enable future automated patching of vulnerable variadic functions. The specific aim of this dissertation is to design a theory and proof of concept tool, called Detector, to automatically detect variadic functions in stripped binary code of C and C++ programs. There are two kinds of binaries, non-stripped binary code which contains debugging information and stripped binary code which generally removes the debugging information from the executable file since this information is not necessary for execution. Variadic functions are flexible in C and C++ because of the absence of the type and number of arguments. They allow the caller to pass an unbounded number of

arguments. The contract between the caller and callee for passing the arguments is implicit so misusing the variadic function is an attractive case for attackers. In unsafe code, attackers can force arguments to have unexpected values or can change the number and the type of arguments, affecting the behavior of the function. For this purpose, we set out to develop a process for detecting variadic function and developed a proof of concept tool, Detector, that can automatically recognize variadic functions in a stripped binary code. The Detector is able to detect the variadic functions in binaries generated by three different compilers: GCC, Clang, and ICC, in both Intel x86 and x64 architectures with different levels of optimization. Detector uses a syntactic analysis to statically identify the variadic function patterns. Each compiler and architecture has its own patterns. First, Detector predicts the type of compiler and architecture based on the instructions found in the stripped binary code. Then, based on the type of compiler and architecture, Detector tracks the variadic function patterns. These patterns include the status of registers which take the variadic arguments and how arguments are passed and located on the stack. Finally, Detector extracts the information of the call site to show which functions are called by the variadic function and also, which functions are calling variadic functions in stripped binary code.

1.1 VARIADIC FUNCTIONS

The role of variadic functions in C and C++ programming languages is to allow calling functions to pass a non-fixed number of arguments to the functions unlike non-variadic functions that take a fixed number of arguments. Programmers use variadic functions to eliminate duplication in code, reduce code size, and increase compilation speeds [1]. To make it clear, the variadic function is called with a variable number of arguments. The variadic functions can be recognized in the source code by the ellipsis meta-operation ("..."). The ellipsis is used as a place holder for variadic arguments, which can accept any number of arguments for that parameter. Take the source code shown in Listing 1.1 as an example of a variadic function. The average() function takes a variable number of arguments

LISTING 1.1: Average function is an example of the variadic function

```
1 #include <cstdarg>
2 #include <iostream>
3 using namespace std;
4 int average (int num, ...){
5   va_list arguments;
6   int sum = 0;
7   va_start (arguments, num);
8   for (int x = 0; x < num; x++)
9     sum += va_arg (arguments, int);
10  va_end (arguments);
11  return sum / num;
12 }
13 int main (){
14  cout<< average (3, 12, 22, 4) <<endl;
15  cout<< average (2, 30, 4) <<endl;
16  }
```

and calculates the average of all of them. On line 4, the arguments of the average() function are num and ellipsis. "Num" is non-variadic argument and ellipsis is a variadic argument. In this example, this num is used to specify the number of variadic arguments. From the main() function, the average() function is called two times on lines 14 and 15. First, with (3, 12, 22, 4), the num is 3. It is the non-variadic argument and specifies the number of variadic arguments. Variadic arguments in the first call are 12, 22, and 4. In the second call, the num is 2. It is the non-variadic argument and the variadic arguments are 30 and 4.

Most variadic functions have been defined by using macros defined in one of two different files stdarg.h and varargs.h, that were written according to ISO C99 standard and ANSI C89 standard, respectively. In listing 1.1, four main macros are used to define the average function as a variadic function. These macros are va_list, va_start, va_arg, and va_end. In the following, the tasks of these macros are defined.

- va_list declares a local variable used to refer to the list of variable parameters.
- va_start() is passed a valid va_list and the name of the last fixed parameter variable before the ellipsis ("...") in the source code. A common practice is to

use this last fixed parameter as an integer to describe the number of arguments that are being passed into the variable parameters it can also be a format string instead of an integer. The va_list variable is used to reference the list of variadic parameters.

- va_arg() is used to parse the next argument from va_list as a specified type and return that value. The return value of va_arg() is the value of the next argument; va_arg() can be called repeatedly to step through the variable number of arguments specified by va_list.

- va_end() is used to clean up the va_list variable.

1.2 VULNERABLE VARIADIC FUNCTIONS

The main concern of this dissertation is that the C and C++ compilers cannot statically check the number and the type of the arguments in the implicit contract between the callee and the caller, which leads to vulnerabilities in the variadic function. This section describes the state of the art by addressing some of the most common problems in using variadic functions where such vulnerabilities have been mentioned.

Memory safety violations can lead to erroneous execution, crashes, and reboot since the attacker can successfully exploit a dangling pointer vulnerability to execute arbitrary malicious code and even bypass address space layout randomization [7]. Misusing the variadic functions in this type of vulnerability can lead to Format String Attack. Format String Attack can occur when the submitted data of an input string is evaluated as a command by the application so the attacker can execute code, read the stack, or cause a segmentation fault [8].

Format String Attack [9] exploits functions such as printf() which can accept a variable number of input arguments. The first argument of printf() is a format string which is used to define the types and numbers of arguments. If the attacker can change the contents of this argument, then it is called a format string attack. C/C++ programming languages use many variadic functions including

printf, fprintf, sprintf, syslog, etc. Memory safety vulnerabilities can happen in these functions. Among efforts to mitigate format string vulnerabilities are tools that restrict the use of the %n qualifier in the format string [10].

In addition to format string vulnerabilities there could be a mismatching number of arguments because of the flexibility of the variadic functions. There might be a mismatch between the type and number of arguments that have been transferred between the callee and the caller,since the contract between the caller and the callee is implicit. Therefore, the attacker has the capability to break the contract between the callee and the caller and then change the number of arguments or use the wrong argument type and change the behavior of the program. The attacker can also hijack an indirect call of the variadic function and violate the implicit contract between caller and callee. Control Flow Integrity or CFI countermeasures specifically prevent illegal calls over indirect call edges. However, even the most precise implementations of CFI, which verify the type of the targets of indirect calls, are unable to fully stop illegal calls to variadic functions [11]. The attacker also has the capability to read or write the memory contents.

If the caller of a variadic function could explicitly specify the number of input arguments it prepares as another input argument, it would make a format string attack much more difficult, if not impossible [12].

Effects of Vulnerable Variadic Functions: The mentioned vulnerabilities can give the attacker the capability to read sensitive data or overwrite the memory contents. The attacker can read data from internal memory locations, overwrite function pointers or change return addresses.

There are three main vulnerabilities related to variadic functions that have been defined by the Common Weakness Enumeration (CWE™) [13] project.

- CWE-628 Function Call with Incorrectly Specified Arguments: CWE-628 includes the wrong variable or reference, an incorrect number of arguments, incorrect order of arguments, or wrong type of arguments.
- CWE-686 Function Call with Incorrect Argument Type: The types of variable arguments cannot be enforced at compilation time.

- CWE-1056 Invokable Control Element with Variadic Parameters: If the relevant code is reachable by an attacker then it may not prevent an attack.

1.3 RESEARCH MOTIVATION AND OBJECTIVES

According to Song et al. [3] "The variadic function's source code does not specify the number or types of these variadic arguments. Instead, the fixed arguments and the function semantics encode the expected number and types of variadic arguments. Variadic arguments can be accessed and simultaneously typecast using va_arg. It is impossible to statically verify that va_arg accesses a valid argument, or that it casts the argument to a valid type. This lack of static verification can lead to type errors, spatial memory safety violations, and uses of uninitialized values."

Although Song et al. [3] state that it is "impossible to statically verify that va_arg accesses a valid argument", we claim that under many cases it is possible and that their claim is in the general case. More importantly, we propose to develop techniques that can clearly determine if a variadic function is susceptible to argument overrun by not accurately checking the limits. We also propose that we could implement techniques to modify variadic functions to define an additional argument insert logic in the function that uses this argument for bounds checking that represents the actual numbers of parameters.

1.4 CHALLENGES FOR DETECTING VARIADIC FUNCTIONS IN BINARY CODE

Detecting variadic functions in binary code is essential and critical for many security reasons. The Detector should overcome the challenges that are described below.

- The Detector does not have access to abstractions and libraries which are used to describe the high-level information of programming languages. To make it more clear, this means the Detector needs to recognize the variadic function abstractions based on behavioral analysis. The Detector just has access to the

stripped binary code and function boundaries. There are no symbol tables and debug information.

- The variadic functions can be called both directly and indirectly. The Detector can check both direct and indirect calls statically. If functions in the program use the variadic arguments of vulnerable variadic function then it will affect the whole program. The Detector shows which functions are called by the variadic function and also, which functions are calling variadic functions in stripped binary code.

- The compilers GCC, Clang, and ICC for both Intel x64 and x86 have their instructions for taking and locating the variadic arguments. Applying data flow analysis is very complex because of the lack of high-level information. The variadic arguments can be located in different locations such as stack and registers. However, the Detector overcomes this challenge by using behavior analysis and following the patterns statically in the binary code.

- Compilers GCC, Clang, and ICC and different optimization levels generate the stripped binary code in different ways. The programmers use optimization levels to reduce the code size and improve performance. Some of the functions use jumps instead of calls in high-level optimization. Therefore, the Detector should recognize the different jumps which enter variadic functions. The variadic functions can be called from multiple locations in the whole program. The Detector overcomes this challenge by capturing all callsite information for each variadic function in the program.

- The Detector identifies variadic functions which execute in the program. Tools that conduct source code analysis find that some variadic functions are just defined in the headers or inline libraries and these variadic functions do not execute in the program. Detector identifies the executed variadic functions in the program.

1.5 SUMMARY OF CONTRIBUTIONS

In this dissertation, we design a novel process to identify the variadic functions in the stripped binary code to assist in the improvement of software security. We also test our process using a proof of concept tool, Detector. The base implementation of Detector is static analysis.

Detector applies deep static analysis to recognize the functionality of variadic functions in all possible paths without executing the program. We precisely use static pattern matching of variadic functions for each compiler and architecture to follow the functionality of variadic functions in the program.

The main goals of this research are summarized as follows:

- Detector predicts the type of the compiler and the architecture by using syntactic analysis. According to the type of compiler and architecture, Detector chooses the pattern of variadic function.
- Detector detects variadic functions automatically in the binary level of Intel x86 and x64 architectures.
- Detector detects variadic functions automatically in the binary code compiled by GCC, Clang, and ICC using optimization levels (-O0 to -O3).
- Detector analyzes the behavior of the function by syntactic analysis to recognize if the behavior of the function matches a variadic function pattern.
- Detector captures the call information of the caller and callee to recognize which functions can be affected by variadic functions and their arguments in the program.
- Detector has more precision and accuracy compared to other existing tools. We support this statement in the experimental result.
- Unlike previous approaches for detecting variadic functions, the only requirements of the Detector are stripped binary code and function boundaries. Other existing tools need more requirements such as accessing the source code, considering symbol tables, and compiling information.

1.6 DISSERTATION OUTLINE

The rest of the dissertation is organized as follows: Chapter 2 covers the relevant background information. Chapter 3 describes applied techniques for designing the theory to identify variadic functions in the stripped binary code automatically. Chapter 4 presents the implementation of Detector for the different compilers GCC, Clang, and ICC. Chapter 5 demonstrates the experimental results of the detected number of variadic functions by comparing it with other available tools. And lastly, Chapter 6 concludes.

CHAPTER 2

Background and Related Works

This chapter discusses the background and related work for this dissertation. Section 2.1 describes binary analysis, techniques of binary analysis, variadic functions in binary code, and passing the arguments from variadic function through other functions. Section 2.2 discusses the related works that includes the following sections.

Section 2.2.1 compares the exiting tools of binary analysis. Section 2.2.2 focuses on the other existing techniques for mitigating the vulnerabilities in source code and binary code of variadic functions. Sections 2.2.3 - 2.2.5 introduce the existing tools related to detecting functions in binary code.

2.1 BACKGROUND

2.1.1 *Binary Analysis*

Binary analysis is one of the ways for evaluating a program when source code is not available. Binary analysis is a difficult task in the absence of function types, boundaries, layout, and all the extra information found in source code.

2.1.2 *Techniques of Binary Analysis*

There are several different types of binary analysis methods. One way to categorize them is based on whether they require execution of the target program.

Static Analysis: Static analysis evaluates the software without execution. This evaluation can include source code, or just the binary executable. Static analysis techniques can analyze all possible control flows of a program, achieving a significant higher coverage of program vulnerabilities and, as a result, produce a significantly lower false negative rate compared to dynamic analysis approaches.

Dynamic Analysis: Dynamic analysis, such as fuzzing, symbolic execution, and taint analysis, requires execution of the target program during analysis. Symbolic execution is an analysis technique that executes programs with symbolic rather than concrete inputs and maintains a path condition that is updated whenever a branch instruction is executed [14]. A symbolic execution engine replaces the inputs with symbolic variables, that are initially set to be anything, and then runs the programs. This way constraints are encoded on the inputs that reach that program point. Based on the idea of modern application's tendency to crash due to random input, Fuzzing was proposed by Miller et al. [15] in 1990. They developed the first Fuzzing tool "Fuzz", which generates streams of random characters. Since then, Fuzz has been proved to be an effective method to find software vulnerabilities. Fuzz was tested on ninety different utility programs on seven versions of UNIX and was able to crash more than 24% of them. Disadvantages of this approach are state-explosion, high cost and low coverage, however it gets a low false positive rate.

Hybrid Analysis: A hybrid approach attempts to combine the benefits of static and dynamic analysis techniques while mitigating their disadvantages. A hybrid approach might use static analysis to reduce state-space explosions by ruling out certain paths, while running dynamic analysis on portions of the code that can not be modeled in a static analysis.

Comparing Binary Analysis Techniques: Static analysis techniques, such as data flow analysis can be utilized to detect vulnerabilities in source code without code execution. Although it is a fast and scalable technique for scanning millions of lines of source code, it has high rate of false positives. On the other side, dynamic analysis, such as fuzzing and symbolic execution, needs the code to be run during analysis, and evaluates limited inputs or runs slowly.

2.1.3 *Functions in Binary code*

Most of the constructs in a program consist of functions. Detecting the functions in the source code is not hard because function declarations, definitions, and layouts

are available. In contrast, identifying functions in binary code is not easy since the only available information is byte streams. The first step of binary analysis is disassembly which translates the code into decoded instructions [16]. The executable file does not need high-level information such as function names, argument types, and return values. Therefore, an essential task is detecting functions in binary code without accessing the high-level information. Reverse engineering [17] is one of the methods that helps to recognize the functions from binary code. Reverse engineering analyzes the compiled code to recognize the behavior of the program. Some of the reverse engineering techniques are disassembling, control flow analysis, function abstraction, call graph construction, software architecture recovery [16]. Fortunately, a companion project that is part of the University of Idaho JIMA tool suite can do this very well [18]. In this dissertation, static analysis and pattern matching have been applied to recognize the variadic functions in binary code, given function boundaries. More precisely, we recognize the patterns of variadic functions syntactically in binary code. These patterns determine the functionality of variadic functions in code.

2.1.4 *Variadic Functions versus Non-variadic Functions in Binary Code*

As mentioned before, variadic functions take a variable number of arguments but non-variadic functions take a fixed number of arguments and pass them to the other functions. In this section, the differences between the variadic functions and non-variadic functions in assembly code are discussed.

Listing 2.1 and Listing 2.2 show the source code of the average() function in two formats as variadic and non-variadic functions. Both variadic and non-variadic functions have been compiled by GCC with x86 architecture.

Listing 2.3 and Listing 2.4 compare the caller function main(), in both variadic and non-variadic functions in assembly code. The compiled code demonstrates that a compiler uses the same structure for calling both variadic and non-variadic functions. Both of them show pushing the arguments on the stack in the same way. (Lines 8-12 in Listing 2.3 and 7-11 in Listing 2.4)

Figure 2.1 compares the callee side of the binary code in both variadic and non-variadic functions. As the compiled code illustrates, the main difference between

LISTING 2.1: Average function in variadic function format

```
#include <cstdarg>
#include <iostream>

using namespace std;

int average ( int num, ... )
{
  va_list arguments;
  int sum = 0;
  va_start ( arguments, num );
  for ( int x = 0; x < num; x++ )
  sum += va_arg ( arguments, int );
  va_end ( arguments );
  return sum / num;
}
int main()
{
  cout<< average ( 3, 12, 22, 4 ) <<endl;
}
```

non-variadic and variadic functions is related to how these functions read their arguments. The yellow is the function entry preamble, the grey highlighted command shows how the variadic function defines the va_start and va_arg macros in binary code, and the green is for function exit.

2.1.5 *Calling Convention for Variadic Functions*

Calling conventions specify how arguments are passed to a function, how return values are passed back out of a function, how the function is called, and how the function manages the stack and its stack frame [19]. As mentioned in Section 2.1.4, the compiler has to use the same calling convention for both non-variadic and variadic function calls. During a variadic function call, only the caller knows the exact number and types of arguments. Therefore, the callee cannot be responsible for deallocating the arguments when it returns to the caller, which has to be the caller's task. Since there could be no difference between non-variadic function and variadic function calls, it follows that in all calls; it is the caller that deallocates the arguments after the callee has returned.

The typical calling convention is the following:

LISTING 2.2: Average function in non-variadic function format

```
1  #include <iostream>
2
3  using namespace std;
4
5  int average ( int num, int a1, int a2, int a3 )
6  {
7    int sum = 0;
8    sum += a1;       sum += a2;       sum += a3;
9    return sum / num;
10 }
11 int main()
12 {
13   cout<< average ( 3, 12, 22, 4 ) <<endl;
14 }
```

1. The caller evaluates the arguments and places them in sequence starting at a known location.

2. The caller saves its return address and branches to the callee.

3. The callee can reference a prefix of the arguments since it knows the location of the first one.

4. The callee returns to the caller by branching to the return address.

5. The caller deallocates the arguments.

On a stack-based implementation, step 1 typically pushes the arguments in reverse order on the stack, and step 5 adjusts the stack pointer by a constant amount. An implementation that allows passing of arguments in registers, stores the first several arguments in registers before pushing any additional arguments on the stack.

