Static Analysis of Stripped Binary Executables to Find Function Parameters, Local Variables and Parameters Used as Pointers in Intel 32-bit and 64-bit Architectures

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Rashmi Shrivastava

Approved by:

Major Professor: Jim Alves-Foss, Ph.D.

Committee Members: Jim Alves-Foss, Ph.D.; Daniel Conte de Leon, Ph.D.; Jia Song, Ph.D.

Department Administrator: Terence Soule, Ph.D.

May 2023

Abstract

With the rapid increase and complexity of cyber-attacks, there is a need to analyze and understand software programs, not only the source code, but also the binary executable. We might encounter situations where source code and debugging symbols are not available, thus we need to analyze low-level executables. The variable types, function parameters and indirect memory access provide the fundamental semantics of a program. Generally, when we compile an executable, the critical information related to variables, types and parameters is lost. This leads to conservative static analysis at the binary-level. A large amount of research has been carried out for decades on binary code type inference, a challenging task that aims to infer typed variables [1].

To improve our ability to perform static and dynamic analysis, the goal of this thesis is to develop a novel algorithm for finding the parameters of a function, the local variables of a function, and the parameter used as a pointer. The purpose of this algorithm is to explore applications in the context of program understanding and to provide useful information about function parameters, even in the absence of debugging information [2]. The approach is not fully sound, which means that there could be false positives or false negatives. The designed algorithm is a step towards determining critical information of any function. The algorithm has been tested over 444 functions in total for GCC and 200 for Clang from stripped binaries compiled on both Intel 32-bit and 64-bit with both O0 and O2 optimizations levels with a success rate of approx. 100% for finding the parameters, 78% and 66% for local variables and 85.5% and 77% for finding parameters as pointers in both GCC and Clang, respectively. We developed it on both 32-bit and 64-bit with O0 and O2 optimization levels. Based on the current development and analysis, we conclude with suggestions for future work and provide some insight into some preliminary ideas to solve such problems.

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 toward my master's would not be possible without him. His analytical mind and insights always inspired me, especially during the time I faced difficulties and setbacks. I greatly appreciate his constant support, efforts, and patience in my research.

Second, I would like to thank my committee members, Dr. Jim Alves-Foss, Dr. Daniel Conte de Leon, and Dr. Jia Song, for their valuable comments on my thesis. I greatly appreciate their time and support in this research.

I would like to thank all my instructors Dr. Jim Alves-Foss, Dr. Min Xian, Dr. Stephen Lee, Dr. Marshall Ma, Dr. Hasan 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, Ms. Arvilla Daffin, 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 research assistant during my master’s tenure. It gave me opportunities to work on interesting problems and improve my technical skills.

I would like to thank all the staff of the College of Graduate Studies for their help and support throughout my study at UofI.

I would like to thank my lab mates for providing encouragement, entertainment, and sometimes just an attentive ear to whatever problem I was facing. I would like to thank many other friends who made my life at UI very enjoyable.

Last, but certainly not least, I wish to acknowledge the Schweitzer Engineering Laboratories (SEL) for supporting me during the course of my graduate studies.

Dedication

To my beloved husband,

Thank you for your unwavering love, understanding, and patience throughout this process. Your constant encouragement and support have been the wind beneath my wings, pushing me to soar higher.

To my dear families,

Your love, prayers, and constant support have been invaluable to me. I appreciate your belief in me, even when I doubted myself. Your unwavering support has been a source of strength for me, and I am humbled to have such amazing people in my life.

Table of Contents

List of Tables

List of Figures

Chapter 1 : Introduction

With the rapid increase and complexity of cyber-attacks, there is a need to analyze and understand software programs, not only the source code, but also the binary executable. We might encounter situations where source code and debugging symbols are not available, thus we need to analyze low-level executable. The variable types, function parameters and indirect memory access provide the fundamental semantics of a program. Generally, when we compile an executable, the critical information related to variables, types and parameters is lost. This leads to conservative static analysis at the binary-level. A large amount of research has been carried out for decades on binary code type inference, a challenging task that aims to infer typed variables [1].

Source-level vulnerability analysis is only possible when we have source code available. As a part of development, often new products are built by using third-party libraries. This may lead to security vulnerabilities as the integration of third-party software does not always come with source code and developers may not fully understand the integrated software. Also, scaling up such products normally leads to threats because full analysis cannot be performed on the complex systems. Further, the non-availability of source code also hampers other analysis paradigms, such as fuzzing and symbolic execution, because even these techniques benefit from the ability to compile, rather than retrofit, instrumentation into the analysis target [10]. Although many software vulnerability analysis tools are out there in the market, most of them dealt with the analysis of source code written in a high-level language.

Over the past decades, efforts have been made towards static analysis of a binary code [4, 5, 6] to develop tools to find bugs and security vulnerabilities. Many techniques have been proposed to improve the recovery of data types [10, 11, 12], code structure [10, 13, 14, 15] and even exact syntactic identity [10, 16]. Usually, static analysis is done without any execution. This could be on source code, intermediate representations (IR), assembly code, or even binary code [3]. From the perspective of cyber security binary analysis is one of the reliable method to recover the lost variables, inference type and understand the control flow within a function in detail. There are two steps in static binary analysis: disassembly of the binary code and static

analysis of the resulting assembly code [7]. Thus, binary analysis plays a key role in reverse engineering where a system is analyzed to identify its objects and their interrelationships which helps us to create representations of the system at a high level of abstraction.

The most widely-used microprocessor architecture is the Intel x86 family of processors, our primary focus is on the ELF binary format executable [9] where our approach is more intuitive and heuristic. The objective of this research is to develop a novel algorithm to find a number of parameters used in a function, its type equivalence and parameters used as a pointer or an array. The primary language used is Python 3.x and tested over 444 functions in total for GCC and 200 for Clang from stripped binaries compiled on both Intel 32-bit and 64-bit with both O0 and O2 optimizations levels. The JIL file of the JIMA tool developed by CSDS of the University of Idaho is used as an input for the algorithm which is demonstrated in detail in later sections. The algorithm has used reaching definition analysis and iterative algorithm by Principles of Program analysis [8].

In the remainder of this chapter, Section 1.1 walks through basic concepts, the need for analyzing binary executables, history, and recent developments related to binary analysis and reverse engineering. Section 1.2 introduces the challenges and problems related to binary analysis as well as the details of the initiative done to resolve that. The Scope and motivation of this research are presented in Sections 1.3 and 1.4 respectively. Section 1.5 concludes with an overview of the organization of the thesis.

1.1 Need for Analyzing Binary Executables

Static binary analysis helps in program analysis and compiler optimizations without running the programs [3]. This process is being more widely used due to several factors, including:

- Integrating software trusted code with third-party libraries/application untrusted code. For example, Firefox browser relies on Adobe Reader for viewing and printing Adobe Portable Document Format (PDF) files and Adobe Flash Player to deliver Adobe Flash experiences by playing Adobe Flash content on web pages [17].

- Module-based software development for productivity and cost control. It is important to have fault isolation among the modules from different vendors to build an entire system for robustness [3].

1.1.1 Overview

Many tools like ATOM [27], EEL [27], Vulcan [29], and Phoenix [28] evaluate executables with the help of symbol-table or debugging information. As a result, in the absence of debugging information, these tools cannot be used for analyzing viruses, worms, or other malicious software, or for analyzing linked third-party libraries. Therefore, binary static analysis paves the way for all possible states a program reaches during execution without compiling the program.

Disassemblers and debuggers are helpful as they separate the raw bytes of code from the data in an executable. Disassemblers, like IDAPro, are good at recovering control-flow data and call graphs that show relationships among each function from an executable [25]. However, disassemblers are not very helpful regarding information about the contents of memory for every instruction and dataflow between instructions that access memory in an executable. Further, when indirect jumps and calls are made, disassemblers cannot always recover control-flow information. So many techniques have been proposed to find high-level dataflow information, but it is possible only through the help of the registers. To further drill it down, using static analysis and heuristics in combination with disassemblers is more feasible and reliable than solely relying on disassemblers and debuggers. In general, we need binary-level static analysis due to many factors:

- The source code is either unavailable or unreliable (non-trusting third-party software) for evaluating malicious code such as viruses, worms, botnets, Trojan horses, and extensions such as browser plug-ins, database add-ons, etc.

- Modern compilers inadvertently produce imprecise code by optimizing the code. As a result, the semantics of binary code is different from that of source code, which is not

```
int *pwd = (int*)malloc(sizeof(int)*length);
read_pwd(pwd,length);
process(pwd,length);
memset(pwd,0,length);
```

Figure 1.1: A source-code fragment to show how some compilers can do aggressive optimizations

built to support semantics for security [2,23]. Hence, compliant compilers lack the necessary knowledge to uphold the level of security that the programmer intended.

- Security analysts attempt to detect certain machine instructions and dynamic security checks into the binary code shortly before them in order to safeguard the subject code. Security engineers place a dynamic security checks immediately before the indirect jump, call, and return instructions, for instance, is done to make sure that a program's execution pathways always adhere to a predetermined control flow graph [18, 19]. Software Fault Isolation (SFI) also ensures that untrusted code does not visit disallowed data regions by dynamic checking instructions right before each indirect memory visiting instruction [20]. But often compiler optimizations discard SFI by changing the order of instructions. The dynamic instructions which were used to isolate the potentially unsafe software code are not executed effectively before the memory accessing instruction. Therefore, binary-level static analysis is useful as it is done after compiling the code.

- To enhance runtime performance, some binary-level optimizations can be made. Also, if numerous modules are statically linked together, some global information is typically available at the binary level. This can enable some optimizations that are not possible in a compiler without whole program optimization due to the requirement to support separate compilation. Optimizers can employ binary code to determine undetected features of some programming languages, like the order in which arguments are evaluated in C.

- The ability to reuse the binary code directly might be helpful when the source code is

lost, inaccessible, or is no longer supported by the original authors [21]. For example, Trojan horses hide themselves by encrypting or compressing internet traffic. The breach would be discovered quickly with improved security if a firewall can extract the decryption and uncompressed functions in binary code directly for decrypting the data.

Consider the code fragment shown in Figure 1.1, is taken from a login program, which temporarily stores the user password in a dynamically allocated buffer pointed to by 'pwd' variable. To shorten the life time of sensitive information, which is password in this case, the programmer zeroes-out the buffer pointed to by password before returning it to the heap while the optimizing compiler such as Microsoft Visual C++ .NET optimizes it away by wiping out the array used by *'memset'* operation to improve runtime efficiency. Hence, this causes a security vulnerability because the residual value in the password array might be read later by adversaries [2]. This also means that sensitive information is exposed in the heap. A similar type of issue was found in the Windows security push in 2002 [2, 22] which could not be discovered by source code. It could be identified only by analyzing the binary level code obtained from an optimized compiler. Yang et al. also reported that some bugs are producing imprecise code after being compiled even with the correct input [3, 23].