2.1.6 *Abstract Stack Analysis*

This section describes how the macros in the variadic function walk through the call stack [4] and call each other to execute the variadic function. Figure 2.2 demonstrates the functionality of the important macros va_start and va_arg for the list of arguments A, B, and C assumed as variadic arguments. va_start initializes the variadic function by pointing to the first argument from the list of arguments. Then,

Listing 2.3: Caller of variadic function in binary code

```
17  {
    0x0804863e <+0>: push   %ebp
    0x0804863f <+1>: mov    %esp,%ebp
    0x08048641 <+3>: and    $0xfffffff0,%esp
    0x08048644 <+6>: sub    $0x20,%esp

18      cout<< average ( 3, 12, 22, 4 ) <<endl;
    0x08048647 <+9>: movl   $0x4,0xc(%esp)
    0x0804864f <+17>:  movl   $0x16,0x8(%esp)
    0x08048657 <+25>:  movl   $0xc,0x4(%esp)
    0x0804865f <+33>:  movl   $0x3,(%esp)
    0x08048666 <+40>:  call   0x80485f4 <average(int, ...)>
    0x0804866b <+45>:  mov    %eax,0x4(%esp)
    0x0804866f <+49>:  movl   $0x804a040,(%esp)
    0x08048676 <+56>:  call   0x80484c0 <_ZNSolsEi@plt>
    0x0804867b <+61>:  movl   $0x8048530,0x4(%esp)
    0x08048683 <+69>:  mov    %eax,(%esp)
    0x08048686 <+72>:  call   0x8048520 <_ZNSolsEPFRSoS_E@plt>
    0x080486df <+161>: mov    $0x0,%eax
20  }
    0x080486e4 <+166>: leave
    0x080486e5 <+167>: ret
```

va_list is used by va_arg to return the arguments from the list one by one. The function determines the types of arguments and passes this information to va_arg which uses it to return the correct variable type.

2.2 RELATED WORKS

2.2.1 *Existing Binary Analysis Tools*

There are a number of tools which use static analysis, dynamic analyis, or hybrid analysis. Some of these tools are discribed in the following:

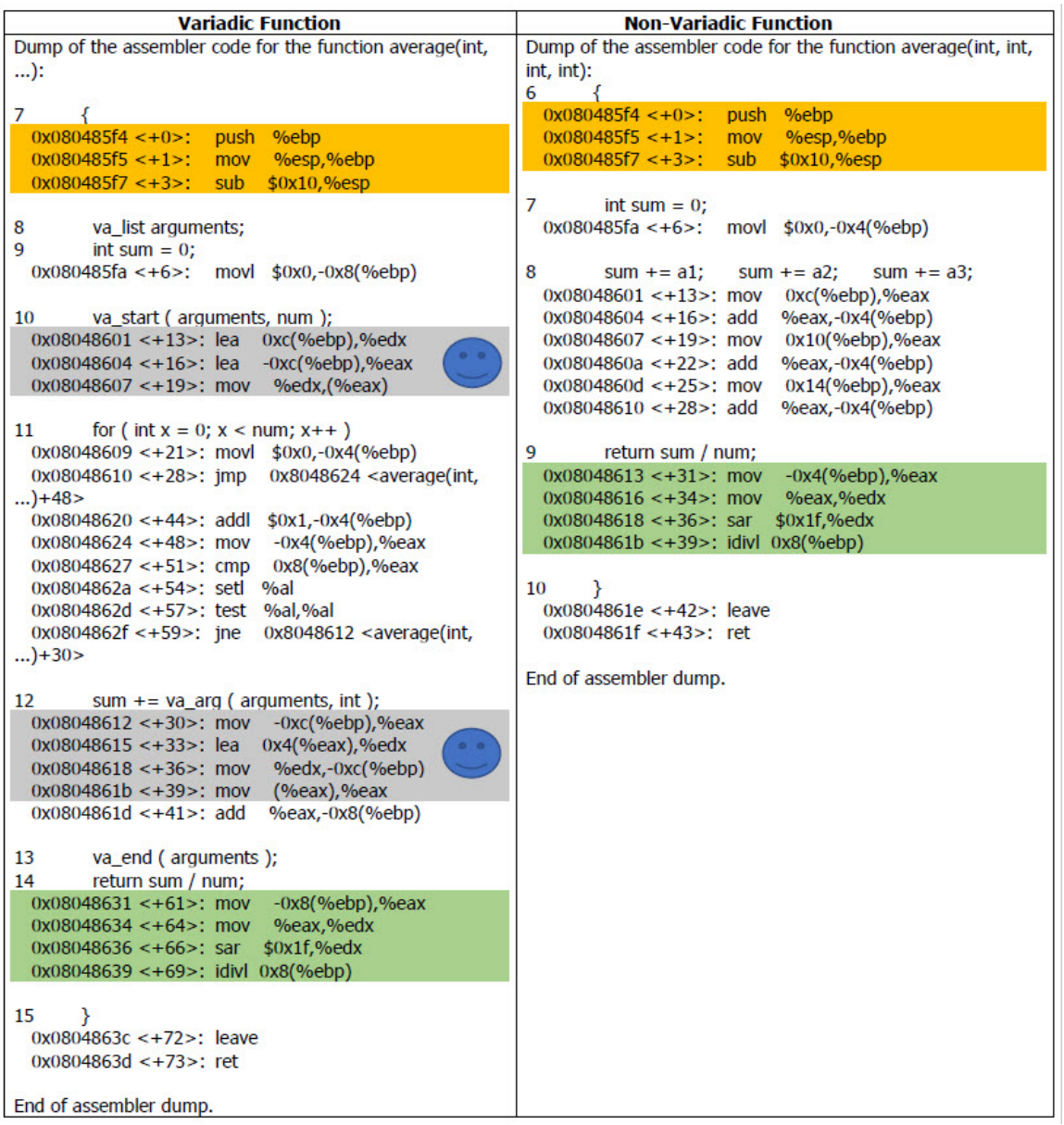
FindBugs [20] is an open source static analysis tool for Java programs that uses finely tuned analyzers called *Bug Detectors* to search for simple bug patterns. Bug patterns are defined as code idioms that are often errors. The detectors contain numerous heuristics to filter out warnings that may be inaccurate or incorrect. FindBugs has also been used on Sun's Java Development Kit (JDK), Sun's Glassfish J2EE server, Eclipse, and in portions of Google's Java codebase. In FindBugs, warnings are grouped into over 380 Bug Patterns which in turn are grouped into *Categories*

Listing 2.4: Caller of non-variadic function in binary code

```
12  {
    0x08048620 <+0>: push   %ebp
    0x08048621 <+1>: mov    %esp,%ebp
    0x08048623 <+3>: and    $0xfffffff0,%esp
    0x08048626 <+6>: sub    $0x10,%esp
13     cout<< average ( 3, 12, 22, 4 ) <<endl;
    0x08048629 <+9>: movl   $0x4,0xc(%esp)
    0x08048631 <+17>:  movl   $0x16,0x8(%esp)
    0x08048639 <+25>:  movl   $0xc,0x4(%esp)
    0x08048641 <+33>:  movl   $0x3,(%esp)
    0x08048648 <+40>:  call   0x80485f4 <average(int, int, int,
       int)>
    0x0804864d <+45>:  mov    %eax,0x4(%esp)
    0x08048651 <+49>:  movl   $0x804a040,(%esp)
    0x08048658 <+56>:  call   0x80484c0 <_ZNSolsEi@plt>
    0x0804865d <+61>:  movl   $0x8048530,0x4(%esp)
    0x08048665 <+69>:  mov    %eax,(%esp)
    0x08048668 <+72>:  call   0x8048520 <_ZNSolsEPFRSoS_E@plt>
    0x0804866d <+77>:  mov    $0x0,%eax
14  }
    0x08048672 <+82>:  leave
    0x08048673 <+83>:  ret
```

such as Correctness, Bad Practice, and Security. All of the bug pattern detectors are implemented using BCEL [21], which is an open source bytecode analysis and instrumentation library. The bug detectors can also be divided into four main categories of Class structure and inheritance hierarchy only, Linear code scan, Control sensitive, and Dataflow. The detectors that use dataflow analysis are the most complex, taking both control and data flow into account. The main limitation of FindBugs is that it largely ignores style issues. The main distinction between a style checker and a bug checker is that violations of style guidelines only cause problems for the developers working on the software. Whereas, warnings produced by a bug checker may represent bugs that will cause problems for the users of the software.

Valgrind [22] is one of the most widespread and used dynamic analyzer tools. One of the great features of Valgrind is support for *shadow values*, which can be used to create instrumentation tools that are difficult to build without this feature. An example of this is a tool that tracks the initialization of every bit in the program's data in order to show when the program accesses uninitialized data [23]. A Valgrind tool is created as a plugin, written in C Language, added to Valgrind's core. A tool



Variadic Function

```
Dump of the assembler code for the function average(int,
...):

7       {
   0x080485f4 <+0>:   push   %ebp
   0x080485f5 <+1>:   mov    %esp,%ebp
   0x080485f7 <+3>:   sub    $0x10,%esp

8           va_list arguments;
9           int sum = 0;
   0x080485fa <+6>:   movl   $0x0,-0x8(%ebp)

10          va_start ( arguments, num );
   0x08048601 <+13>:  lea    0xc(%ebp),%edx
   0x08048604 <+16>:  lea    -0xc(%ebp),%eax
   0x08048607 <+19>:  mov    %edx,(%eax)

11          for ( int x = 0; x < num; x++ )
   0x08048609 <+21>:  movl   $0x0,-0x4(%ebp)
   0x08048610 <+28>:  jmp    0x8048624 <average(int,
...)+48>
   0x08048620 <+44>:  addl   $0x1,-0x4(%ebp)
   0x08048624 <+48>:  mov    -0x4(%ebp),%eax
   0x08048627 <+51>:  cmp    0x8(%ebp),%eax
   0x0804862a <+54>:  setl   %al
   0x0804862d <+57>:  test   %al,%al
   0x0804862f <+59>:  jne    0x8048612 <average(int,
...)+30>

12          sum += va_arg ( arguments, int );
   0x08048612 <+30>:  mov    -0xc(%ebp),%eax
   0x08048615 <+33>:  lea    0x4(%eax),%edx
   0x08048618 <+36>:  mov    %edx,-0xc(%ebp)
   0x0804861b <+39>:  mov    (%eax),%eax
   0x0804861d <+41>:  add    %eax,-0x8(%ebp)

13          va_end ( arguments );
14          return sum / num;
   0x08048631 <+61>:  mov    -0x8(%ebp),%eax
   0x08048634 <+64>:  mov    %eax,%edx
   0x08048636 <+66>:  sar    $0x1f,%edx
   0x08048639 <+69>:  idivl  0x8(%ebp)

15      }
   0x0804863c <+72>:  leave
   0x0804863d <+73>:  ret

End of assembler dump.
```

Non-Variadic Function

```
Dump of the assembler code for the function average(int, int,
int, int):
6       {
   0x080485f4 <+0>:   push   %ebp
   0x080485f5 <+1>:   mov    %esp,%ebp
   0x080485f7 <+3>:   sub    $0x10,%esp

7           int sum = 0;
   0x080485fa <+6>:   movl   $0x0,-0x4(%ebp)

8           sum += a1;     sum += a2;     sum += a3;
   0x08048601 <+13>:  mov    0xc(%ebp),%eax
   0x08048604 <+16>:  add    %eax,-0x4(%ebp)
   0x08048607 <+19>:  mov    0x10(%ebp),%eax
   0x0804860a <+22>:  add    %eax,-0x4(%ebp)
   0x0804860d <+25>:  mov    0x14(%ebp),%eax
   0x08048610 <+28>:  add    %eax,-0x4(%ebp)

9           return sum / num;
   0x08048613 <+31>:  mov    -0x4(%ebp),%eax
   0x08048616 <+34>:  mov    %eax,%edx
   0x08048618 <+36>:  sar    $0x1f,%edx
   0x0804861b <+39>:  idivl  0x8(%ebp)

10      }
   0x0804861e <+42>:  leave
   0x0804861f <+43>:  ret

End of assembler dump.
```

FIGURE 2.1: The differences between variadic and non-variadic functions of Average function in GCC x86

plugin's main task is to instrument code fragments that the core passes to it. Due to this modular architecture, it is easy to develop a new tool to add to the Valgrind's core and extend its functionalities. While incredibly useful, supporting shadow values entails a more heavyweight approach to instrumentation that is unsatisfactory when efficiency is the primary goal. Also, Valgrind fails to detect dangling pointers to re-allocated data locations. The reason is that Valgrind tries to detect use-after-

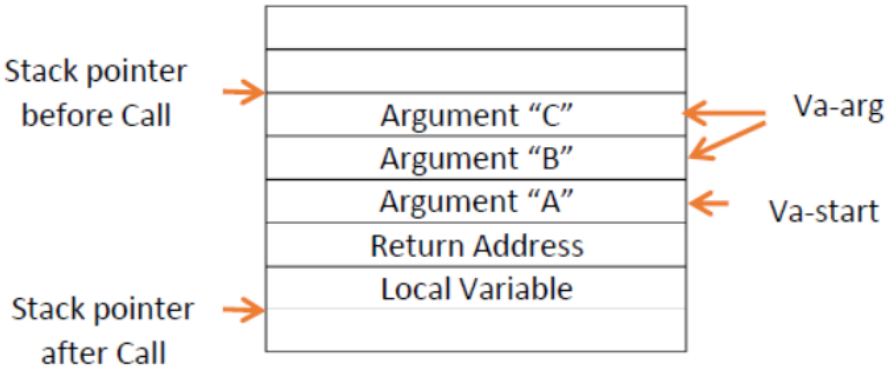


FIGURE 2.2: Arguments on the stack

free bugs by marking locations which were de-allocated in a shadow memory space. Accessing a newly de-allocated location can be detected this way. However, it fails to detect errors after the area is re-allocated for another pointer: the area is registered again and the invalid access remains undetected. Another limitation is that there is no support for Windows platforms.

Dr. Memory [15] is a dynamic analyzer tool to monitor and identify memory-related errors based on binary instrumentation. Unlike Valgrind, however, it runs on both Linux and Windows platforms and has less runtime. Dr. Memory wraps heap functions and checks for addressability of accesses. In addition, it propagates definedness through registers as well as memory, and raises errors for undefined accesses of consequence such as dereferencing an undefined pointer. One important limitation with tools such as Dr. Memory and Valgrind is that these tools only find out-of-bounds and use-after-free bugs for heap memory with (typically) no false positives [24]. The tools based on binary instrumentation cannot find out-of-bounds bugs in the stack variables (other than beyond the top of the stack) or globals.

PinOS [25] is an extension to Pin for whole-system instrumentation. Pin is a dynamic binary analysis tool that provides a high-level API for run-time instrumentation of programs. PinOS can be used to instrument both kernel and user-level code. To achieve whole-system instrumentation, PinOS is built on top of the Xen [14] hypervisor with Intel VT technology. Transparency is an important concern in dynamic binary instrumentation frameworks. For example, position-independent code calculates relative offsets and uses address modes when relative offsets are within supporting ranges. Therefore, a dynamic binary instrumentation framework may break such assumptions when relocating basic blocks, so it must change some instructions in the program to create an illusion that every address is the same as in a native run [26]. To achieve transparency, PinOS make use of hardware virtualization.

Chucky [27] is a method to expose missing checks in C source code by combining static tainting and techniques for anomaly detection. Instead of struggling with the limits of automatic approaches, Chucky is mainly useful at assisting a human analyst by providing information about missing security checks and potential problems. Additionally, it does not require external information and additional annotations

to identify missing checks. A bit more concretely, Chucky includes a five-step procedure, which can be executed for each source and sink referenced by a selected function: Robust parsing, Neighborhood discovery, Lightweight tainting, Embedding of functions, and Anomaly Detection. The robust parser first extracts conditions, assignments and API symbols from the source code. Then, functions in the code base operating in a similar context to that of the selected function using techniques inspired by natural language processing are identified. Lightweight tainting is then performed for the function under examination and all its neighbors in top-down and bottom-up direction to determine only those checks associated with a given source or sink. Next, the selected function and its neighbors are embedded in a vector space using the tainted conditions such that they can be analyzed using machine learning techniques. Finally, model of normality over the functions is computed, such that anomalous checks can be identified by large distances from this model. Chucky suffers from a few limitations: it cannot verify whether missing security checks truly lead to vulnerabilities in practice and therefore is better suited for finding vulnerabilities in stable code; it makes no attempts to evaluate expressions and semantically equivalent checks; it is opaque to the practitioner and thus a control or refinement of the detection process is impossible.

BitBlaze [28] combines virtualization and symbolic execution for software analysis available in an open-source release. One of the main advantages of BitBlaze is supporting unified binary analysis. This is particularly a useful feature for using common off-the-shelf (COTS) programs where source code is not attached. Not to mention that analyzing low-level code such as assembly or binary code is more difficult than analyzing source code due to the lack of structured information [29]. BitBlaze, however, is capable of doing so by means of root-cause analysis. BitBlaze is composed of three main components namely *Vine*, *TEMU*, and the *Rudder*. Vine is the static module of BitBlaze to translate the assembly instructions into a simple, formally specified intermediate language (IL). Additionally, a set of common static utilities such as control flow, data flow, optimization, symbolic execution, and weakest precondition calculation are provided by Vine. TEMU is the dynamic analysis module of BitBlaze, which provides a set of core utilities for extracting OS-level

semantics, user-defined dynamic taint analysis, and a clean plug-in interface for user-defined activities. Rudder enables mixed concrete and symbolic execution at the binary level, using the aforementioned utilities provided by Vine and TEMU. In Rudder, a solver (or decision procedure) is used to recognize the range of the memory region with a symbolic address. BitBlaze is a great and powerful analysis tool. However, similar to other available tools, it has some minor limitations such as lack of formal semantics for the IL itself and it does not handle bi-endian architectures such as ARM correctly [30].

DARWIN [31] is a automated and scalable debugging methodology for root causes of errors using software regressions. Software regression testing is a well-known concept in most software development projects. In its simplest form, it involves re-testing a test suite as a program changes from one version to another. Mathematically speaking, given two program versions P, $P\prime$, and a test t which passes in P while failing in $P\prime$- the aim is to find a bug report explaining the root cause of the failure of t in $P\prime$. DARWIN constructs and composes the path conditions of test t in program versions P, $P\prime$ in trying to come up with a bug report explaining an observed regression. A notable feature of DARWIN approach is that it handles hard-to-explain bugs, like code missing errors, by pointing to code in the reference program. DARWIN is built on top of the BitBlaze platform. Unlike BitBlaze, it has dedicated modules for formula manipulation and optimization. The only known limitation with DARWIN is running time. On a test-case performed by the authors in [32], DARWIN took 543 minutes (or 9 hours) to perform the debugging.