1.1.2 Advantages

Due to the very low-level and complex nature of binary code, binary-level static analysis is highly challenging. Yet, as explained, there are advantages of doing static analysis on binary code over source code.

- Binary analysis has the potential of finding security vulnerabilities regardless of source code availability. Many software are open source projects and due to the protection of intellectual property, owners do not release their source code. Often, source code is not supported by the original developers. Since binary-level analysis requires binary code only, source-level analysis is not possible in the absence of source code [30].

- Analyzing programs at statement or a block of statements, such as loops, switch statements, etc., might lead to higher false-positive rates as data flow information

```
int internal(int val1) {
   if(val1 >= 0)
        return 1;
    else return 0;
}
```

internal.c

```
extern int internal(int val2);
int external(int val2)
{
    int val3 = val2 * val2;
    return internal(val3);
}
```

external.c

Figure 1.2: An *example for global* information (Above image is referred from [3])

would be incorrect. This is taken care of in binary analysis as code is analyzed after being executed in the presence of compiler switches, exact control flow modelling is automatic; for instance, buffer checks need not be matched [30]. Static detection techniques have been applied to viruses, polymorphic worms detection, rootkit tool detection and spyware-like behavior [32, 33, 34, 9].

- Since modern compilers optimize and manipulate the code either by dead code elimination or changing the sequence of instructions, analysis of the code after compilation does not seem to be easy or precise. However, these optimizations can be understood at a low level as binary code is the final product generated in the last phase. Further, sometimes program written in high-level language has embedded inline assembly which is usually possible to understand by binary analysis. Some other optimizations like removing memory clearing of cryptographic keys can also be detected with the help of static analysis and automated tools. This is accomplished by statically analyzing the code and locating any potential bugs using a variety of methods, including disassembly, decompilation, and symbolic execution. This analysis, for instance, may reveal code lines where memory cleaning instructions are conditional or may be removed by the compiler for optimization. Additionally, binary-level defects linked to memory clearing of cryptographic keys can be found via pattern matching, heuristics, rule-based analysis, etc. with the aid of automated tools and software analysis frameworks [30].

- Binary code is generated by linking separate object files together which have global information. Usually, this information is difficult to get before compilation. Some compilers have the tendency to compile one module at a time which makes it hard to analyze other modules during compilation. Thus, static analyzers use a conservative approach and make assumptions regarding the functions of other modules at the source level and intermediate representation level which again gives incorrect results. However, there are some less common compilers such as LLVM which support whole program optimizations. Let us consider two files *'internal.c'* and *'external.c'*, as shown in Figure 1.2, that have an internal function and an external function, respectively. The external function from file *'external.c'* calls the *'internal'* function which returns either 0 or 1. And since *val3* will always be greater than or equal to zero, the output of the *'external'* function will always be the integer value 1. If compiling both files separately, the compiler will not be able to obtain this information as there is no link between both files. However, with a linked binary file generated from both of the files, static analyzers could obtain this information in advance and, therefore, determine that function *'external'* will always return 1 as the output. Thus, global information which is not present during compilation can be found in a binary file which helps in optimization too [3].

- Sometimes, it is not possible to predict the behavior of a compiler. Many compilers can accept a call made to a function with fewer parameters than what is required. For a compiler this unsafe code is easy to compile as the remaining parameters at the call site will be overwritten by the default values in the function being called. The analysis of these fewer or extra parameters can be obtained from an executable [31] whereas source analysis will not make a proper approximation.

- There are numerous programming languages, but none of them is appropriate for all types of jobs. Nowadays, developers are using hybrid programming to overcome the shortcomings of each language. There are several examples, such as Java interfaced with C or C++ via the Java Native Interface (JNI) to access legacy libraries, inline assembly embedded with C++/C/Java, and so on. As a result, binary-level static

analyzers are more efficient because they only need to analyze in machine language, whereas source-level static analyzers must support multiple languages, which can lead to errors [3].

1.1.3 Recent Developments

As discussed previously, a lot of effort has been made towards analyzing applications written in high-level languages such as Java, C, or PHP [9]. This has resulted in a variety of approaches to finding security flaws. The static analysis techniques are safe and computationally feasible for the set of values, or behaviors, during run-time [35]. These techniques are useful as they provide information without actually running the program and thus prevent any transformation or optimization (injecting malicious code) that a compiler might generate. Table 1.1 (adapted from [7]) shows the comparison of some prevalent tools which are used for static binary code analysis. Static analysis follows a logical order of steps that are applied to low-level language. Most of the analysis work has been done on accessing memory conservatively. Static binary code analysis is a two-step process, disassembly and analysis.

Disassembly refers to the process of statically decomposing an executable into its corresponding assembly code by several processes [7].

- Linear Sweep: A Linear disassembler works by assuming that an executable is made up of numerous portions of sequential streams of binary code. However, instructions are not always sequential and could be broken up by data which might be mistakenly treated as instructions. Also, compilers may insert NOPs or other filler bytes to help align instructions to certain byte boundaries.

- Travelling recursively: In this technique, for each control transfer command, an attempt is made to find all potential target addresses. The disassembler takes into account the target and the fall-through (the instruction following a conditional jump instruction) addresses any instruction.