A Binary Analysis Platform, aka BAP [30], is a complete re-design of Vine with some additional features. It provides platform-independent utilities to extract control flow graphs and program dependence graphs, to perform symbolic execution and to perform precondition calculations. Unlike BitBlaze, BAP is designed with the BAP Intermediate Language (BIL) which is a small and formally specified language to model instruction evaluation as compositions of variable reads and writes in a functional style. BAP is equipped with a front-end component for lifting binary code for the supported architectures to the BIL, and a back-end component which is the implementation of program analyses and verifications for low-level code. The

prominent features of BAP include: common code representations such as program dependence graphs, value set analysis, verification capabilities using Dijkstra and Flanagan-Saxe style weakest pre-conditions and interfaces with several SMT solvers, and code optimization. Limitations of BAP's front-end are [33]: it supports only ARMv4, it does not manage the processor status registers, and it does not handle banked registers for the privileged modes and coprocessor management.

Similar to PinOS, SPIDER [26] is a stealthy program instrumentation framework based on hardware virtualization technique. In order to provide sufficient transparent trapping, a novel primitive called invisible breakpoint is proposed. As opposed to traditional breakpoints, all the side-effects of an invisible breakpoint are hidden from the guest to guarantee transparency. SPIDER uses Extended Page Tables (EPT) to implement such invisible breakpoints at the hypervisor level to avoid any unexpected in-guest execution. Additionally, Spider provides data watch point which enables the monitoring of memory read/write at any address. Having set invisible breakpoints, SPIDER is able to trap the execution of program at arbitrary guest physical address. However, since paging is enabled in majority of modern operating systems, the processor often uses virtual address instead of physical address to reference memory. This means, it is more desirable to have the ability to trap the execution of program at arbitrary guest virtual address in the program's address space. Instead of monitoring every change of virtual-to-physical mapping, SPIDER only needs to monitor the change of virtual-to-physical mapping at each breakpoint address. This avoids using heavy-weight techniques such as shadow page tables. The main limitation of SPIDER is its scalability. This is because determining where to place breakpoints with SPIDER is a manual process and requires either manual action or an in-guest agent, which either prohibits scalability or hinders the stealth capability of the system.

Peach fuzzing platform [34] is a popular fuzzer for vulnerability discovery. As already mentioned, fuzzing is a dynamic technique in which malformed input is provided to an application in an attempt to trigger a crash or other undesired behavior. Peach was originally released in 2004 by the IOACTIVE, implemented in Python language. Being more of a black-box fuzzer, PEACH forgoes any analysis

of the program under test and simply generates lots of inputs to be executed against the program under test. The input of Peach fuzzer is a Peach-pit file that describes the targeted input format to be able to produce syntactically valid inputs. The main limitation of Peach and other fuzzing tools which construct malformed input data from predefined format specifications (such as Peach-pit) is that the cost of generating production rules used by fuzzing tools is difficult especially when the format specifications are undocumented and the source code of the application is not available [35].

Automatic Exploit Generation (AEG) [36] was the first system to tackle the problem of both identifying bugs (memory corruption vulnerabilities) and automatically generating exploits. More specifically, AEG introduced preconditioned symbolic execution as a way to focus symbolic execution towards a particular part of the search space. As already explained in previous section, symbolic execution techniques bridge the gap between static and dynamic analysis and provide a solution to cope with the limited semantic insight of fuzzing. The key idea is to use symbolic values as input instead of actual data, and to represent values of program variables as symbolic expressions [37]. As a result, the outputs computed by a program are expressed as a function of the symbolic inputs. The initial AEG system worked solely on source code to find bugs, then used dynamic binary analysis to generate control-flow hijack exploits. Later on (in 2010), Mayhem was introduced as an AEG tool on executable code. Mayhem was able to manage symbolically executed program paths, reasoning about symbolic memory addresses without exhausting memory. AEG worked solely on source code. The power of symbolic execution provided by both AEG and Mayhem, however, comes at a price: The analysis often suffers from a vast space of possible execution paths, in particular for large programs. This means automatic exploit generation by means of semantic execution techniques is far from being solved. Table 2.1 briefly summarizes the mentioned tools.

TABLE 2.1: Summary of tools for binary analysis

Name	Method	Language Support	Main Features	Limitations
AEG	Dynamic (Symbolic)	Both source and binary codes	Automatically finds bugs and generates working exploits, providing actionable information to help developers decide which bugs to fix first.	Scalability issues (not practical for programs with deep bugs).
BAP	Dynamic	Assembly	Common code representations (e.g., static single assignment, a dataflow framework with constant folding, dead code elimination), Verification capabilities via Dijkstra and Flanagan-Saxe style proofs. Precise modeling of x86 instructions, which includes updating of CPU flags	Does not manage the processor status registers. Does not handle banked registers for the privileged modes co-processor management.
BitBlaze	Fusion of static and dynamic techniques	Assembly	Semantics based Binary centric approach Handling packed/encrypted/obfuscated code	Lack of a formal semantics for the IL. Some reported instruction issues in Vine such as dealing with big-endian memory operations.
Chucky	Static	C	Identification of missing checks. Anomaly detection on conditions. Top-down and bottom-up analysis.	Makes no attempts to evaluate expressions and semantically equivalent checks. Opaque to the practitioner and thus control or refinement of the detection process is impossible.
DARWIN	Dynamic	Assembly	Identifying failure-inducing code changes for evolving softwares.	Long processing time.

Continued on next page

Table 2.1 – *Continued from previous page*

Name	Method	Language Support	Main Features	Limitations
Dr. Memory	Dynamic	NA	Monitoring and identifying memory-related errors based on binary instrumentation.	Does not handle out-of-bounds errors for most stack variables and globals.
FindBugs	Static	Java	Searching simple bug patterns. Bug detectors are implemented using BCEL	Uses existing patterns.
Peach	Dynamic (Fuzzing)	NA	Generates malformed inputs and feeding them to the test program that result in undesired behavior.	Difficulty in generating malformed inputs that provide sufficient program coverage.
PinOS	Dynamic	NA	Performs pervasive fine-grain instrumentation (whole-system instrumentation) Uses hardware virtualization (Xen hypervisor).	Insufficient transparency Lack of support for OS-level semantics extraction and layered annotative execution.
SPIDER	Dynamic	NA	Flexible instruction-level trapping based on hardware virtualization. Transparent (by implementing invisible breakpoints). Does not introduce high overhead on the target program	Manual intervention to set invisible breakpoints as would normally be done with debugging.
Valgrind	Dynamic	C	Support for shadow values. Easy to write new plugins.	Fails to detect dangling pointers to reallocated data locations. Does not handle out-of-bounds for most stack variables and globals. Runtime overhead (in excess of 10x).

2.2.2 *Existing Mechanisms for Mitigating Vulnerabilities of Variadic Functions*

There are some existing mechanisms which have the ability to mitigate vulnerabilities in both source code and binary code of variadic functions. Some of these mechanisms are introduced in this section.

Format Guard [5] prevents format vulnerabilities in the program by using a variation on argument counting in the source code. Format Guard uses specific properties of GNU CPP (the C Pre-processor) to extract the number of arguments then send the number of arguments to the safe printf wrapper. The safe wrapper parses the format string and compares the number of arguments. Format Guard has capability to alert and kill the process if the format string calls more arguments than the wrapper has.

Memory corruption can occur by attackers. First, the attackers identify the memory layout and reveal the information then execute the actual exploit of memory that can be stack, heap, or libraries [38]. Address Space Layout Randomization (ASLR) [6] does not let attackers jump to the exploited function in memory by randomly changing the start of the address space. Therefore, the attackers cannot recognize the location of the libraries, stack and heap because the memory layout is randomized for each execution.

Data Execution Prevention (DEP) [39] has the capability to stop attackers who try to insert and execute the code in non-executable memory by marking all memory locations in a process as non-executable unless the location contains executable code. Modern operating systems use data execution prevention as a security feature since DEP lets only data [40] in executable location to be executed by programs. DEP does not have the capability to prevent attacks that do not rely on executed instruction in the data area.

Control Flow Integrity (CFI) [41] protects applications from attackers who want to control the behavior of the program by redirecting the flow of execution arbitrarily, controlling the program behavior. It cannot stop all variadic function attacks though, since a huge number of variadic function prototypes exist and CFI relies on the function prototypes, and therefore it can only catch a subset of them. CFI needs an

accurate control flow graph (CFG) of the source code to detect the advanced code-reuse attacks where attackers misuse the executable code which is already present in the memory.

2.2.3 *Existing Tools for Detecting Functions in Binary Code*

There are some tools that have been designed to detect functions in binary code. These tools have been introduced in the following.

Bao et al. [42] proposed BYTEWEIGHT which is a new automatic function identification algorithm. This approach automatically learns key features for recognizing functions and can therefore easily be adapted to different platforms, new compilers, and new optimizations. They evaluated their tool against three well-known tools that feature function identification: IDA, BAP, and Dyninst. Function boundary results of BYTEWEIGHT were evaluated on the data set which includes benchmarks of SPEC CPU 2017 compiled with GCC and ICC compilers, from no optimization to the highest optimization level for both Intel x86 and x64 architectures. For the Intel x86 architecture, the average precision of BYTEWEIGHT is around 92.78%, and the recall average is 92.29%. The F1 score is around 92.53%. For the Intel x64 architecture, the average precision of BYTEWEIGHT is around 93.22%, and the recall average is 95.52%. The F1 score is around 92.87%.

Shin et al. [43] proposed a new approach for recognizing Functions in Binaries using Neural Networks. Binary analysis facilitates many important applications like malware detection and automatically fixing vulnerable software. In this paper, the authors propose applying artificial neural networks to solve important yet difficult problems in binary analysis. Specifically, the authors tackle the problem of function identification, the crucial first step in many binary analysis techniques. Function boundary results of Shin et al. were evaluated on the data set which includes benchmarks of SPEC CPU 2017 compiled with GCC and ICC compilers, from no optimization to the highest optimization level for both Intel x86 and x64 architectures. For the Intel x86 architecture, the average precision of the proposed approach is around 97.75%, and the recall average is 95.34%. The F1 score is around 96.53%. For

the Intel x64 architecture, the average precision of Shin et al. is around 94.85%, and the recall average is 89.91%. The F1 score is around 96.53%.

Andriesse et al. [44] introduced Nucleus, a "compiler-agnostic" function detection algorithm for binaries. The authors use linear disassembly coupled with in-line data and padding code detection followed by basic block detection. These blocks are then connected into control flow graphs. Based on the premise that intraprocedural control flow instructions tend to be different than interprocedural instructions, they isolate subgraphs of basic blocks for each function. The function detection code of Nucleus is publicly available [45] and therefore was used in our comparisons. Function boundary results of Nucleus were evaluated on the data set which includes benchmarks of SPEC CPU 2017 compiled with GCC, Clang, and ICC compilers, from no optimization to the highest optimization level for both Intel x86 and x64 architectures. For the Intel x86 architecture, the average precision of Nucleus is around 97.75%, and the recall average is 95.34%. The F1 score is around 96.53%. For the Intel x64 architecture, the average precision of Nucleus is around 94.85%, and the recall average is 89.91%. The F1 score is around 96.53%.

Qiao et al. [16] presented an approach that recovers high-level information of the function in COTS binaries. This information includes calls, return values, types and numbers of function parameters. Accuracy of recovering this information is important to identify function boundaries. Their approach recovers function calls and return values by static analysis. The static analysis lets the approach have both forward and backward views. Then, the system recovers the entry and exit points of the function by checking control flow and data flow techniques. The entry and exit points of function help traverse a function body. The approach checks all possible paths until control flow exits the function. Function boundary results of Qiao et al. were evaluated on the data set which includes benchmarks of SPEC CPU 2006 with GCC and ICC compilers, from no optimization to the highest optimization level for both Intelx86 and x64 architectures. For the Intel x86 architecture, the average precision of Qiao et al. is around 98.65%, and the recall average is 98.09%. The F1 score is around 98.37%. For the Intel x64 architecture, the average precision of Qiao

et al. is around 99.12%, and the recall average is 99.00%. The F1 score is around 99.06%.

Alves-Foss et al. [18] introduced JIMA which takes stripped binaries and returns a list of possible function boundary locations. JIMA starts the process of detecting functions in binary code by sorting list of possible function addresses. JIMA finds all control flow operations such as calls and jumps in linear disassembly generated by objdump. It then sets the first address as a possible function start address and the next one as the next function address. The algorithm processes instructions from the start address until it reaches the next function address or exit of the function. Function boundary results of JIMA were evaluated on the data set which includes benchmarks of SPEC CPU 2017 compiled with GCC, Clang, and ICC compilers, from no optimization to the highest optimization level for both Intel x86 and x64 architectures. For the Intel x86 architecture, the average precision of JIMA is around 99.60%, and the recall average is 99.79%. The F1 score is around 99.70%. For the Intel x64 architecture, the average precision of JIMA is around 99.89%, and the recall average is 99.92%. The F1 score is around 99.90%. As the results show, JIMA detects function starts and bounds better than other existing tools.

2.2.4 *Existing Tools for Extracting Information about Functions in Binary Code*

There have been several research efforts on inference and extraction of high-level information such as variable types, data structures, and object-oriented entities and functions from executable binaries using both static and dynamic analysis techniques. Some tools built for binary analysis provide function identification as part of their functionality, usually using relatively simple heuristics or hand-coded signatures such as Dyninst [46] which provides a high-level platform-independent interface for dynamic binary analysis and uses the signatures to recognize function starts in ELF x86 binaries.

Alrabaee et al. [47] proposed Semantic Integrated Graph (SIG) which is a novel representation of binary code to identify reused functions by matching traces of control flow graph, register flow graph, and function call graph. These program analysis methods have been merged with each other in a joint data structure. SIG

consists of different types of traces such as normal traces, AND-traces, and OR traces to analyze the matching of control flow graphs, register flow graphs, and function call graphs. If the result exactly matches it means that the function was reused in binary code, but if the result demonstrates the inexact matching, this approach uses the graph edit distance algorithm to measure the degree of similarity between the result and the actual function definition.

Li and Wang [48] developed a unified framework to support both cross-platform analysis and interactive analysis in binary codes by applying symbolic execution and taint tracking techniques upon the popular IDA Pro tool. Interactive analysis lets users mark some sources such as memory locations or registers based on their assumptions to check quickly the targeted instructions. Cross-platform analysis works in different instructions without depending on architectures and platforms. They support the cross-platform analysis by adopting a unified binary code intermediate representation (IR) and implementing the core analysis algorithms of symbolic execution and taint tracking on this IR. Cross-platform analysis performs a symbolic execution engine on the platform-independent REIL IR [49] which has been designed for static code analysis. Taint analysis lets the user mark the targeted source and then symbolizes the value of that source. The user can analyze different inputs to detect the vulnerabilities or use the input of the program as a tainted source. Since this approach supports the interactive analysis, the user has the ability to follow the taint sinks to mark different memory locations or registers.

2.2.5 *Existing Tools for Detecting Variadic Functions in Binary Code*

There are two tools that have been designed for detecting variadic functions. They are described in this section.

Biswas et al. [11] proposed Hexvasan which detects vulnerable variadic functions at runtime by counting and checking the number and type of the arguments in variadic functions. Hexvasan was implemented on top of the LLVM compiler framework. Hexvasan consists of two main components: static analysis and dynamic analysis. Static analysis works on LLVM IR and it targets variadic functions by parsing the source code files. Based on the prototype of functions, Hexvasan will

TABLE 2.2: Summary of tools for detecting variadic functions

Tools	Architecture	Compiler	Requirement
TypeArmor	x64	GCC-Clang	Non- stripped binary code
Hexvasan	x64	Clang	Source code and Non-stripped binary code
Detector	x86 – x64	GCC-Clang-ICC	Stripped binary code

decide which instrumentation should be run on LLVM IR. By applying dynamic analysis, Hexvasan can verify a mismatch of the number and type of the arguments at runtime. Hexvasan has the capability to capture and store the information from the call sites into the metadata. This metadata compares and verifies the number and type of arguments that they have passed from the caller with the extracted arguments of the callee. If Hexvasan recognizes mismatching number or type of the arguments then it will abort the program.

Veen et al. [50] proposed TypeArmor which recovers the call site signatures and callee prototypes to prevent code-reuse attacks in binaries, without accessing the source code. TypeArmor uses static analysis to capture the number of arguments between the call sites. It recovers call site information by mapping caller to callee with data flow abstractions such as reaching definition and liveness analysis to count the arguments. For this purpose, the authors leverage the Dyninst binary analysis framework.

Table 2.2 shows the differences between the existing tools for detecting variadic functions and our tool Detector. Both TypeArmor and Hexvasan support only the Intel X64 architectures but the Detector can recognize the variadic functions in binary code in both Intel X86 and X64 architecture. TypeArmor can detect variadic functions in C and C++ source code that has been compiled by GCC and Clang. Hexvasan can support the C and C++ source code that has been compiled by Clang. Our tool, Detector, is different and can analyze C and C++ code which has been compiled by GCC, Clang, or ICC. The requirement of Detector and TypeArmor is just the binary code but Hexvasan needs the source code besides the binary code.

CHAPTER 3

Design of The Detector

3.1 OVERVIEW OF APPROACH

This chapter of the dissertation presents a new approach for automatically identify the variadic functions in stripped binaries by using a combination of reverse engineering and static binary analysis techniques. Specifically this chapter focuses on applying this technique to our proof of concept tool, Detector. The approach focuses on stripped Intel x86 and x64 instruction set architectures from C/C++ progress that are compiled by GCC, Clang, or ICC. Analyzing stripped binary code is very challenging since stripped binary code only includes the low-level information such as instructions and register uses. The approach applies syntactic and semantic analysis to recognize the behavior of the variadic functions. Static analysis allows Detector to determine variadic functions without executing the code. Detector identifies variadic functions with patterns that have been defined as the syntax of the variadic functions. Each compiler and architecture has its own syntax of variadic functions. Therefore, the Detector uses semantic analysis to track the valid path of syntax in different compilers.