- Recovering Indirect Jump Table (IJT): As an optimization, compilers will create tables of addresses for switch case statements the addresses in these tables are used though

Tool	Supported OS			Supported binary format		Analysis Technique	Intermediate Representation
	Windows	Linux	Mac OS	Executable and Linkable Format (ELF)	Portable Executable (PE)		
IDAPro	✓	✓	✓	✓	✓	By several heuristics & symbol table; recursive traversal	
JIMA		✓		✓		Linear sweep; recursive; heuristics ; value set analysis; limited symbolic execution	✓
obj-dump		✓		✓		Linear Sweep	
vine		✓		✓		Symbolic execution	✓
code-surfer	✓				✓	Value set analysis; computes system dependence graph	✓
veracode	✓			✓		Taint tracking; fault detection; high level representation	

Table 1.1: Comparison of tools for binary-level analysis.

```
d1: movl 0x10(%ebp), %eax
d2: %eax, -0x4(%ebp)
d3:-0x4(%ebp), %eax
```

Figure 1.3: Sample assembly code in AT&T syntax

indirect jump instructions. Normally, it is not feasible to recover the IJT table. However, Cifuentes and Van Emmerik[58], made it possible by performing intraprocedural slicing to extract the related instructions for any indirect jump by determining the ranges using the program's use-def chains. But, they did not track values that are not in registers.

Static Analysis of Assembly Code: There are various techniques to perform static analysis:

- Analysis of control flow: A control flow graph maps the connection between sets of sequential blocks. It is prepared by examining the control statements like conditional jump, calling other functions, etc. After identifying all call sites and targets, nodes and edges were constructed which represent basic blocks and execution flow between the blocks, respectively.

- Data flow: This technique gives information about the possible set of values at every instruction in a program by using the classic reaching definition algorithm. Using control flow analysis, it determines all the valid paths that reach that point and how data traverses those paths. In the research and development of our approach, a similar reaching definition algorithm was used which is elaborated in detail in Section 3.2.1. Consider the code snippet in Figure 1.3. A reaching definition analysis states that the value stored in register *'%eax'* at d1 is a reaching definition to the value stored in stack location '*-0x4(%ebp)'* at d2 which is used in d3.

- Alias Analysis: An alias means that the same memory location gets accessed by two different references. For example, if two pointers are referring to the same memory

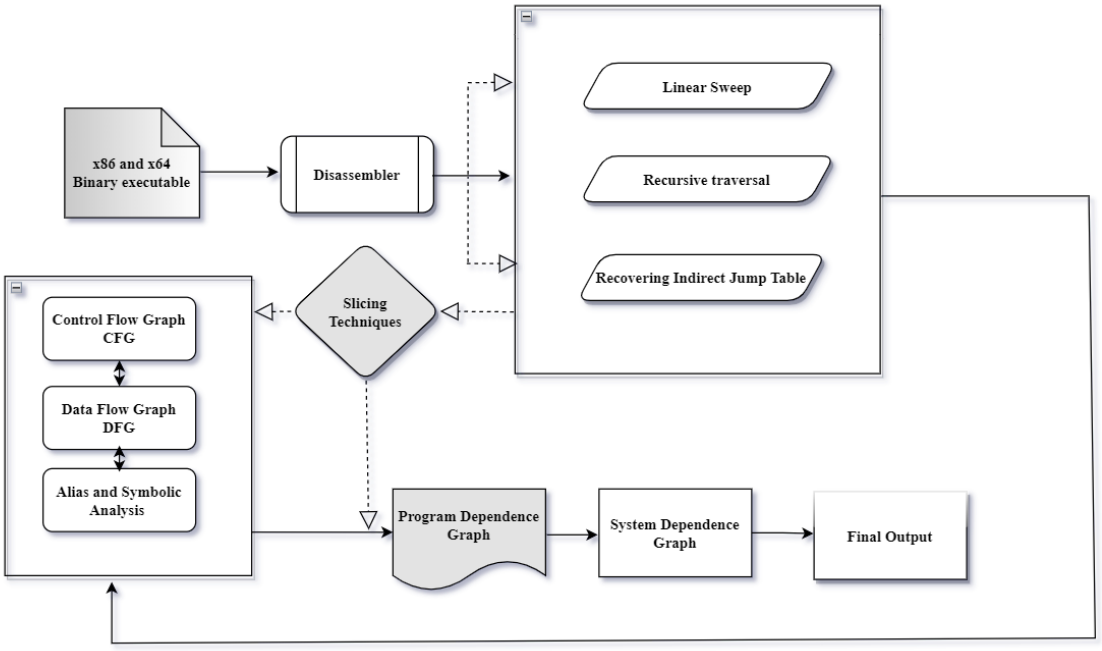

Figure 1.4: A Generalized Flow Diagram of processes of static binary tools.

location, then both are aliases of each other. Obtaining the precise memory location using alias analysis is challenging. Researchers such as Balakrishnan and Reps have proposed the implementation of value sets approximation techniques where the values of memory locations using strided intervals are calculated [31]; Debray et al. proposed approximating memory ranges stored in each register using address descriptors [36], etc.

- Symbolic execution: It is feasible to assess the values of variables and valid constraints by replacing the definitions of variables with symbols using backward or forward propagation along with a data flow analysis without even executing the program [7]. A tool called IntScope detects integer overflow vulnerabilities by symbolic evaluation [37].

- Slicing of Program: A program sliced at a node ‘n’ is represented by the sequence of statements that may get impacted if the slicing was made at forward or backward sequences. An interprocedural slicing method was proposed by Kiss et al. [38] where CFG (control flow graph) and CDG (control dependency graph) were combined into a

program dependence graph (PDG). All the PDGs were then integrated into a System Dependence Graph (SDG) and the slice was computed.

A lot of work towards disassembly and static binary analysis has been done in addition to the ones discussed earlier. A few attempts have been made toward indirectly accessing the memory. Amme et al. created an algorithm for data-dependence analysis [39] and Guo et al. proposed a low-level intermediate representation for pointer analysis which tracks registers in a flow-sensitive manner [40]. Scharwz et al. combined linear sweep and recursive traversal techniques and created a hybrid algorithm where they tested the output of both disassembler techniques to determine if they were correct by matching them [42]. Orso et al. created a disassembly method for use on obfuscated code [43] which relied on CFG checks to verify disassembly correctness. If there is any branch to the middle of an instruction then it indicates an error. Figure 1.4 shows the flow diagram of the process of static binary tools.

1.2 Challenges Encountered in Analyzing a Binary Executable

Analyzing an executable is advantageous, however, it presents some major issues such as robustness of analysis, data mixed with code, indirect control flow, deliberate code obfuscation, no run-time information and a high false-positive rate as it becomes difficult to identify the relevant properties of data and its objects which leads to poor accuracy. Modified binary becomes, thus, more challenging to understand or deconstruct. The remainder of this section discusses some of these challenges in more detail.

1.2.1 Debug/Symbol Information is Not Present

The debugging or symbol information is completely absent in cases of stripped binaries and potentially malicious programs including viruses, worms, etc. In these situations, we need to find the entry point, tracking of memory and register at all instructions. This led to the development of a few techniques where data and its objects were recovered from Intermediate Representation. However, these techniques have certain disadvantages. For example, EEL operates on SPARC binaries [41] cannot identify functions that are called by indirect control

```
void displayString(char str[])
{
int i = 0;
printf("String: ");
while (str[i] != '\0') {
printf("%c", str[i]);
i++;
}
printf("\n");
}
```

Figure 1.5: Source code fragment of function in C

transfer and does not work for variable-length ISA. An example of function identification which is still very common is *'ebp'* based stack unwinding in x86 in a heuristic fashion. Nowadays, it is not valid anymore as call frame information (CFI) forms the basis of a modern stack and the records of the same are embedded in the majority of commercial off-the-shelf binaries. Experiments demonstrate that CFI is difficult to comprehend, and even when a case study is performed, the results were not reliable. Some commercial tools are being utilized for obfuscation to prevent attacks. In order to protect the author's intellectual property, proprietary algorithms, and license from getting tampered with, various program modifications are made in order to retain the semantics of the original program. Thus, commercial tools are used for obfuscation which is one of the reasons for missing information about transformed data. Additionally, malicious code is frequently obfuscated to thwart detection and analysis. Statically, attackers encrypt the code, decrypt it at runtime and get destroyed after execution. In order to prevent code from being examined, or even decompiled, adversaries can reorder the blocks or instructions and introduce effect-free instructions like 'nop' or trash bytes that are almost never executed. Sometimes, a call to a single branch function is redirected from many function's calls to insert the junk bytes just after the call instruction to avoid disassembling [43]. According to Moser et al., static analysis alone is not enough for detecting malware since opaque constant primitives are used in challenging ways [44].

1.2.2 Absence of Data-Flow Tracking

Unlike source code analysis, information about data flow is not explicitly present in binary

```
0000000000001189 <displayString>:
1189:endbr64
118d:pushq  %rbp
118e:movq   %rsp,%rbp
1191:subq   $0x20,%rsp
1195:movq   %rdi,-0x18(%rbp)       // char str[]
1199:movl   $0x0,-0x4(%rbp)         // i=0
11a0:leaq   0xe5d(%rip),%rdi #2004 <_IO_stdin_used+0x4>
11a7:movl   $0x0,%eax
11ac:callq  1090 <printf@plt>     //printf("String: ")
11b1:jmp    11d1 <displayString+0x48> //loop starts
11b3:movl   -0x4(%rbp),%eax
11b6:movslq %eax,%rdx
11b9:movq   -0x18(%rbp),%rax
11bd:addq   %rdx,%rax
11c0:movzbl (%rax),%eax
11c3:movsbl %al,%eax
11c6:movl   %eax,%edi
11c8:callq  1070 <putchar@plt>  //printf("%c", str[i])
11cd:addl   $0x1,-0x4(%rbp)     //i++
11d1:movl   -0x4(%rbp),%eax
11d4:movslq %eax,%rdx
11d7:movq   -0x18(%rbp),%rax //accessing array start addr
11db:addq   %rdx,%rax  //getting addr of ith element
11de:movzbl (%rax),%eax // reading ith element
11e1:testb  %al,%al
11e3:jne    11b3 <displayString+0x2a> //(str[i]!='\0') loop to 11b3
11e5:movl   $0xa,%edi
11ea:callq  1070 <putchar@plt> //printf("\n")
11ef:nop
11f0:leaveq
11f1:retq
```

Figure 1.6: Disassembly of compiled function from Figure 1.5 in AT&T syntax.

code. The variables provide a surface-level abstraction of the program's memory address space which is tracked by tools for source code analysis. Hence, it is difficult to track the flow of data through memory as source code analysis do not make use of use-def chains or def-use sets. A def-use chain represents the flow of a variable's value from its definition to its uses while a use-def chain while a use-def chain represents the flow of a variable's value from its uses back to its definition. However, data flow information gets exposed by data analysis and other ways. For example, Value Set Analysis (VSA) [2] is a dataflow analysis algorithm used to access memory locations.