Figure 3.1 shows the components of Detector in a high-level design. The input of Detector is the stripped binary code without the source code, debugging information, or a Symbol table. For Detector to work, some features of the binary must be defined such as the boundary of the functions in the program, type of the compiler, and optimization level (-O0 to -O3) as discussed in Section 3.2. We assume we get function boundaries from JIMA, as discussed in Section 2.2.3 and we also assume the programs use one of the standard calling conventions. First, Detector maps stripped binary code to the defined data representation. Since Detector needs to know the type of the compiler and architecture, it uses tags based on the low-level information of the data representation to determine them. This low-level information includes register type, size, and the prologue of the function in the

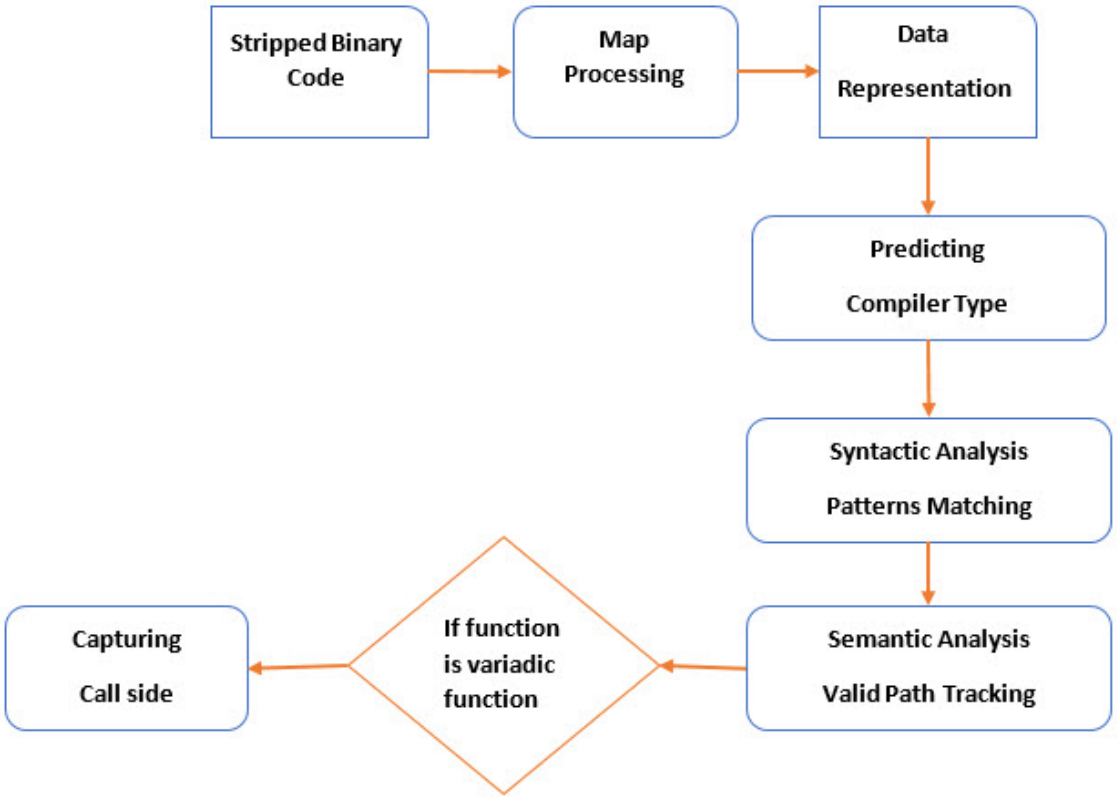


FIGURE 3.1: Detector design in high-level abstraction

binary code. Syntactic analysis with pattern matching allows Detector to identify variadic functions statically in stripped binary code.

These patterns include patterns of argument registers, floating-point arguments, offset of the argument registers, last non-variadic argument, and the va_arg section. Then, Detector uses semantic analysis to track the valid path of patterns. When a match is found, Detector captures the call side of the variadic functions. Variadic functions can be called directly and indirectly from other functions. Developers can use this information to know which functions are affected by calling variadic functions.

The approach consists of the following main components for identifying variadic functions in stripped binary code that are compiled with GCC, Clang, and ICC with different optimization levels.

- Static analysis

- Designing new data representation
- Map processing
- Syntactic analysis
- Identifying type of compiler and architecture
- Pattern matching
- Semantic analysis
- Capturing call-side information

We performed enhanced static analysis by pattern matching to observe the behaviour of the variadic functions. First we designed a new abstract data representation to define the syntax of variadic functions. Then, we applied semantic analysis to track the valid paths of patterns based on the type of the compiler and architecture.

3.2 ASSUMPTIONS

The prerequisites and assumptions for Detector are addressed as follows:

- **Function boundaries**: Function boundaries give the start address and end address of all functions in the program. This information is necessary for Detector to distinguish variadic functions from all other functions in the program. The Detector uses JIMA [18] to access the function boundaries in the program. JIMA takes stripped binary code and returns a list of function boundaries by using both static and behavioral analysis. Detector uses JIMA since the precision and accuracy of it is better than the other available tools such as IDA 7.0 [51], Ghidra 9.0.1 [52], and Nucleus [44].
- **Compiler type and optimization level**: This dissertation focuses on three compilers GCC, Clang, and ICC in both Intel x86 and x64 architectures. Therefore, the input to the Detector should be stripped binary code that is the output of one of the GCC, Clang, and ICC compilers, in one of four optimization levels. The performance for other compilers is not studied.

- **Standard calling convention**: The C program uses two primary calling conventions. Detector requires that the programs use one of these CDECL or STDCALL. The first calling convention is CDECL [53] which passes the arguments through the registers or stack in right-to-left order, and the returned values are stored in the register eax. The calling function cleans the stack. This allows CDECL functions to have variable-length argument lists (aka variadic functions). For this reason, the number of arguments is not appended to the name of the function by the compiler, and the assembler and the linker are therefore unable to determine if an incorrect number of arguments is used [54]. The second calling convention is STDCALL, which passes the arguments the same way as in CDECL on the stack in right-to-left order or registers [55]. C++ programs use the THISCALL standard calling convention where the pointer points to the class object that is passed in ecx, the arguments are passed right-to-left on the stack or registers, and the return value is stored in eax register [56].

3.3 STATIC ANALYSIS

The main goal of this dissertation is to design a process to detect variadic functions from C and C++ programs compiled by GCC, Clang, and ICC compilers, and implement a proof of concept tool. Static analysis allows us to determine variadic functions without executing the programs. Static analysis techniques can analyze all possible control flows of a program, achieving a significantly higher coverage of program vulnerabilities and, as a result, produce a significantly lower false-negative rate compared to dynamic analysis approaches.

Our approach applies static analysis to automatically identify variadic functions in binary code. Static analysis consists of two main sections, Syntactic analysis and Semantic analysis. Syntactic analysis helps to identify the patterns of the variadic functions, and semantic analysis helps to track the valid path of patterns.

Because of the absence of source code, analyzing the stripped binary file is very challenging, and there is no information about size, location, layout of the function, function parameters, or local variables. Therefore, we defined a new data

representation to analyze the stripped binary code. This new data representation is described in Section 3.4.

3.4 ABSTRACT DATA REPRESENTATION

As mentioned in Section 2.1, some binary analysis tools designed a new language representation such as BAP [30] and Valgrind[22], or some of them adopted the available language representations such as Hexvasan [11] and SecondWrite[57] which used LLVM IR.

Our approach describes a new data representation to analyze the stripped binary code. The data representation lets Detector analyze the stripped binary code by applying syntactic analysis. That means the instructions of binary code can have more meaning than just address, data, source, and destination in binary code. This data representation has been designed based on the specific requirements that describe the syntax of the variadic functions. In the following, more information has been described for designing and analyzing this data representation.

3.4.1 *Syntactic Analysis of Variadic Functions*

Syntactic analysis determines how the symbols or characters of the language can be combined with each other [58]. The Intel x86 assembly language has been used in two main syntax branches, namely, Intel and AT&T. Our approach defines the grammar of data representation based on the AT&T syntax. The order of parameters in AT&T is source before destination, and also, the size of the operator is included in the opcode and can be b for byte, q for qword, l for long (dword), and finally w for word. Syntactic analysis with data representation grammar maps all instructions in the stripped binary code to a specific instruction format(Id, Address, Opcode [Args]). Table 3.1 describes the symbols with the "::= " read as "is defined to be" and "|" which means the symbols can be replaced based on the rules. The syntax of the grammar has been defined by the permutations of the Intel x86 and x64 instructions in Table 3.1. In the next section, permutations of the symbols of Opcodes and Args are described.

TABLE 3.1: Symbolic definition of the AT&T syntax

Symbols	Definitions
<Id>::= <0>|<1>| ... |<N>	Ids help to track and find instructions easily. The numbers are based on the number of instructions in the decoded executable binary file.
<Address>::=<Address0>| <Address1>|...<AddressN>	Address shows all the instruction addresses.
<Opcode>::= <mov>|<push>|<pop>|<lea>| <add>|<sub>|<inc>|<dec>| <imul>|<idiv>|<and>|<or>| <xor>|<not> |<neg> |<shl>| <shr> |<jmp>|<jconditions> | <cmp> | <call>	<mov> copies the data item from one operand to another. <push> places the data item on the top of the stack. <pop> removes the data item from the top of the stack. <lea> load effective address places the address specified by the source to destination. <add> adds the two operands and stores the result in the second one. <sub> subtracts the first operand from the second one and stores the result in the second one. <inc> increments the value of the operand. <dec> decrements the value of the operand. <imul> multiplies two operands or three operands. <idiv> divides the contents of the second operand by the second one and stores the result in the second operand. <and><or><xor> used for specified logical operations. <not> negates logically the contents of the operand. <neg> negates the contents of the operand. <shl> shift left. <shr> shift right. <jmp> jumps to another instruction. <jconditions> jump based on the conditions equal, not equal, grater than, less than, etc.. <cmp> compares the operands with each other. <call> is the subroutine of the calling instructions.

Continued on next page

Table 3.1 – *Continued from previous page*

Symbols	Definitions
<Args>::= <[reg,reg]>|<[reg,mem]>| <[mem,reg]>|<[const,reg]>| <[const,mem]>|<[reg]>| <[mem]>|<[const]>| [<reg><reg><const>] |[<reg><mem><const>|<Address>	Reg means registers, Mem means memory locations , Const means constant numbers

3.4.2 *Context-Sensitive Grammar*

The new language of the approach which has been defined to read and analyze the stripped binary code is based on context-sensitive grammar (CSG) [59]. Since the approach just focuses on AT&T syntax, the permutations of the symbols of assembly Intel x86 and x64 in grammar are very important. In the following, the permutations of the critical Opcodes and Args symbols have been demonstrated. Map processing, which is described in the next section, extracts the required information and makes the data representation from compilers. Note that we only focus on Opcodes that are relevant for detecting variadic functions.

- <mov><Args> ::= <mov><[reg,reg]>| <mov><[reg,mem]>| <mov><[mem,reg]>| <mov><[const,reg]>| <mov><[const,mem]>
- <push><Args> ::= <push><[reg]> |<push><[mem]>| <push><[const]>
- <pop><Args> ::= <pop><[reg]> |<pop><[mem]>
- <lea><Args> ::= <lea><[mem,reg]>
- <add><Args> ::= <add><[reg,reg]>| <add><[reg,mem]>| <add><[mem,reg]>| <add><[const,reg]>| <add><[const,mem]>
- <sub><Args> ::= <sub><[reg,reg]>| <sub><[reg,mem]>| <sub><[mem,reg]>| <sub><[const,reg]>| <sub><[const,mem]>
- <imul><Args> ::= <imul>< [reg,reg,const] | <imul> [reg,mem,const]

- <idiv><Args >::= <idiv><reg> | <idiv><mem>
- <and><Args> ::= <and><[reg,reg]>| <and><[reg,mem]>| <and><[mem,reg]>| <and><[const,reg]>| <and><[const,mem]>
- <or><Args> ::= <or><[reg,reg]>| <or><[reg,mem]>| <or><[mem,reg]>| <or><[const,reg]>| <or><[const,mem]>
- <xor><Args> ::= <xor><[reg,reg]>| <xor><[reg,mem]>| <xor><[mem,reg]>| <xor><[const,reg]>| <xor><[const,mem]>
- <not><Args> ::= <not><reg> | <not><mem>
- <neg><Args> ::= <neg><reg> | <neg><mem>
- <shl><Args> ::= <shl><[reg,reg]>| <shl><[reg,mem]>| <shl><[const,reg]>| <shl><[const,mem]>
- <shr><Args> ::= <shr><[reg,reg]>| <shr><[reg,mem]>| <shr><[const,reg]>| <shr><[const,mem]>
- <jmp><Args> ::= <jmp><Address>
- <jcondition> ::= <je><Address>| <jne><Address>|<jz><Address>|<jg><Address> |<jl><Address> |<jge><Address> |<jle><Address>
- <cmp><Args> ::= <cmp><[reg,reg]>| <cmp><[reg,mem]>| <cmp><[mem,reg]>| <cmp><[const,reg]
- <call> <Args> :: = <call> <Address>

3.5 MAP PROCESSING

Map processing is responsible for building a map from native structures in Intel x86 and x64 binary code to defined abstract representation. Map processing gives the capability to Detector to detect patterns of variadic functions syntactically. Our

TABLE 3.2: Example of mapping from binary x86 to abstract representation

binary Instruction (x86)	Abstract Representation Instruction
`0x4008fa movl $0x8,0xd0(%rbp)`	`inst (131 0x4008fa movl [(imm 0x8),(mem [(offset -0xd0),(base rbp)])])`
`0x400904 movl $0x30,0xcc(%rbp)`	`inst (132 0x400904 movl [(imm 0x30),(mem [(offset -0xcc),(base rbp)])])`
`0x40090e lea 0x10(%rbp),%rax`	`inst (133 0x40090e leaq [(mem [(offset 0x10),(base rbp)]),(reg rax)])`
`0x400912 mov %rax,-0xc8(%rbp)`	`inst (134 0x400912 movq [(reg rax),(mem [(offset -0xc8),(base rbp)])])`
`0x400919 lea -0xb0(%rbp),%rax`	`inst (135 0x400919 leaq [(mem [(offset -0xb0),(base rbp)]),(reg rax)])`
`0x400920 mov %rax,-0xc0(%rbp)`	`inst (136 0x400920 movq [(reg rax),(mem [(offset -0xc0),(base rbp)])])`

approach maps all the lines in the stripped binary code to the specific list of abstract instructions (Id, Address, Opcode [Args]). Table 3.2 shows the mapping from a few instructions used by the average function, which is a variadic function, to abstract representation. In this table, the native Intel x86 instructions are shown in the first column, and the corresponding abstract representation instruction is shown in the second column. After mapping the binary code to our abstract representation, the detection of the sequence patterns of the variadic function will be started.

3.6 IDENTIFYING THE TYPE OF COMPILER AND ARCHITECTURE

Detector needs to know the type of the compilers and the level of optimization because the compilers have different structures for translating the source code to binary code in different levels of optimization. Therefore, recognizing the patterns of syntax and tracking the semantic valid path of patterns depends on the type and level of compiler optimization. Detector detects the type of compiler and architecture based on the type of temporary registers, sizes of integers and memories, and the prologue of the function call. As an example, the size of registers and memory on Intel x86 architecture is 32 bits while it is 64 bits on Intel x64. The prologue of the function without optimization uses base pointer (rbp) and with optimization

TABLE 3.3: Tags and the type of compilers with their architectures

Tags	Compilers & Architectures
MV	GCC (X64)
CL	Clang (X64)
IC	ICC (X64)
GS	GCC (X86)

uses stack pointer (rsp) in x64 architecture. The prologue of the function without optimization uses base pointer (ebp) and with optimization uses stack pointer (esp) in Intel x86 architecture. Detector uses a label which is called "tag" in the program to categorize the type of compilers and the architectures. Table 3.3 shows the tags and the type of compilers with their architectures.

3.7 PATTERN MATCHING

The main goal of Detector's syntactic analysis is to statically detect the variadic functions. To achieve this purpose, the behavior of these functions should be considered. The variadic function takes arguments and then stores them on the stack. Some key features such as existing argument registers, floating-point arguments, and passing and storing the arguments on the stack can help to determine whether the function is a variadic function or not.

3.7.1 *Status of the Registers*

Analyzing the status of the registers helps to recognize and capture the transactions and dependency of the registers with each other [60]. Detector records the value and address of the registers that demonstrates the variadic function behavior to distinguish between the local variables and the arguments based on the assembly instructions which include the following four classes:

- **Arithmetic**: add, sub, mul, imul, div, idiv, etc.
- **Logical**: and, or, xor, test, shl.
- **Generic**: mov, lea, call, jmp, jle, etc.

LISTING 3.1: Argument registers in variadic function of benchmark XZ-R

```
1  400891: mov    %rdi,-0xe4(%rbp)
2  400897: mov    %rsi,-0xa8(%rbp)
3  40089e: mov    %rdx,-0xa0(%rbp)
4  4008a5: mov    %rcx,-0x98(%rbp)
5  4008ac: mov    %r8,-0x90(%rbp)
6  4008b3: mov    %r9,-0x88(%rbp)
```

- **Stack**: : push, pop.

Paying attention to the Generic and Stack classes is important for analyzing the behavior of the variadic function in binary code. Generic class which includes call or jump might show the function calls. Also, the variadic function uses the stack class to push the arguments on the stack; therefore, following these two classes increases the precision of detecting variadic functions. Registers generally have three statuses in assembly code: read-before-write, write-before-read and clear. Based on their status, Detector can guess the number of arguments. Listing 3.1 demonstrates the argument registers in one of the variadic functions in benchmark XZ-R, which is compiled by GCC x64. If rdi, rsi, and rdx have read-before-write statuses, then the function has at least three arguments. But if the last one, r9, has read-before-write status, then Detector can assume this function at least takes six arguments. A variadic function has to assume six or more argument where a non-variadic function may not. Some other published approaches assume six or more arguments implies variadic functions, resulting in false positives.

3.7.2 *Tracking Patterns of Argument Registers*

As one of the features of a variadic function, Detector should detect the registers which take arguments. The System V ABI uses rdi, rsi, rdx, rcx, r8, and r9 as argument registers in CDECL [61]. The argument registers can be from the following list in compilers GCC, Clang, and ICC in both Intel x64 and x86 architectures.

Argument register list = [rdi, rsi, rdx, rbx, rcx, r8, r9, r8d, r9d, edi, esi, edx, ecx, r10, r11, r12, r13, r14, r15]

Listing 3.2: Argument registers in data representation

```
1 inst(39538, 0x425eeb, movq, [reg(rdi),mem([offset(-0xd8),base(
    rbp)])]),
2 inst(39539, 0x425ef2, movq, [reg(rsi),mem([offset(-0xa8),base(
    rbp)])]),
3 inst(39540, 0x425ef9, movq, [reg(rdx),mem([offset(-0xa0),base(
    rbp)])]),
4 inst(39541, 0x425f00, movq, [reg(rcx),mem([offset(-0x98),base(
    rbp)])]),
5 inst(39542, 0x425f07, movq, [reg(r8),mem([offset(-0x90),base(rbp
    )])]),
6 inst(39543, 0x425f0e, movq, [reg(r9),mem([offset(-0x88),base(rbp
    )])]),
```

Listing 3.3: Patterns of argument registers

```
1 <Argument-Registers>:: = <movlist> <Source-Reg> <Destination-Reg
    > | <pushlist> <Source-Reg> <Destination-Reg>
2 <movlist>:: = <mov> | <movl> | <movq>
3 <pushlist>:: = <push> | <pushl> | <pushq>
4 <Source-Reg>:: = <rdi> |<rsi> | <rdx>| <rbx>  | <rcx> | <r8> | <
    r9> | <r8d> | <r9d> | <edi> | <esi> | <edx>| <ecx> | <r10> |
    <r11> | <r12> | <r13> | <r14> | <r15>
5 <Destination-Reg>:: = <offset> <rbp> | <offset> <rsp> | <offset>
     <ebp> | <offset> <esp>
```

Listing 3.2 demonstrates the patterns of argument registers in our data representation. These argument registers have been stored in the local variables section in the stack. Listing 3.2 is a piece of average() code from Listing 1.1 that has been compiled by GCC for the Intel x64 architecture.