Example 1.2.2 The code fragment shown in Figure 1.5 and Figure 1.6 is used to demonstrate how memory locations get accessed. The function displays the string (character array) by looping through the array. It is compiled at O0 optimization level with GCC compiler for 64-bit Intel ISA. For Intel 64-bit, parameters are passed by using registers, *'%rdi'* being the first parameter. At index '1195', a pointer to the character array (a parameter of function 'displayString') is now stored in the local variable on the stack at the location *'-0x18(%rbp)'*. The value of another local variable 'i', stored on stack at *'-0x4(%rbp)'* location used as a counter is set to zero initially as stated at index '1199'. The program contains a 'while' loop from the index '11b1' to '11e3' traversing through each element by i^{th} index. Inside the loop, a call is made to a function 'putchar@plt' to display the content of that element and then the counter is incremented by 1 at '11cd'. From '11d7' to '11de', *'%rax'* is used to access the first element's address and then add the current address of i^{th} to the value in *'%rax'* to finally fetch the content of the character array at i^{th} value. This is done at index '11de'. The comparison is being done at '11e1' to check the exit condition of the loop.

As shown in this example, memory is accessed either through specifying an absolute address (directly) or by reference of an address expression of the $[base \pm (index \times scale) + offset]$ (Intel format) indirectly. The base and index are registers, and scale and offset are integer constants with the scale as a factor of 1, 2, 4, or 8. Therefore, tracking the contents of every memory location in the program could be an alternative for retaining the data-flow information. Additionally, values at local variables are stored on stack and copied back and forth to registers, unlike source code that uses the variable name. However, it is hard to keep track of contents at each memory location statically as the size of address spaces is quite large in modern machines [2]. Hence, information regarding data flow and data-objects could be only deduced in the case of stripped binaries.

1.2.3 Lack of Structure Information and Memory-Access Expressions

Binary code lack's structure and does not have a defined control flow graph and not a single instruction is guaranteed to be unique because two instructions in an x86 program could overlap. Indirect call and jump instructions make it very challenging to create an accurate and

comprehensive control flow graph from binary code. Direct branch and call instructions are relatively easy to recover the branch targets and call targets as they are encoded in the instructions themselves. Thus, it is imperative to make approximate assumptions conservatively. As first stated in the prototype of CFI [18, 19], any function could have been called by an indirect call when the address is known. Further, the only information present in any binary code is *'byte'* as a structure along with *'register'* and *'memory location'* as high–level types. Other high-level type information like data types (integer, float, etc.) and data structures (such as pointers, arrays, user-defined struct, etc.) get lost during the compilation of the program. Even worse, there is no function boundary or procedure defined in binary code [3]. It is quite challenging to say where the function starts or ends. Additionally, it is impossible to find whether code and data are coiled together or entirely separate from each other in the code section.

Many techniques have been used to obtain information about memory-access expressions. The methods proposed are generally approximate assumptions or unsound. Let us try to determine the dependence of data between instructions of an executable. Considering example 1.2.2 and Figures 1.5 and 1.6. Let us name instruction at index '1199' as i1, at index '11cd' as i2 and at '11d1' as i3. Instruction i1 is data dependent on instruction i2 as the instruction i1 has written value 0 to local variable -0x4(%rbp) which is read and incremented by 1 at i2. However, i2 is not dependent on i3 at index '11d1' where the value of -0x4(%rbp) is being read but not written in the loop. Debray et al. [36] proposed an algorithm of alias-analysis stating that any memory written by any instruction could affect the memory read operation for any other instruction. This led to the inference that i2 is dependent on both i1 and i3 which is a poor overly-conservative approach gives high false positive rates. Kiss et al. [38] proposed that two memory locations are aliases of each other by heuristics which may fail to classify any data dependency between two instructions like i1 and i2. A few of the many reasons why memory-access expressions are difficult to identify are [2]:

- Word-sized address values could be constructed from improperly aligned reads and writes as memory accesses are not required to be symmetric.

```
add 3, %edx
shl %eax, %ecx
jg  label1
```

Figure 1.7: A code fragment explaining the absence of implicit features in some compilers for x86 ISA.

- Manipulations done on memory addresses are a bit complicated. For example, instruction at index '1195' in Figure 1.6, some arithmetic operations must be performed to dereference the actual value of a local variable to get a memory location. Added expressions on indirect memory addressing makes it difficult to comprehend.

- At the hardware level, there is no way to differentiate between a memory address and a normal integer value as types are not well defined.

- There is no information about the heap as it is changing dynamically, which is a shortcoming for static analysis. Abstractions of the heap where the assumption is to have one summary node per malloc site does not give useful information in binaries. And, due to scalability issues, complex abstractions are not applied.

1.2.4 Complex Architecture and Self-Modifying Code

Intel x86 is the most prevalent ISA (Instruction Set Architecture) of the Complex Instruction Set Computing (CISC) family. These instructions range between one to fifteen bytes in length with opcodes of one, two or three bytes long having up to four prefixes, each taking one byte. Consider an instruction *prefetchnta 0x56e58955.* The hexadecimal encoding for this is *0x0f 18 05 55 89 e5 56* of which *0x55* represents push *'%ebp'* in hexadecimal notation. Further, *0x89 e5* is a two-byte encoding of *'mov %esp, %ebp'* and *'push %esi'* is denoted as *'0x56'* in the hexadecimal form. Overall, this means that depending on the start byte we can interpret this in many ways or even execute different instructions based on the start address. The fourth byte of that instruction can assume to be a typical start of any function in an executable [3], which can affect static function detection. Hence, the ISA itself is very complicated to start with.