Each compiler has its own syntax of argument registers in different levels of optimization. Listing 3.3 describes the general grammar of argument register patterns. The argument registers can push or move on the stack so both pushlist and movlist opcodes have been used in the grammar. The Source-Reg is all possible argument registers in the list. Destination-Reg shows that the argument registers can store on the stack with both stack pointer and base pointer.

If the type of the variadic arguments is float, then the variadic function follows the patterns which have been described in the next section.

Listing 3.4: Float-point argument registers in variadic function of benchmark XZ-R

```
1 inst(39421, 0x425c83, vmovaps, [reg(xmm0),mem([offset(-128),base
    (rbp)])]),
2 inst(39422, 0x425c88, vmovaps, [reg(xmm1),mem([offset(-112),base
    (rbp)])]),
3 inst(39423, 0x425c8d, vmovaps, [reg(xmm2),mem([offset(-96),base(
    rbp)])]),
4 inst(39424, 0x425c92, vmovaps, [reg(xmm3),mem([offset(-80),base(
    rbp)])]),
5 inst(39425, 0x425c97, vmovaps, [reg(xmm4),mem([offset(-64),base(
    rbp)])]),
6 inst(39426, 0x425c9c, vmovaps, [reg(xmm5),mem([offset(-48),base(
    rbp)])]),
7 inst(39427, 0x425ca1, vmovaps, [reg(xmm6),mem([offset(-32),base(
    rbp)])]),
8 inst(39428, 0x425ca6, vmovaps, [reg(xmm7),mem([offset(-16),base(
    rbp)])]),
```

Listing 3.5: Patterns of float-point argument registers

```
1 <Floiting-point-Arguments>:: = <vmovapslist> <xmm-Reg> <
    Destination-Reg>
2 <vmovapslist>:: = <movaps> | <vmovaps>
3 <xmm-Reg>:: = <xmm0><xmm1><xmm2><xmm3><xmm4><xmm5><xmm6><xmm7> |
    <xmm7><xmm6><xmm5><xmm4><xmm3><xmm2><xmm1><xmm0>
4 <Destination-Reg>:: = <offset> <rbp> | <offset> <rsp> | <offset>
    <rax>
```

3.7.3 *Tracking Patterns of Floating-point Arguments*

The floating-point arguments are passed by xmm registers. Listing 3.4 demonstrates the section of floating-point arguments in our data representation. This listing is from the code of one of the variadic functions in benchmark XZR that was compiled by GCC for the Intel x64 architecture.

Each compiler has its own syntax for using xmm registers to pass floating point arguments. Listing 3.5 describes the general syntax of floating-point arguments. The sequences of the xmm numbers are different in compilers GCC, Clang, and ICC. Both compilers GCC and Clang, use the base pointer and stack pointer for storing the floating-point arguments in local variables. The ICC uses the register rax for this task.

TABLE 3.4: Different last non-variadic pattern organization between GCC and Clang

GCC (va_start)	Clang (va_start)
40090e:lea 0x10(%rbp),%rax 400912:mov %rax,-0xc8(%rbp) 400919:lea -0xb0(%rbp),%rax 400920:mov %rax,-0xc0(%rbp)	4009c2:lea -0x20(%rbp),%r9 4009c6:mov %r8d,-0x4(%rbp) 4009ca:movl $0x0,-0x24(%rbp) 4009d1:lea -0xe0(%rbp),%r10 4009d8:mov %r10,0x10(%r9) 4009dc:lea 0x10(%rbp),%r10 4009e0:mov %r10,0x8(%r9)

3.7.4 *Tracking Patterns of last non-variadic argument*

Before taking the variadic arguments, the variadic function takes and stores the last non-variadic argument of the program through the va_start macro. The Detector follows the patterns of finding the address of the last non-variadic argument to increase the chance of identifying variadic functions in binary code. Same as the previous patterns, each compiler and optimization level has its own structure for taking and storing the last non-variadic argument. Table 3.4 shows the va_start section of the average() function from Section 1.1. In this piece of code, the last non-variadic is "num" parameter. The code was compiled by GCC for Intel x64 in the first column and by Clang for Intel x64 in the second column. In GCC, at address 40090e, the lea instruction finds the effective address of the last non-variadic argument. This is stored in rax, which is a temporary variable. Then at address 400912, the last non-variadic argument is stored in a local variable.

Clang has different organization for taking and storing the last non-variadic argument. First at address 4009c2, Clang stores an address of local variable in r9 and then Clang uses r9 as a reference to the va_start data structure. Again at addresses 4009d1 and 4009d8, Clang takes an address of local variable and stores it in r9 + 0x10. Finally, at address 4009dc, Clang finds the address of the last non-variadic argument and stores the address in r9 + 0x8.

TABLE 3.5: Argument registers save area offsets

0	rdi
0x8	rsi
0x10	rdx
0x18	rcx
0x20	r8
0x28	r9
0x30	xmm0
0x40	xmm1
0x48	xmm2
...	...

LISTING 3.6: Patterns of argument register offset

```
1 inst(39554, 0x425f41, movq, [segreg(fs), value(0x28),reg(rax)]),
2 inst(39555, 0x425f4a, movq, [reg(rax),mem([offset(-0xb8),base(
    rbp)])]),
3 inst(39556, 0x425f51, xorl, [reg(eax),reg(eax)])
```

3.7.5 *Tracking Patterns of Argument Registers offset*

As mentioned before, the argument registers take variadic arguments in the program. Based on the prologue of the function, the registers which take the arguments need to be saved in the register save area. Since the variadic arguments are taken by argument registers, Detector needs to know the offset of the register save area of the argument registers.

The register save area is space which allocates only those registers that need to be saved for a function [62]. The offset of the register save area is fixed for each argument register. Table 3.5 shows the argument registers save area offsets in variadic functions.

Listing 3.6 is an example of a variadic function. It demonstrates the patterns of the register save area for the argument register r9, since this register is stored at location 0x28 in the save area (see Table 3.5).

3.7.6 *Tracking Patterns of starting va_arg*

Relying just on detecting argument registers and float arguments is not sufficient to prove that the function is variadic. Because in the Intel x86 architecture using

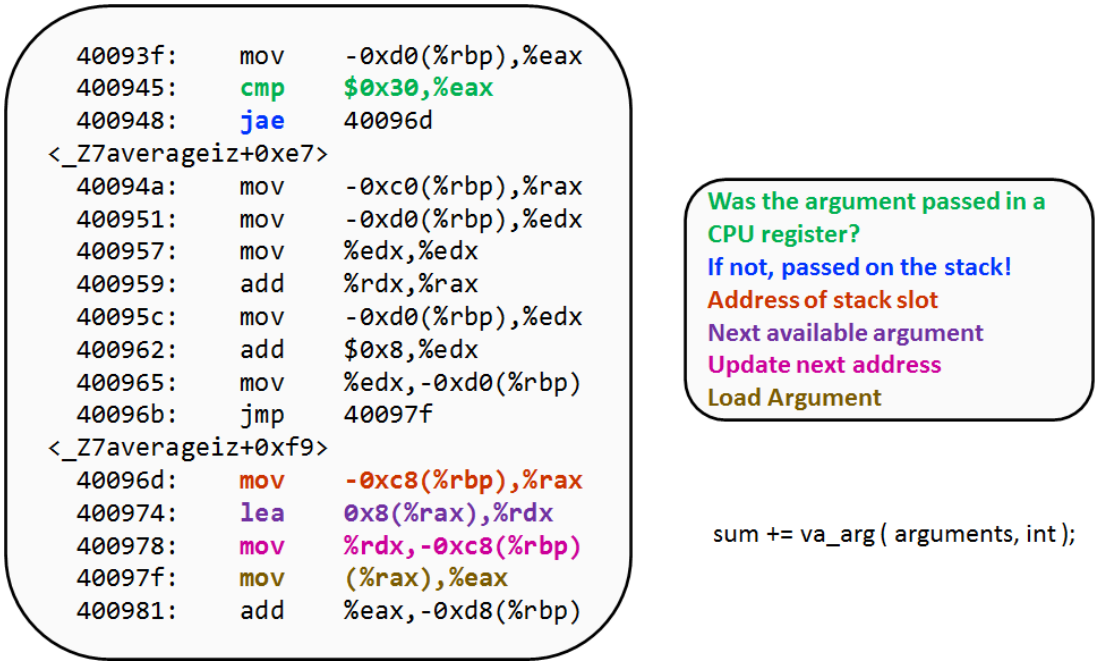


FIGURE 3.2: Patterns of va_arg in variadic function

the STDCALL calling convention, there are no argument registers. Based on the behavior of the variadic function, recognizing patterns of passing and location of the arguments is very important. The main task of va_arg is retrieving the variadic arguments from the va_list. Figure 3.2 shows tracking the arguments via va_arg. This binary code is for va_arg section of the average() function from Section 1.1. The code was compiled by GCC for Intel x64. Taking and storing the arguments with va_arg is different from taking and storing the arguments in non-variadic functions. At address 400945, register eax is compared with 0x30. The eax register is used to keep track of how many argument bytes have been used by va_arg. The first 0x30 bytes are assumed to be passed in registers and are stored in local variables. The rest of the arguments are passed on the stack, controlled by the caller.

Table 3.6 shows the relationship between location and the argument's number. In section 3.7.7 the location of the variadic arguments in Intel x64 and x86 has been described. Therefore, the 0x30 is the location of the seventh argument and this address is going to store in the CPU register. Because it was passed as a CPU register, the argument was stored in the local variable. Now, the va_arg starts walking on the stack to locate and update the local variables.

TABLE 3.6: Locations and the argument's number in va_arg

Location	Argument's number
0x0	First Argument
0x8	Second Argument
0x10	Third Argument
0x18	Fourth Argument
0x20	Fifth Argument
0x28	Sixth Argument
0x30	Seventh Argument

TABLE 3.7: Location of the variadic arguments in x86 and x64 architectures

Architectures	Function Parameters	Locations
X64	Just First six non-floating-point arguments and the first eight floating-point arguments	CPU Registers
X86	All of the variadic arguments	Stack

3.7.7 *Location of the variadic arguments in Intel x64 and x86*

Considering the location of the variadic arguments can be one of the methods to recognize the variadic functions. Detector needs to figure out the location of all the arguments. Depending on the ABI, the location of the variadic arguments is different in the Intel x86 and x64 architectures. Table 3.7 shows that the first six non-floating-point arguments and the first eight floating-point arguments in the Intel x64 architecture are passed to the registers and the remaining arguments are passed on the stack [50]. All variadic arguments in the Intel x86 must be passed on the stack.

3.8 SEMANTIC ANALYSIS

The syntactic analysis describes the set of patterns of the main sections of variadic functions. Detector uses semantic analysis which reveals the meaning of syntax. The semantic analysis allows Detector to recognize the valid paths of defined patterns to detect variadic functions with more accuracy. These paths demonstrate the behavior of the variadic function based on the type of compiler and architecture. By using semantic analysis, Detector has the capability to follow the valid paths of patterns for

TABLE 3.8: Parameters of the semantic analysis

Semantic Names	Patterns	Tasks
Arg-Reg	Argument Registers	Track the set of patterns Argument Registers
Float-Reg	Floating-point Arguments	Track the set of patterns Floating-point Arguments
LNV	Last non-variadic-Argument	Track the set of patterns of last non-variadic argument
FS	Offset of Argument Registers	Track the set of last saving register area of argument register
Va-Arg	Taking variadic arguments in va_arg section	Track the set of taking and locating argument in va_arg section
TAG	Type of Compiler and Architecture	Look at the low- level information such as memory size and integers of the first range of the function boundaries in the program

different compilers and architectures. Table 3.8 shows the names used in semantic analysis of detecting variadic functions.

Listing 3.7 shows the semantic logics for each compiler tag. In principle, $\bigwedge$ denotes AND operation and $\bigvee$ denotes OR operation. Each tag has its own organization of the patterns. The MV tag used for the GCC x64, first comes up with the argument register patterns. Then Detector looks for the floating-point argument patterns. After these two patterns, the MV can have an offset of the last argument register pattern or last non-variadic argument. For the CL tag used in Clang x64, first Detector follows the floating-point argument patterns. Then Detector tracks the argument register patterns. After these two patterns, CL can have the last non-variadic argument or offset of the last argument register patterns. For the IC tag used in ICC x64, first Detector should follow the argument register patterns. Then Detector tracks the floating-point argument patterns. After these two patterns, IC can have the last non-variadic argument patterns. The GS tag used in GCC x86, does not have argument register and floating-point register patterns. It just comes up with the last non-variadic argument and va_arg patterns.

Figure 3.3 demonstrates the organization of the patterns in each compiler. The first step is identifying the type of compiler and architecture. Based on the type of

Listing 3.7: Semantic logic for each compiler tag

```
1 MV -> (Arg-Reg) ⋀(Float-Reg)⋁ (FS)⋁ (LNV)
2 CL -> (Float-Reg) ⋀ (Arg-Reg) ⋁ (LNV) ⋁ (FS)
3 IC -> (Arg-Reg) ⋀ (Float-Reg)⋁(LNV)
4 GS -> (LNV) ⋀ (Va-Arg)
```

compiler and architecture, Detector tracks the patterns which have been defined for each tag. The green nodes demonstrate the patterns which should be detected in the data representation. In contrast, blue nodes may not exist in the data representation.

3.9 CAPTURING CALL SIDE OF VARIADIC FUNCTIONS

Function boundaries are beneficial for Detector since the functions are related to each other by the direct and indirect function calls. If Detector identifies a variadic function, then it can determine which functions have been affected by that variadic function. This feature gives Detector the capability to inform the developer to check which functions are calling variadic function and also, the variadic function is calling which functions in the stripped binary code. Knowing the functions which call variadic functions in the program directly or indirectly is very useful for detecting possible attacks. These functions push their parameters and return addresses on to the stack. The attackers can exploit the variadic functions by changing the number or type of arguments. Detecting vulnerabilities to these kind of attacks needs a dynamic analysis of the binary code. The program should be executed to recognize the changing of number and type of the arguments in variadic functions. In section 2.2, dynamic analysis methods and vulnerable variadic functions have been discussed.

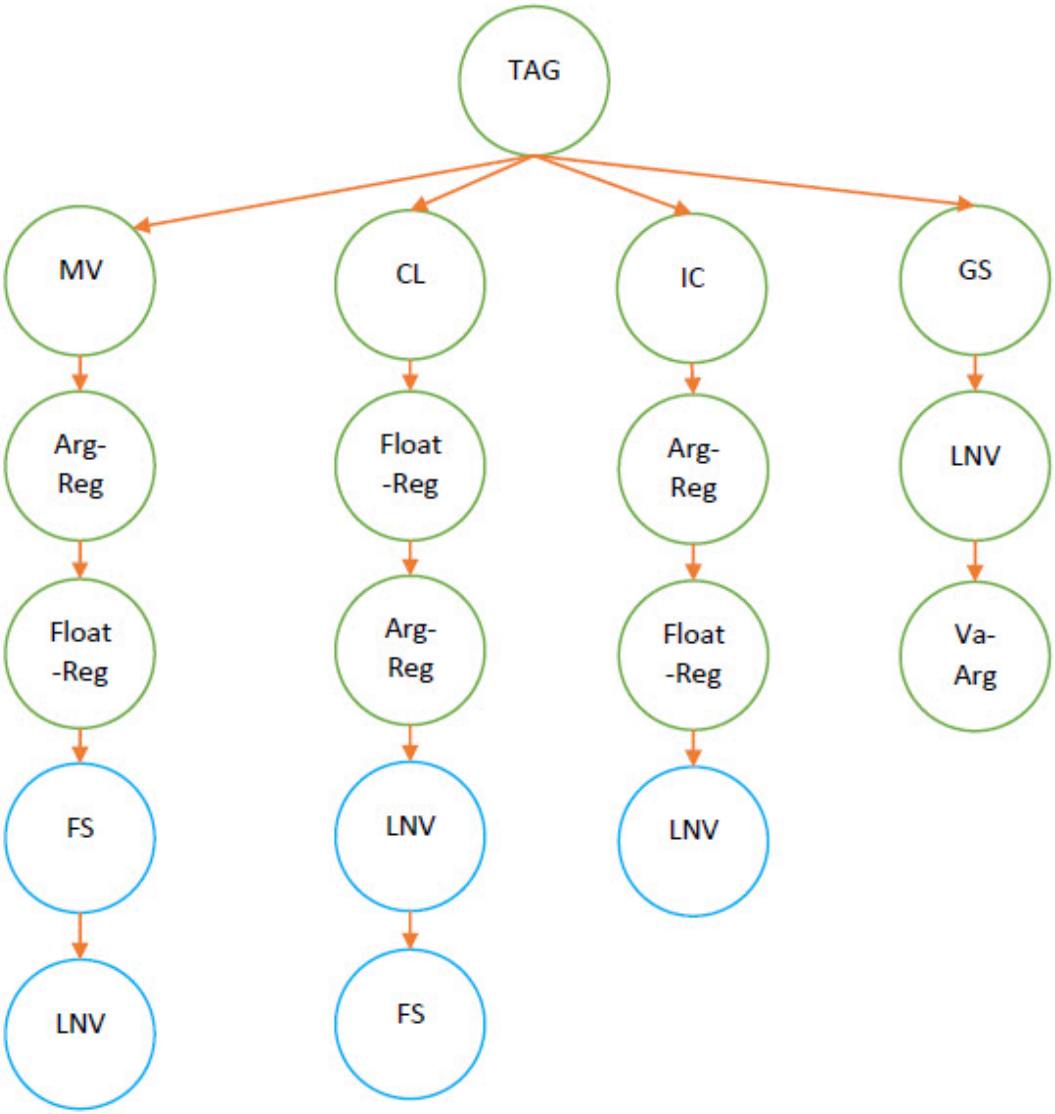


FIGURE 3.3: An overview of the steps of patterns organization for each tag of the compiler for detecting variadic functions

CHAPTER 4

Implementation Of Detector

This chapter summarizes the implementation of Detector which identifies variadic functions in stripped binary code. This stripped binary code can be compiled by one of the compilers GCC, Clang, and ICC in different levels (O0 to O3) on both Intel x86 and x64 architectures. Figure 4.1 demonstrates the inputs and outputs of Detector in a high-level design. The input files include only the stripped binary code and function boundaries generated by JIMA [18], which gives a list of possible function boundaries of the program. Detector uses these input files and detects variadic functions in the program automatically. The output of Detector is all possible variadic functions in the program and the call-side information for each detected variadic function.

4.1 ALGORITHM OF THE DETECTOR

Detector was written in Python3. The high level description of the algorithm used by Detector for detecting variadic functions is presented in Listing 4.1. Detector is implemented in different phases described in a high-level as follows. First, Detector uses stripped binary code and function boundaries as inputs. Based on the function boundaries, Detector selects the first function boundary range as a subset of instructions in the program. Detector identifies the tag of this subset of instructions based on the memory size and the prologue of the unknown function. Next, Detector looks at the valid path of variadic functions in all ranges of function boundaries by using semantic analysis to detect variadic functions in the whole program. If Detector recognizes variadic functions in the program, then for each of the variadic functions it calculates some useful information. This information includes the number of argument registers, the address of the last non-variadic argument and offset of the argument register. In addition to this information, Detector captures the call side information for each variadic function in the program.

Listing 4.1: Pseudocode of detecting variadic functions by Detector

```
get stripped binary code
map stripped binary code to instructions
get function boundaries

for the first range in function boundaries
  detect the tag of compiler

if the tag is MV then
  for all ranges in function boundaries
    get instruction subsets within instruction subset
  if patterns of MV is in subset then
    This function is variadic function

else if the tag is CL then
  for all ranges in function boundaries
    get instruction subsets within instruction subset
  if patterns of CL is in subset then
    This function is variadic function

else if the tag is IC then
  for all ranges in function boundaries
    get instruction subsets within instruction subset
  if patterns of IC is in subset then
    This function is variadic function

else if the tag is GS then
  for all ranges in function boundaries
    get instruction subsets within instruction subset
  if patterns of GS is in subset then
    This function is variadic function

if this subset is variadic function then
  retrieve information:
    address of last non variadic
    argument registers
           save- registers
  generate callsite for variadic function
```

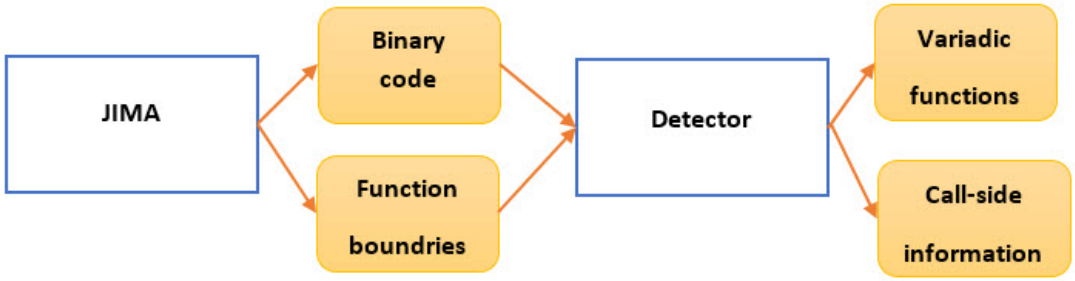


FIGURE 4.1: Implementation of Detector in High-Level Design

4.2 DIFFERENT PHASES OF DETECTOR OPERATION

The main phases of Detector are shown in Figure 4.2. First, the syntactic analysis phase determines the organization of the symbols and characters based on our data representation. The map processing phase uses syntax analysis to map all the lines in stripped binary code to our data representation instructions. Detector selects the first range from the function boundaries. Tag identification phase recognizes the type of compiler which compiled the program. Next, the semantic analysis phase determines which patterns should be recognized based on the tag of the compiler. The pattern matching phase uses semantic analysis rules to enable Detector to follow the valid path of patterns to identify the variadic function. Detector is able to show number and address range of variadic functions in the program. Capturing call side phase uses these address ranges and retrieves the caller of the variadic function.

4.3 OUTPUT OF THE DETECTOR

Not only can Detector detect variadic functions in the program, but also it can retrieve useful information to help programmers detect the vulnerable variadic functions in the future. Listing 4.2 shows the result of a variadic function compiled by GCC x64 architecture. The start address of this variadic function is 0x425c4b and the end address is 0x425d22. This variadic function uses six argument registers which include r9, edi, rsi, r8, rdx, and rcx. The offset is for the argument register r9. The memory address which includes 0x10 is for the last non-variadic argument in the program. The variadic function calls two functions with start addresses of 0x425c39

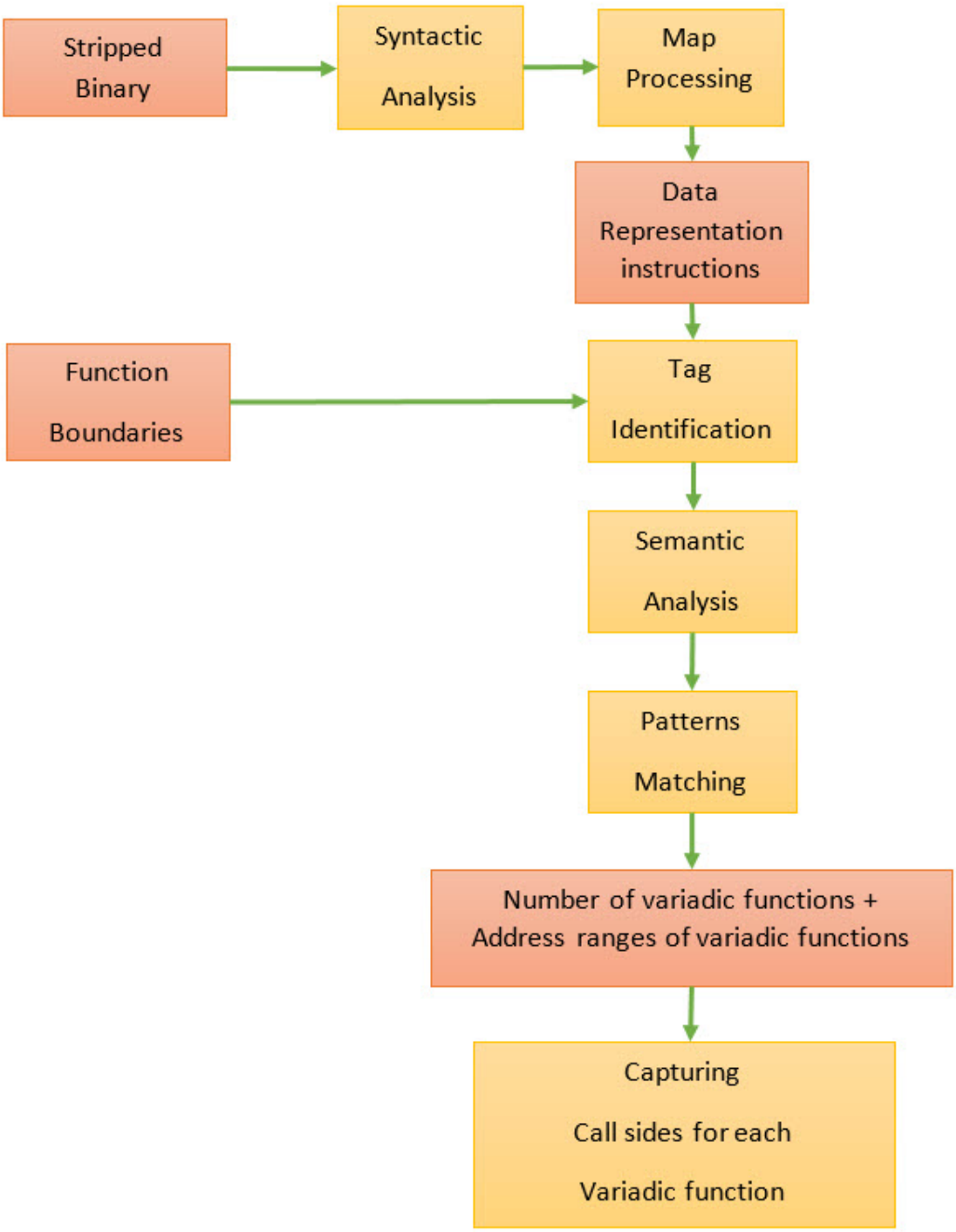


FIGURE 4.2: Different phases of the Detector Operation

Listing 4.2: Output of the Detector for a variadic function in benchmark XZR

```
1 0x425c4b ---> 0x425d22:
2 -------------
3 Number of arg registers: 6
4 Arg registers: {'r9', 'edi', 'rsi', 'r8', 'rdx', 'rcx'}
5 Offset register: ('fs', '0x28', 'r9')
6 Last non-variadic addr: 0x10
7 This VF calls: ['0x425c39', '0x400b10']
8 This VF is called by: [('0x421e29', 1), ('0x421e71', 4), ('0
     x426058', 2), ('0x42665b', 1), ('0x426fe8', 1)]
9 -------------
```

and 0x400b10. This variadic function is called by five functions in the program. The tuple used in the call side shows the number of times that the variadic function was called by other functions in the program. For example, ('0x421e71', 4) means that the function with the start address of 0x421e71 calls the variadic function in the program four times.

CHAPTER 5

EXPERIMENTAL RESULTS

This chapter details experimental results for detecting variadic functions in SPEC CPU 2006 and SPEC CPU 2017 benchmarks. These benchmarks have been chosen as a case study to execute and calculate the precision, recall, and F1 of Detector. These benchmarks consist of both variadic and non-variadic functions. We compared our experiments results with other tools, namely Type Armor [50] and Hexvasan [11].

- Type Armor can identify variadic functions in C and C++ programs that are compiled by GCC and Clang compilers. Type Armor does not have access to the source code but it needs to access the non-stripped binaries as requirement. The results of Type Armor analysis show that Type Armor has access to the symbol table because the name of the functions is available beside the function ranges. Type Armor can capture the number of arguments in the call site. It just relies on the argument register, not the float-point arguments in the program. Type Armor focuses on just detecting variadic functions in x64 architecture.

- Hexvasan identifies variadic functions in C and C++ programs that are compiled by Clang compiler. Hexvasan needs to access both source code and binary code to detect the variadic functions. Hexvasan has access to the source code so it can detect variadic functions which are defined in the headers. Some of these functions do not exist in the binary code since they are not executed in the program. Hexvasan can detect vulnerable variadic functions in the program. Hexvasan focuses on detecting just the variadic functions in x64 architecture.

As can be seen by this comparison, Detector has more capabilities than these available tools for detecting variadic functions. Detector can identify variadic functions in C and C++ programs that are compiled by GCC, Clang, and ICC. Detector does not need to access the source code or the symbol table, but uses JIMA for function boundaries [18]. Detector can detect variadic functions in both Intel x86 and x64 architectures.

Ground Truth The ground truth for each benchmark was determined by manually counting the number of variadic and non-variadic functions in the source code. Then, we calculated the precision, recall, and F1 by comparing results to the generated ground truth.

We compare the precision, recall, and F1 metrics of Detector compared with other tools. For calculating them, we need true negative, false positive, false negative, and finally true positive results of variadic function identification. These metrics for the tools are described as follows:

- **True Negative** (TN) means the function is a non-variadic function and the tool detects it as non-variadic function.
- **False Negative** (FN) means the function is a variadic function but the tool detects it as non-variadic function.
- **False Positive** (FP) means the function is a non-variadic function but the tool detects it as variadic function.
- **True Positive** (TP) means the function is a variadic function and the tool detects it as variadic function.

Precision (P): For comparing the tools with each other, we need to know when the tools have correct results. Precision means the percentage of reported variadic functions that are correctly reported, not false positive. The precision is the ratio of the true positive as the numerator and the sum of true positive and false positive results as the denominator.

$$\frac{|TP|}{(|TP| + |FP|)}$$

Recall (R): Recall denotes the percentage of true variadic functions detected by the tools. The recall is the ratio of the true positive as the numerator and and the sum of true positive and false negative results as the denominator.

$$\frac{|TP|}{(|TP| + |FN|)}$$

F1: F1 is a weighted average of precision and recall. F1 is calculated by the following formula. This is used to give a balanced comparison between tools, addressing both false positives and false negatives.

$$\frac{(2 * P * R)}{(P + R)}$$

5.1 SPEC CPU 2017

SPEC CPU 2017 [63] has been chosen to compare the results of the tools with each other. The benchmarks selected from SPEC CPU 2017 have both variadic and non-variadic functions in their programs. Detector was compared just with TypeArmor in SPEC CPU 2017. We downloaded TypeArmor and ran it for the selected benchmarks. For the number of variadic detections, we show that Detector works better than TypeArmor.

5.1.1 *Detecting Variadic Functions in GCC x64*

Detector was compared with TypeArmor for the number of detected variadic functions in the binary code compiled by GCC compiler and x64 architecture. Figure 5.1 demonstrates the differences in precision between Detector and TypeArmor. The blue columns show the precision of Detector and red columns show the precision of TypeArmor. Detector identifies some non-variadic functions as variadic functions since some non-variadic functions use argument registers to take arguments and store them on the stack. Some of the TypeArmor benchmarks do not have results because TypeArmor has segmentation fault errors when running them. As Figure 5.1 shows, Detector has more precision than TypeArmor. The precision average for Detector is 98.70% while for the TypeArmor it is 28.53%. This means TypeArmor mislabeled many non-variadic functions as being variadic.

Figure 5.2 demonstrates the differences in recall between Detector and TypeArmor. The blue columns show the recall of Detector and the red columns show the recall of TypeArmor. The averages of Detector and TypeArmor for the recall score are 100%. That means both of them find all variadic functions.

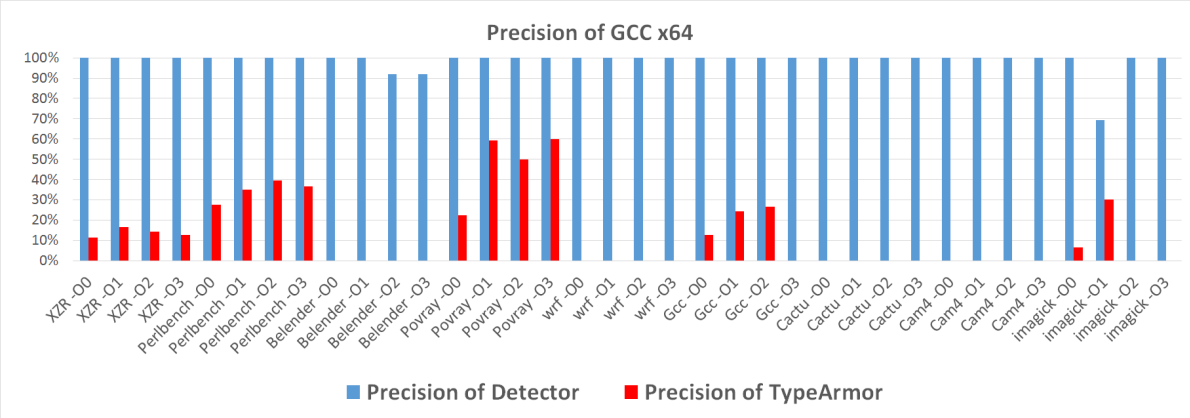


FIGURE 5.1: Precision of GCC x64 for comparing Detector and TypeArmor in SPEC CPU 2017

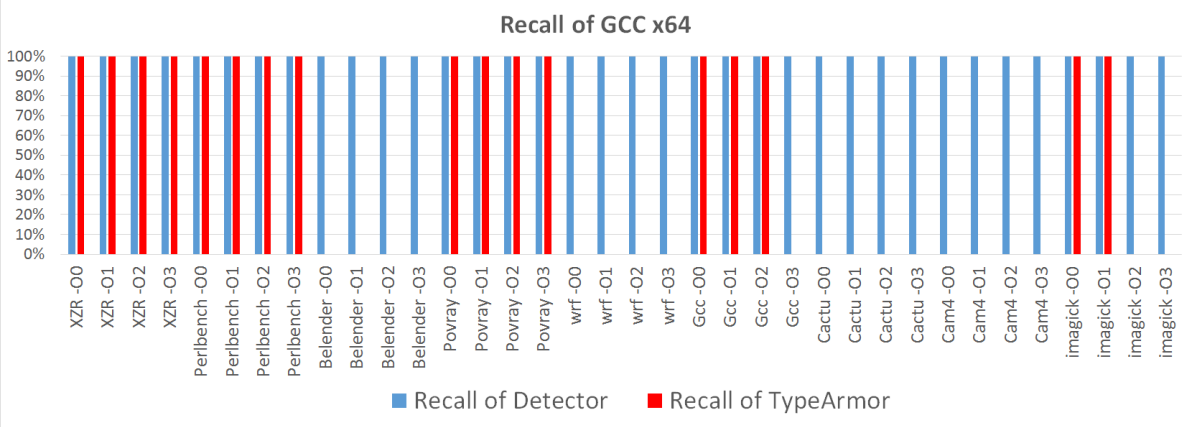


FIGURE 5.2: Recall of GCC x64 for comparing Detector and TypeArmor in SPEC CPU 2017

Figure 5.3 demonstrates the differences in F1 between Detector and TypeArmor. The blue columns show the F1 of the Detector and red columns show the F1 of the TypeArmor. Some benchmarks do not have results because TypeArmor has segmentation fault errors in them. As Figure 5.3 shows, Detector has a higher F1 score than TypeArmor. The average of F1 score for the Detector is 99.26% while for the TypeArmor it is 42.11%.