Some compilers do not take care of some features of a processor like implicit definitions of the flag register in x86 as they are very complicated and expensive to simulate. This may cause issues in data flow analysis in modern microprocessors. People who work with static analysis tools where a particular family of compiler does not consider some important features of the processor architecture that are used by other compiler families. In x86, arithmetic and logical instructions set these flags implicitly. Consider the instructions *'add* and *'shl'* as shown in Figure 1.8. Instructions like *'add'*, *'sub'*, *'shl'*, etc., generally have six flags - SF, ZF, AF, PF, CF and OF [45]. The first instruction 'add' adds 3 to *%edx* (following AT&T syntax) and then *'shl'* shifts the value in *%ecx* to left by *%eax* bits. *'jg'* jumps to label1 if ZF=0 and OF=SF [45]. Depending on the contents of *%eax*, these Instruction Flags are set. If the content of the *%eax* register is zero, there is no effect on any flags except for OF which is affected by 1-bit shifts. Thus, we can see some unmodeled effects [3].

Sometimes during execution, code either gets modified or a new code is generated. So, we cannot consider binary code complete, as we do not have knowledge of some code prior to program execution. As this type of self-modifying code is not present until runtime, static analysis cannot be performed since this is all dynamic in nature. It has been observed that untrusted code can hide and generate malicious code at runtime which destroys the actual code after the execution. Hence, we need dynamic analysis in addition to static. The untrusted code can be run on a simulator to get fed into static analyzers later. Yet again, dynamic code only opts for a single flow of instruction during execution which again limits the scope of analyzing any executable [3].

1.2.5 Finding function parameters and local variables for 32-bit and 64-bit

Analyzing the parameters of a function is an important step towards extracting semantics accurately and perfectly from binaries. First, parameters are examined at function and indirect call-sites. Targets of indirect calls can be reduced if we keep the count of arguments generated before call-sites and consumed by functions. Incorrect transfers at call-sites can be limited by having the knowledge of counts of parameters. Getting the semantics correct requires the reaching definition analysis of every instruction to track the direct or indirect calls. Sometimes,

as we have seen, the architecture of x86 and x64 is very complex and is instruction oriented. Therefore, the process of finding the parameters can sometimes lead to false positive results too. Also, since modern compilers optimize the code in many ways, this may confuse the typical pattern of instructions and then the recovery of data from registers and memory location from binary executables gets aggravated.

Based on a given architecture's calling conventions, it may be statically possible to find the actual number of parameters/arguments and local variables used in a particular function. This process initially aimed at a given processor (Intel [45]) can be generalized later and is higher in performance compared to other approaches. Since every ISA has different set of instructions and optimization levels depending on the compiler, finding parameters can be challenging at times. Thus, the algorithm presented in this thesis is not complete but is an approach we believe can work for all ISA's. The logic for this algorithm is the memory locations accessed directly in the stack without any assignment of value are considered to be a parameter. In AT&T syntax, the first value in the instruction is the source followed by a destination. Hence, keeping track of all the sources and destinations and verifying whether the source is already used in the destination at every instruction might give us the memory location of those parameters. In a 32-bit GCC compiler, the number of arguments is the references to EBP + (n>=8) within a given function. In 64-bit GCC, we get the values from the registers used directly as the source. However, for the Clang compiler, the logic changes drastically as the parameters are defined with respect to the stack depth. Generally, in 32-bit Clang, we consider parameters as the memory locations used directly as a source on the stack of the form ESP+ (n>stack depth). However, in 64-bit Clang, registers are used directly, and the values are stored at the memory locations of the form ESP- (n<= stack depth). Thus, it is quite challenging to create a general algorithm which deals with different compiling techniques.

Further, for the local variables also, both the compilers follow different strategies like the arguments. In GCC, the negative offset to all references of EBP can be stated as local variables i.e. EBP – (n>=8). And in Clang, usually all the memory locations of the form ESP- (n<= stack depth) is considered to be the locations where the local variables are stored. We can look for values onto the stack, as these values are often function parameters. The rationale behind this

```
push ebp
mov ebp, esp
mov eax, [ebp+8] ; load pointer parameter from stack into eax
```

Figure 1.8: An example showing the form a parameter in a binary

theory is explained further in section 2.2. Also, we can check for the instructions that use the stack pointer (ESP) or base pointer (EBP) to access the function parameters. However, these strategies present certain limitations and challenges which are discussed in Chapter 5.

1.2.6 Finding function parameters used as pointers for 32-bit and 64-bit

Analyzing memory access to find the contents of the memory location is one of the most challenging tasks since low-level instructions use explicit memory address and indirect addressing to manipulate data. Further, in stack manipulation a function may manipulate the stack in complex ways, such as by pushing and popping values to save and restore registers or directly modifying the stack pointer. This can make it difficult to identify the location of function parameters on the stack. Sometimes, calling or returning target values out of the function bounds makes it hard to keep track of the flow. The way function arguments and return values are passed between functions is governed by calling conventions. Different compilers and platforms may use different calling conventions, which can make it difficult to identify the location of function parameters in the code. Obtaining the semantics precisely can be divided among inter-procedural, IR, and constructing SDG. This is done by creating a set of reaching definitions at every instruction and inspecting the source and destination by dereferencing the pointer and checking whether it is a parameter or not. To obtain the signature of subroutines, Cristina [46] developed an inter-procedural analysis method which identifies the set as the actual call parameters and calculates the data set containing the live data locations at each call site.