Table 5.1 summarizes the average of precision, recall, and F1 for both TypeArmor and Detector. Detector increases precision over TypeArmor by around 70.17%. That means Detector reports less non-variadic functions as variadic functions than TypeArmor. There is no difference between the average of recall in the TypeArmor

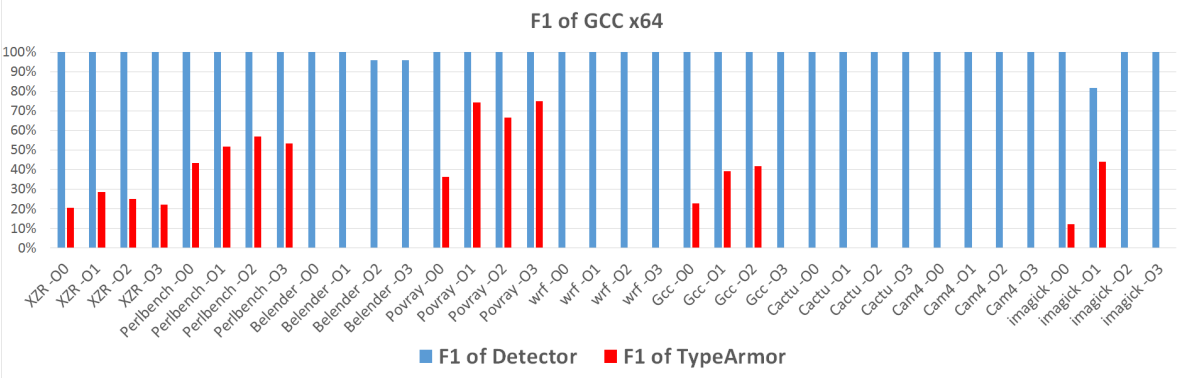


FIGURE 5.3: F1 of GCC x64 for comparing Detector and TypeArmor in SPEC CPU 2017

TABLE 5.1: The average of precision, recall, and F1 for both TypeArmor and the Detector in GCC x64

	Precision	Recall	F1
TypeArmor	28.53%	100.00%	42.11%
Detector	98.70%	100.00%	99.26%
Improvement	+70.17%	-	+57.15%

and Detector. Due to increased precision, Detector increases the average of F1 score around 57.15%.

Table 5.2 gives detailed precision, recall, and F1 score of Detector for detecting the variadic functions in code that is compiled by GCC x64 compiler.

5.1.2 *Detecting Variadic Functions in Clang x64*

Detector was compared with TypeArmor for the number of detected variadic functions in binary code compiled by the Clang compiler and x64 architecture. Figure 5.4 demonstrates the differences in precision between Detector and TypeArmor. The blue columns show the precision of Detector and red columns are for TypeArmor. Detector identifies some non-variadic functions as variadic functions for Imagick benchmark. Some TypeArmor benchmarks do not have results because TypeArmor has segmentation fault errors in them. As Figure 5.4 shows Detector has more precision than TypeArmor. The average precision of the Detector is 98.68% while for the TypeArmor it is 27.60%.

TABLE 5.2: The average precision, recall, and F1 for the Detector in GCC x64

SPEC 2017 Benchmarks	TN	TP	FN	FP	Precision	Recall	F1
XZR -O0	545	5	0	0	100.00%	100.00%	100.00%
XZR -O1	402	1	0	0	100.00%	100.00%	100.00%
XZR -O2	400	1	0	0	100.00%	100.00%	100.00%
XZR -O3	378	1	0	0	100.00%	100.00%	100.00%
Perlbench -O0	5125	31	0	0	100.00%	100.00%	100.00%
Perlbench -O1	2625	23	0	0	100.00%	100.00%	100.00%
Perlbench -O2	2531	23	0	0	100.00%	100.00%	100.00%
Perlbench -O3	2396	23	0	0	100.00%	100.00%	100.00%
Blender -O0	50719	26	0	0	100.00%	100.00%	100.00%
Blender -O1	39547	23	0	0	100.00%	100.00%	100.00%
Blender -O2	39758	23	0	2	92.00%	100.00%	95.83%
Blender -O3	39758	23	0	2	92.00%	100.00%	95.83%
Povray -O0	2105	16	0	0	100.00%	100.00%	100.00%
Povray -O1	1679	16	0	0	100.00%	100.00%	100.00%
Povray -O2	1689	15	0	0	100.00%	100.00%	100.00%
Povray -O3	1616	18	0	0	100.00%	100.00%	100.00%
Wrf -O0	7600	2	0	0	100.00%	100.00%	100.00%
Wrf -O1	7231	2	0	0	100.00%	100.00%	100.00%
Wrf -O2	7222	2	0	0	100.00%	100.00%	100.00%
Wrf -O3	7185	2	0	0	100.00%	100.00%	100.00%
Gcc -O0	26830	61	0	0	100.00%	100.00%	100.00%
Gcc -O1	13060	41	0	0	100.00%	100.00%	100.00%
Gcc -O2	12925	41	0	0	100.00%	100.00%	100.00%
Gcc -O3	12219	41	0	0	100.00%	100.00%	100.00%
Cactu -O0	3435	24	0	0	100.00%	100.00%	100.00%
Cactu -O1	2714	9	0	0	100.00%	100.00%	100.00%
Cactu -O2	2744	9	0	0	100.00%	100.00%	100.00%
Cactu -O3	2677	9	0	0	100.00%	100.00%	100.00%
Cam4 -O0	4574	3	0	0	100.00%	100.00%	100.00%
Cam4 -O1	4011	3	0	0	100.00%	100.00%	100.00%
Cam4 -O2	4019	3	0	0	100.00%	100.00%	100.00%
Cam4 -O3	3975	3	0	0	100.00%	100.00%	100.00%
Imagick -O0	2962	9	0	0	100.00%	100.00%	100.00%
Imagick -O1	2194	9	0	4	69.23%	100.00%	81.82%
Imagick -O2	2205	9	0	0	100.00%	100.00%	100.00%
Imagick -O3	2282	9	0	0	100.00%	100.00%	100.00%

Figure 5.5 demonstrates the differences in recall between Detector and TypeArmor. The blue columns show the recall of the Detector and the red columns show the recall of the TypeArmor. The recall average of TypeArmor is 100%, but it

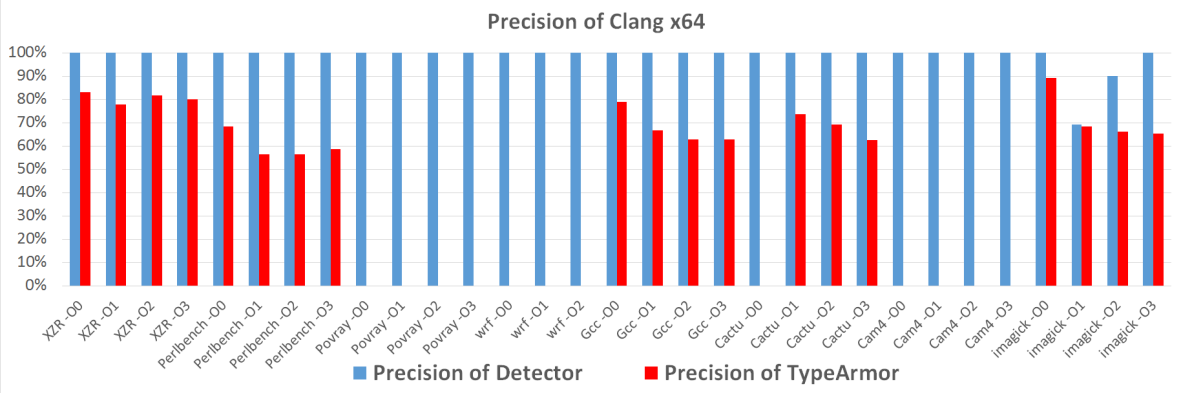


FIGURE 5.4: Precision of Clang x64 for comparing Detector and TypeArmor in SPEC CPU 2017

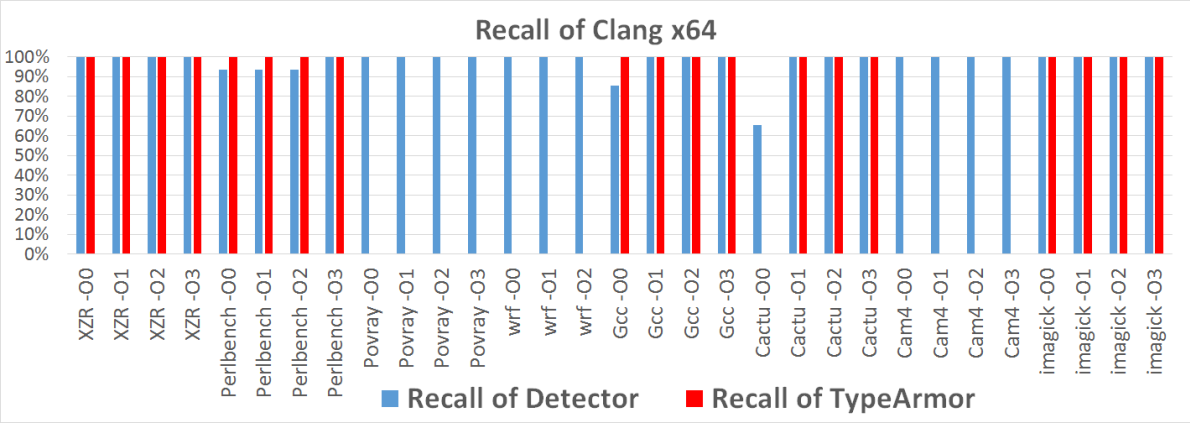


FIGURE 5.5: Recall of Clang x64 for comparing Detector and TypeArmor in SPEC CPU 2017

had segmentation fault for three benchmarks. The recall average of the Detector is 97.79% since it missed detection some variadic functions in Perlbench, Gcc, and Cactu benchmarks.

Figure 5.6 demonstrates the differences in F1 scores between Detector and TypeArmor. The blue columns show the F1 score of the Detector and red columns show the F1 of the TypeArmor. Some benchmarks do not have results because TypeArmor had segmentation fault errors when running them. As Figure 5.6 shows, Detector has a higher F1 score than TypeArmor. The F1 average of the Detector is 97.99% while for the TypeArmor it is 44.67%.

Table 5.3 summarizes the average of precision, recall, and F1 for both TypeArmor and Detector. Detector increases the precision over TypeArmor by around 71.08%

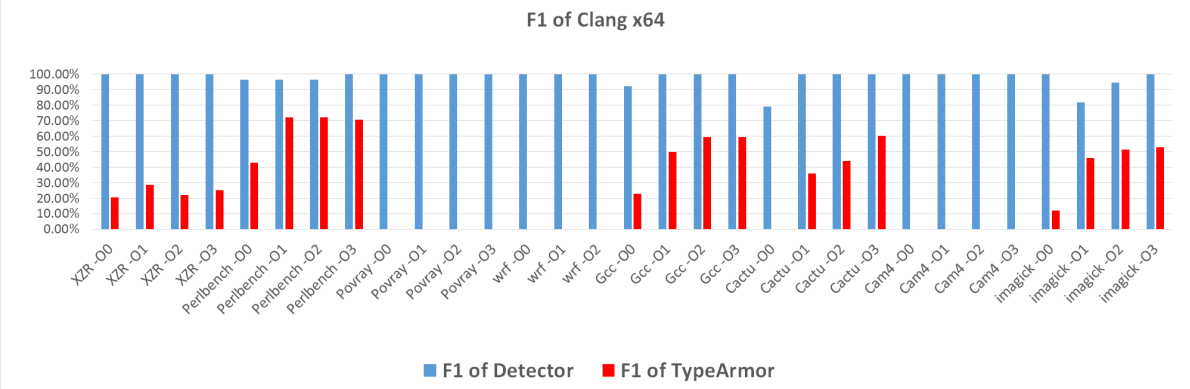


FIGURE 5.6: F1 of Clang x64 for comparing Detector and TypeArmor in SPEC CPU 2017

TABLE 5.3: The average of precision, recall, and F1 for both TypeArmor and Detector in Clang x64

	Precision	Recall	F1
TypeArmor	27.60%	100.00%	44.67%
Detector	98.68%	97.79%	97.99%
Improvement	+71.08%	-2.21%	+53.32%

but decreases the recall around 2.21%. That means the Detector missed some variadic functions but mislabled a lot less non-variadic functions when compared to TypeArmor. Detector increases the average of F1 score over TypeArmor by around 53.32%.

Table 5.4 demonstrates the precision, recall, and F1 score of Detector for detecting the variadic functions in code that is compiled by Clang x64 compiler.

5.1.3 *Detecting Variadic Functions in ICC x64*

Table 5.6 demonstrates the precision, recall, and F1 score for detecting the variadic functions in code that is compiled by ICC x64 compiler. TypeArmor and Hexvasan cannot detect the variadic functions which are compiled by ICC x64. Detector identifies all the variadic functions in the program. However, it also detects some non-variadic functions as variadic functions. Figure 5.7 demonstrates the precision, recall, and F1 score of Detector for detecting the variadic functions in code that is compiled by ICC x64 compiler.

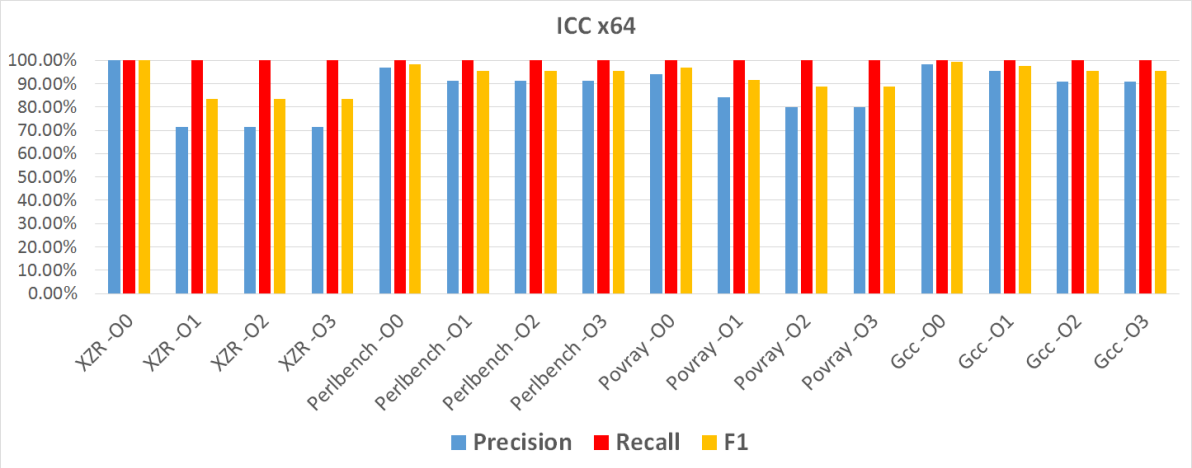


FIGURE 5.7: Precision, recall, and F1 score of ICC x64 in SPEC CPU 2017

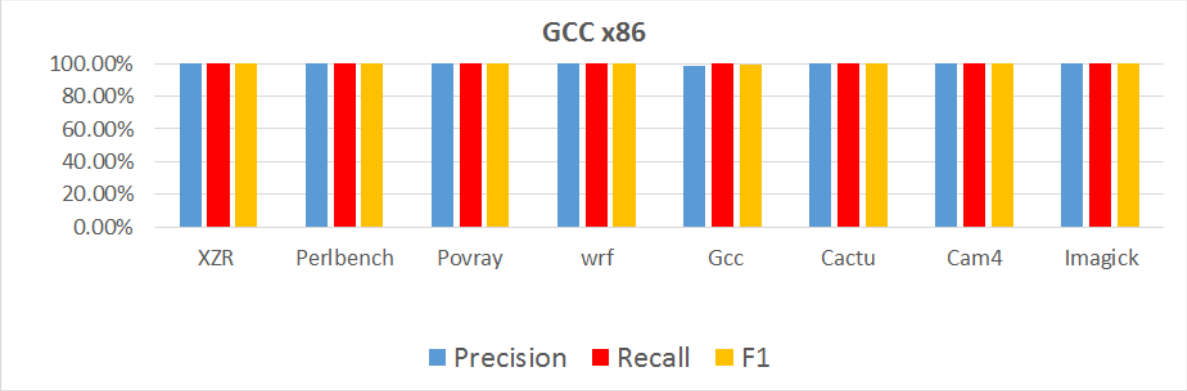


FIGURE 5.8: Precision, recall, and F1 score of GCC x86 in SPEC CPU 2017

Table 5.5 summarizes the average of precision, recall, and F1 for Detector. The average precision of Detector is around 87.43% and the recall average is 100%. The F1 score is around 93.29%.

5.1.4 *Detecting Variadic Functions in GCC x86*

Table 5.7 demonstrates the precision, recall, and F1 score for detecting the variadic functions in code that is compiled by GCC x86 compiler. TypeArmor and Hexvasan do not support the GCC x86 compiler for detecting variadic functions. Detector identifies all variadic functions in the program. However, it also detects some non-variadic functions as variadic functions. Figure 5.8 demonstrates the precision, recall, and F1 score of Detector for detecting the variadic functions in code that is compiled by GCC x86 compiler. The average precision of the Detector is around 99.80% and the recall average is 100%. The F1 score is around 99.90%.

5.2 SPEC CPU 2006

Hexvasan can identify variadic functions in programs compiled by the Clang x64 compiler. The authors of Hexvasan did not publish their tool completely, so we used their published results. Therefore, we compiled the SPEC CPU 2006 benchmarks for Detector and TypeArmor to compare with the results of Hexvasan. Since Hexvasan accesses source code, it detects some variadic functions in headers that are not in the executable files.

Detecting Variadic Functions in Clang x64: Table 5.8 demonstrates the number of variadic functions that the tools detect in the program. The total number of variadic functions column shows the exact number of variadic functions in the program. This table helps compare the tools with each other. For example, there is no variadic function in the benchmark astar. Detector detects one non-variadic function as a variadic function, but Hexvasan detects four non-variadic functions as variadic functions. The number of non-variadic functions detected by Detector is less than both TypeArmor and Hexvasan. The second example, there is no variadic function in the benchmark soplex. Detector does not detect any variadic functions in the program. TypeArmor detects seven non-variadic functions as variadic functions and Hexvasan detects two non-variadic functions as variadic functions.

TABLE 5.4: The average precision, recall, and F1 for the Detector in Clang x64

SPEC 2017 Benchmarks	TN	TP	FN	FP	Precision	Recall	F1
XZR -O0	545	5	0	0	100.00%	100.00%	100.00%
XZR -O1	547	1	0	0	100.00%	100.00%	100.00%
XZR -O2	373	1	0	0	100.00%	100.00%	100.00%
XZR -O3	374	1	0	0	100.00%	100.00%	100.00%
Perlbench -O0	3002	29	2	0	100.00%	93.55%	96.67%
Perlbench -O1	2995	29	2	0	100.00%	93.55%	96.67%
Perlbench -O2	2313	29	2	0	100.00%	93.55%	96.67%
Perlbench -O3	2299	29	0	0	100.00%	100.00%	100.00%
Povray -O0	2028	16	0	0	100.00%	100.00%	100.00%
Povray -O1	2023	16	0	0	100.00%	100.00%	100.00%
Povray -O2	1532	16	0	0	100.00%	100.00%	100.00%
Povray -O3	1525	16	0	0	100.00%	100.00%	100.00%
Wrf -O0	7592	2	0	0	100.00%	100.00%	100.00%
Wrf -O1	7406	2	0	0	100.00%	100.00%	100.00%
Wrf -O2	7224	2	0	0	100.00%	100.00%	100.00%
Gcc -O0	26677	53	9	0	100.00%	85.48%	92.17%
Gcc -O1	26553	62	0	0	100.00%	100.00%	100.00%
Gcc -O2	11797	61	0	0	100.00%	100.00%	100.00%
Gcc -O3	11756	61	0	0	100.00%	100.00%	100.00%
Cactu -O0	3435	17	9	0	100.00%	65.38%	79.07%
Cactu -O1	3262	26	0	0	100.00%	100.00%	100.00%
Cactu -O2	2596	25	0	0	100.00%	100.00%	100.00%
Cactu -O3	2588	25	0	0	100.00%	100.00%	100.00%
Cam4 -O0	4622	2	0	0	100.00%	100.00%	100.00%
Cam4 -O1	4321	2	0	0	100.00%	100.00%	100.00%
Cam4 -O2	4002	2	0	0	100.00%	100.00%	100.00%
Cam4 -O3	4040	2	0	0	100.00%	100.00%	100.00%
Imagick -O0	2962	9	0	0	100.00%	100.00%	100.00%
Imagick -O1	2194	9	0	4	69.23%	100.00%	81.82%
Imagick -O2	2205	9	0	1	90.00%	100.00%	81.82%
Imagick -O3	2282	9	0	0	100.00%	100.00%	100.00%

TABLE 5.5: The average of precision, recall, and F1 for the Detector in ICC x64

	Precision	Recall	F1
Detector	87.43%	100.00%	93.29%

TABLE 5.6: The average precision, recall, and F1 for the Detector in ICC x64

SPEC 2017 Benchmarks	TN	TP	FN	FP	Precision	Recall	F1
XZR -O0	544	5	0	0	100.00%	100.00%	100.00%
XZR -O1	436	5	0	2	71.43%	100.00%	83.33%
XZR -O2	407	5	0	2	71.43%	100.00%	83.33%
XZR -O3	407	5	0	2	71.43%	100.00%	83.33%
Perlbench -O0	3113	31	0	1	96.88%	100.00%	98.41%
Perlbench -O1	2854	31	0	3	91.18%	100.00%	95.38%
Perlbench -O2	2654	31	0	3	91.18%	100.00%	95.38%
Perlbench -O3	2654	31	0	3	91.18%	100.00%	95.38%
Povray -O0	2250	16	0	1	94.12%	100.00%	96.97%
Povray -O1	1917	16	0	3	84.21%	100.00%	91.43%
Povray -O2	1883	16	0	4	80.00%	100.00%	88.89%
Povray -O3	1883	16	0	4	80.00%	100.00%	88.89%
Gcc -O0	26768	61	0	1	98.39%	100.00%	99.19%
Gcc -O1	14864	61	0	3	95.31%	100.00%	97.60%
Gcc -O2	12512	61	0	6	91.04%	100.00%	95.31%
Gcc -O3	12512	61	0	6	91.04%	100.00%	95.31%

TABLE 5.7: The average precision, recall, and F1 for Detector in GCC x86

SPEC 2017 Benchmarks	TN	TP	FN	FP	Precision	Recall	F1
XZR	549	5	0	0	100.00%	100.00%	100.00%
Perlbench	5111	31	0	0	100.00%	100.00%	100.00%
Povray	2026	16	0	0	100.00%	100.00%	100.00%
wrf	6400	2	0	0	100.00%	100.00%	100.00%
Gcc	26893	61	0	1	98.39%	100.00%	99.19%
Cactu	3279	25	0	0	100.00%	100.00%	100.00%
Cam4	4576	2	0	0	100.00%	100.00%	100.00%
Imagick	2964	9	0	0	100.00%	100.00%	100.00%

TABLE 5.8: Differences between Detector, TypeArmor, and Hexvasan in Clang x64

SPEC 2006 Benchmarks	Detector	TypeArmor	Hexvasan	Total Number of Variadic Functions
astar	1	0	4	0
soplex	0	7	2	0
sjeng	2	0	4	0
omnetpp	20	21	48	20
namd	0	0	24	0
milc	1	3	21	0
mcf	0	0	3	0
libquantum	2	2	91	2
lbm	0	0	3	0
hmmer	2	13	9	2
h264ref	2	16	85	2
gobmk	14	31	35	14
bzip2	1	0	3	0

CHAPTER 6

Conclusions and Future Works

This chapter concludes the findings of this dissertation and highlights future work. Our research goal was to develop a theory and prototype automatic tool called Detector for identifying variadic functions in stripped binary code of C and C++ programs. Detector uses syntactic analysis and semantic analysis to recognize the behavior of the variadic functions in the binary code. We used SPEC CPU 2017 and 2006 benchmarks to evaluate Detector compared with other existing tools. By using measures of precision, recall, and F1, we demonstrate that Detector works better than the other tools.

6.1 RESEARCH CONTRIBUTIONS

Detecting variadic functions in binary code without existing source code and high-level information is very difficult. We began our research by reviewing the existing binary analysis tools to be aware of methods for analyzing stripped binary code. We designed Detector to statically recognize variadic functions in stripped binary code. That means Detector does not need to execute the program to check the variadic functions, so it considers all the paths in the program.

By reviewing existing variadic function detection tools for binary code, we choose to detect variadic functions in the program with behavior analysis. Detector analyzes the behavior of the functions to check whether they are variadic functions or not. The behavior of the variadic functions can be recognized by tracking some patterns of instructions. These patterns define the main tasks of the variadic functions that are taking the variable number of arguments and store them on the stack for the caller.

We defined new abstract data representation which lets Detector analyze the stripped binary code easier. Detector applies syntactic analysis and map processing which converts all lines of the binary code to our data representation. Detector uses

semantic analysis to find valid paths of variadic function patterns in our abstract data representation.

We choose code compiled by GCC, Clang, and ICC compilers as the input to Detector. Since each compiler has its own organization for compiling the source code, the patterns of variadic functions are different in these compilers. Detector assigns a tag for each compiler and defines the path of patterns based on the tag of the compiler. Detector uses tag identification for recognizing the type of compiler and architecture. Finally, Detector was compared with two available existing tools, Type Armor and Hexvasan. We evaluated the number of variadic functions that are detected by these tools.

For C and C++ source code that was compiled by GCC x64, Detector was only compared with Type Armor since Hexvasan does not accept GCC compiled code. The results show that Detector has more accuracy than the Type Armor for recognizing their variadic functions, with an F1 score of 99.26% compared to 42.11%.

For C and C++ source code that was compiled by Clang x64, Detector was compared with Type Armor and Hexvasan in both SPEC CPU 2006 and 2017. The differences between the number of variadic functions of Detector and the real number of them in the code was less than the other tools, with an F1 score of 97.99% compared to 44.67%.

For C and C++ source code that was compiled by ICC x64, Detector is the novel tool which works with this compiler. Both Type Armor and Hexvasan cannot support the compiled code by ICC x64. Therefore, we compared the results of the Detector with the exact number of the variadic functions in the source code. The F1 score was 93.29%.

Again for C and C++ source code that were compiled by GCC x86, Detector is the novel tool that works with this compiler. Both Type Armor and Hexvasan cannot support the compiled code by GCC x86. Therefore, we compared the results of the Detector with the exact number of the variadic functions in the source code. The F1 score was 99.90%.

6.2 FUTURE WORKS

Static and dynamic binary analysis techniques could be combined for detecting vulnerable variadic functions without accessing the source code. Currently, Detector can identify variadic functions in binary code which are compiled by GCC, Clang, and ICC. Detecting the most number of variadic functions in stripped binary code allows the security team to investigate vulnerable variadic functions with more accuracy. It also sets the stage for automated analysis and repair. The following are some future works.

- Ideally, one of the future goals can be designing an algorithm that shows the list of arguments with their types. This algorithm can be designed by data flow analysis methods to give us the view of the contract between callee and caller. The contract between the caller and callee for passing the arguments is implicit, so misusing the variadic function is an attractive case for attackers. The attacker can take advantage of this situation and change the number and type of arguments. This algorithm can help developers to improve the security of their programs by understanding the number and type of arguments.

- Can we detect if variadic function uses a good boundary check? Variadic functions are flexible. They allow the caller to pass an unbounded number of arguments. Designing a boundary check does not let the functions pass more arguments than the caller wanted. This boundary check makes tolerance for the number of arguments that are passed between the caller and callee. This boundary check needs to be designed with dynamic analysis methods since the number of arguments should be checked at run time.

- Can we use call-site information to find the maximum number of arguments? Calculating the maximum number of arguments helps to know the locations where the variadic function's arguments have been filled on the stack. This area of the arguments and local variables in the stack should be protective of the attacker. Since the attacker can read sensitive data from the stack, read data from anywhere in memory or overwrite function pointers, return addresses,

and counters. Protecting this location of the stack from the attacker lets variadic functions send the correct arguments to the other functions.

- If we know it is variadic function, can we pass a new argument as the number of arguments? Now Detector can find the variadic functions in the programs. If the number of arguments calculates and reports to the programmer, then the programmer can follow any sign of a changing number of arguments in the programs. Some of the variadic functions have been defined by one non-variadic argument to show the number of variadic arguments. The programmer can compare the reported number of arguments with this non-variadic argument.

- How we can prevent the attack in vulnerable variadic functions? There are different types of vulnerabilities for variadic functions such as buffer overflow, format string attack, or mismatching the number of arguments. The buffer overflow can happen if the attacker can overwrite extra data on the stack. Another common vulnerability is a format string attack, which can happen mostly with printf. As we know, the first argument of printf is the format string. If the attacker accesses to this argument, which is used to define remaining arguments, then it is called format string attack. Finally, mismatching vulnerability is the difference of the number and type of the arguments between caller and callee. The goal is to design a tool to patch these types of vulnerabilities in variadic functions at runtime. After detecting the vulnerable variadic functions, the tool can report the type of vulnerability and provide the proper security mechanisms.

Patching vulnerable variadic functions Variadic functions are attractive for attackers since the contract between the caller and callee is implicit. Therefore, attackers can change the type or number of arguments in the variadic functions. Future work involves automatically patching vulnerable detected variadic functions.

- **Predicting Vulnerable variadic functions:** Predicting Vulnerable variadic functions by using dynamic analysis methods such as reaching definition analysis and liveness analysis. Reaching definition analysis determines who can assign

the value to the variable or memory location, while Liveness analysis lets us have backward view for following the variables. Through this analysis, we can detect the source of values to see if they are bounded by the code or not.

- **Format string attack:** The vulnerability of changing the type of arguments refers to a format string attack. Finding solutions for detecting format string attacks in binary code is an important security issues. This attack gives the attacker the capability to manage and control the program. It can cause changing the functionality or behavior of the program. Changing the type of the arguments can be prevented by checking the characters between the arguments and the output of the function.

- **Mismatching the number of arguments:** This type of attack can be detected by counting the byte of arguments in different compilers. Designing the counter for checking Check the number of arguments passed and used in the program. The number of arguments can be calculated from the memory address of the first argument to the last memory address of the local variable.

Bibliography

[1] D. Gregor, J. Järvi, and G. Powell, "Variadic templates (revision 3)," 2006.

[2] B. C. d. S. Oliveira and J. Gibbons, "Typecase: A design pattern for type-indexed functions," in *Proc. 2005 ACM SIGPLAN workshop on Haskell*, pp. 98–109, 2005.

[3] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "SoK: sanitizing for security," in *IEEE Symposium on Security and Privacy*, pp. 1275–1295, 2019.

[4] F. Spoto and G. Levi, "Abstract interpretation of prolog programs," in *International Conference on Algebraic Methodology and Software Technology*, pp. 455–470, Springer, 1999.

[5] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "Formatguard: Automatic protection from printf format string vulnerabilities," in *USENIX Security Symposium*, 2001.

[6] B. Spengler, "Pax: The guaranteed end of arbitrary code execution," *G-Con2*, 2003.

[7] R. Ma, L. Chen, C. Hu, J. Xue, and X. Zhao, "A dynamic detection method to C/C++ programs memory vulnerabilities based on pointer analysis," in *2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing*, pp. 52–57, 2013.

[8] "Format string attack."

[9] M. F. Ringenburg and D. Grossman, "Preventing format-string attacks via automatic and efficient dynamic checking," in *Proc. 12th ACM conference on Computer and communications security*, pp. 354–363, 2005.

[10] T. Tsai and N. Singh, "Libsafe 2.0: Detection of format string vulnerability exploits," *white paper, Avaya Labs*, 2001.

[11] P. Biswas, A. Di Federico, S. A. Carr, P. Rajasekaran, S. Volckaert, Y. Na, M. Franz, and M. Payer, "Venerable variadic vulnerabilities vanquished," in *26th USENIX Security Symposium*, pp. 186–198, 2017.

[12] W. Li and T. Chiueh, "Automated format string attack prevention for win32/x86 binaries," in *Twenty-Third Annual Computer Security Applications Conference*, pp. 398–409, 2007.

[13]

[14] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[15] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 213–223, IEEE, 2011.

[16] R. Qiao, *Accurate Recovery of Functions in COTS Binaries*. PhD thesis, The Graduate School, Stony Brook University: Stony Brook, NY, 2017.

[17] W. Ying, L.-z. GU, Z.-x. LI, and Y.-x. YANG, "Protocol reverse engineering through dynamic and static binary analysis," *The Journal of China Universities of Posts and Telecommunications*, vol. 20, pp. 75–79, 2013.

[18] J. Alves-Foss and J. Song, "Function boundary detection in stripped binaries," in *Proc. 35th Annual Computer Security Applications Conference*, pp. 84–96, 2019.

[19] B. Dang, A. Gazet, and E. Bachaalany, *Practical reverse engineering: x86, x64, ARM, Windows kernel, reversing tools, and obfuscation*. 2014.

[20] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, no. 12, pp. 92–106, 2004.

[21] M. Dahm, J. Van Zyl, and E. Haase, "Byte code engineering library," 2002.

[22] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 89–100, 2007.

[23] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pp. 175–183, IEEE, 2010.

[24] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pp. 309–318, 2012.

[25] P. P. Bungale and C.-K. Luk, "Pinos: a programmable framework for whole-system dynamic instrumentation," in *Proc. 3rd international conference on Virtual execution environments*, pp. 137–147, 2007.

[26] Z. Deng, X. Zhang, and D. Xu, "Spider: Stealthy binary program instrumentation and debugging via hardware virtualization," in *Proc. 29th Annual Computer Security Applications Conference*, pp. 289–298, 2013.

[27] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. 2013 ACM SIGSAC conference on Computer & communications security*, pp. 499–510, 2013.

[28] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *International Conference on Information Systems Security*, pp. 1–25, Springer, 2008.

[29] B. Zeng, G. Tan, and G. Morrisett, "Combining control-flow integrity and static analysis for efficient and validated data sandboxing," in *Proc. 18th ACM conference on Computer and communications security*, pp. 29–40, 2011.

[30] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *International Conference on Computer Aided Verification*, pp. 463–469, Springer, 2011.

[31] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani, "Darwin: An approach to debugging evolving programs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 21, no. 3, pp. 1–29, 2012.

[32] D. Qi, H. D. Nguyen, and A. Roychoudhury, "Path exploration based on symbolic output," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, pp. 1–41, 2013.

[33] M. Dam, R. Guanciale, N. Khakpour, H. Nemati, and O. Schwarz, "Formal verification of information flow security for a simple arm-based separation kernel," in *Proc. 2013 ACM SIGSAC conference on Computer & communications security*, pp. 223–234, 2013.

[34] M. Eddington, "Peach fuzzing platform," 2011.

[35] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, pp. 497–512, IEEE, 2010.

[36] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[37] C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *2011 33rd International Conference on Software Engineering (ICSE)*, pp. 1066–1071, IEEE, 2011.

[38] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX security symposium*, vol. 98, pp. 63–78, 1998.

[39] P. Team, "Pax non-executable pages design & implementation," *Avaliable: http://pax. grsecurity. net*, 2003.

[40] J. Mcdonald, *Art of Software Security Assessment*. Pearson Professional Education, 2006.

[41] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[42] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "BYTEWEIGHT: Learning to recognize functions in binary code," in *23rd USENIX Security Symposium*, pp. 845–860, 2014.

[43] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th USENIX Security Symposium*, pp. 611–626, 2015.

[44] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 177–189, 2017.

[45] VUSec, "Compiler-agnostic function detection in binaries," Jun 2020.

[46] G. Ravipati, A. R. Bernat, N. Rosenblum, B. P. Miller, and J. K. Hollingsworth, "Toward the deconstruction of dyninst," *Univ. of Wisconsin, technical report*, p. 32, 2007.

[47] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi, "Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code," *Digital Investigation*, vol. 12, pp. S61–S71, 2015.

[48] C. Wilson and L. J. Osterweil, "Omega—a data flow analysis tool for the c programming language," *IEEE Transactions on Software Engineering*, no. 9, pp. 832–838, 1985.

[49] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis," 2009.

[50] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *IEEE Symposium on Security and Privacy*, pp. 934–953, 2016.

[51] Hex Rays SA, "IDA Pro disassembler and debugger," 2019.

[52] E. Cole, "Static taint analysis of binary executables using architecture-neutral intermediate representation," tech. rep., All Computer Science and Engineering Research, 2019.

[53] C. Jeon and Y. Cho, "A robust steganography-based software watermarking," in *Proc. 2012 ACM Research in Applied Computation Symposium*, pp. 333–337, 2012.

[54] M. Barr, *Programming embedded systems in C and C++*. O'Reilly Media, 1999.

[55] A. Fog, *Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms*. Technical University of Denmark, 2020.

[56] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers, principles, techniques*. Addison wesley, 1986.

[57] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proc. 8th ACM European Conference on Computer Systems*, pp. 295–308, 2013.

[58] K. Slonneger and B. L. Kurtz, *Formal syntax and semantics of programming languages*. 1995.

[59] S. Maruyama, Y. Tanaka, H. Sakamoto, and M. Takeda, *Context-sensitive grammar transform: Compression and pattern matching*. 2008.

[60] S. Alrabaee, N. Saleem, S. Preda, L. Wang, and M. Debbabi, "Oba2: An onion approach to binary code authorship attribution," *Digital Investigation*, vol. 11, pp. S94–S103, 2014.

[61] H. A. Jelodar, *Deep Learning in Attention Networks*. PhD thesis, State University of New York at Stony Brook, 2017.

[62] D. R. Ditzel and H. R. McLellan, "Register allocation for free: The C machine stack cache," in *Proc. First International Symposium on Architectural Support for Programming Languages and Operating Systems*, pp. 48–56, 1982.

[63] Standard Performance Evaluation Corporation, "SPEC CPU® 2017."