As discussed in section 1.2.5, after fetching formal parameters, we need to check for the instructions that access the memory pointed to by the pointer parameter. Consider an example shown in Figure 1.8 which states that if the function takes a single pointer parameter, it might

be of the form [ebp+8]. In this example, the value of the pointer parameter is loaded from the stack into the EAX register. However, the exact details of how function parameters are passed and accessed can vary depending on the calling convention used by the compiler, so we may need to consult the documentation for the specific compiler and architecture.

1.3 The Scope of Our Work

As we have seen previously, analyzing a stripped binary executable is quite hard and challenging, we have designed a novel approach or a basic algorithm which finds the formal parameters, local variables and parameters used as pointers in a function. The approach is :

- It uses the static analysis tool, JIMA and the formatted file called JIMA Intermediate Language (JIL), the actual binary tool which does the analysis by linear sweep, recursive traversal, heuristic approaches, jump table lookup, Intermediate Representation, value set analysis for jump table, limited symbolic execution, exception block detection, etc. techniques. This algorithm is developed as an additional functionality for the JIMA tool. It starts off by finding a set of target addresses at each instruction through reaching definition analysis explained in Chapter 3.

- For 32-bit compiler O0 for both GCC and Clang, we assume parameters are passed directly on the stack and referenced using *'%ebp'* register. We use EBP + (n>=8) as formal parameters and the reference of the form EBP - (n>=4) as local variables. At every instruction from the start of the function, we add all the sources and destinations until we reach the target.

- For 64-bit O0 for both GCC and Clang, the values are directly fetched from the registers instead of passing the value onto the stack as it has enough memory. Generally, the value from the registers gets stored in the memory locations where local variables are stored of the form EBP - (n>=4) for later use. Again, at every instruction, by keeping track of sources and destinations up till that instruction and finally, find the unique sources by checking if any source is there in the destination or not. If it is not present in the destination, then we consider it a parameter.

- For 32-bit O2 for both GCC and Clang, the parameters are not referenced by the use of *%ebp* base register rather via the stack pointer *%esp* with respect to the memory locations that lesser than the stack depth. As we know, the stack makes space by the instruction *'subl $0x20, %esp'*. This means any memory location of the form *'%0xn(esp)'< stack depth* will a local variable. Thus, any memory address greater than the stack depth will likely to be a parameter. Also, sometimes, it deals with the registers directly where it pushes '*%ebx*' to store some value which might get used later in the function. At the O2 level, there are other possible scenarios too, we handled them by taking them case by case and we are still incorporating that into our algorithm.

- Similarly, for 64-bit O2 for both GCC and Clang, we followed the heuristics where registers are used directly to get the values which are stored in local variables to be used later. The process of finding the parameters is similar to O0. However, there are differences in how the local variables are obtained since in the O2, the values from registers are first moved to a register and from there to a local variable. Hence, we sometimes get extra registers as parameters and extra local variables. O2 is difficult to handle as the compiler optimizes the code and sometimes these heuristics do not work.

- The algorithm can work on any program compiled from a high-level language or assembly. Even with the compiler optimizations, the success rate is quite high. Additionally, the approach mentioned above is able to work on the executable that modifies the code section on-the-fly, i.e., self-modifying code. This algorithm also handles obfuscations like memory-access reordering, instruction reordering, junk-byte insertion, register renaming, etc.

- Finding the parameters and local variables leads to the analysis of the parameters used as pointers. The first step of the analysis is to dereference every source and destination and finally compare both with the parameters. If the source or destination is a parameter then it is a parameter used as a pointer. If it does not matches then it traverses back through the backward analysis which means tracing backwards and knowing from

where the value is coming in the current pointer. If the pointer is pointing to the value which is either coming from a register or a local variable where the register saved its value for later use, then that pointer is pointing to a parameter. This means that it is again a parameter used as a pointer. If it is coming from the local variable, then we ignore that pointer.

- In order to understand how to find local variables and pointer used as a parameter, we need to understand the workflow, reaching definition and iterative algorithm. They are explained in detail in Chapter 3.

1.4 Motivation and Contributions

The thesis is focused on the development of an algorithm to find the number of parameters/arguments and local variables in a function. Also, the algorithm finds parameters used as pointers in functions. This can be further extended to check for arrays passed as parameters. The ultimate goal behind this research is to statically recover the signature of a functions data types - integer, unsigned, signed; arrays; user-defined structures, etc. from stripped binaries. The approach presented in this thesis is an added functionality for the tool JIMA and uses the format JIMA Intermediate Language (JIL) built and distributed by the University of Idaho under the Center for Secure and Dependable Systems. JIMA is an open source software tool designed for the analysis of binary executables. It has been tested for over 650 functions from stripped binutils, coreutils and findutils binaries including O0 and O2 levels compiled with both GCC and Clang and are matched with the ground truth. The algorithm works on the *'Executable and Linkable format'* (ELF) which is a popular file format and currently works on Intel [45] ISA 32-bit and 64-bit. Sometimes, if a binary is being compiled on 32-bit and a 64-bit parameter was passed, the parameter was broken into two 32-bit parameters. On other occasions, only two parameters were actually used in the function definition despite the function declaration where three were passed. These were the times when it broke or found the ground truth depending on the scenario. Since the compilation techniques are different for GCC and Clang, we observed a few abnormalities in Clang too where every 'struct' or 'array element' is considered as a local variable. Apart from these, there are other

limitations and discrepancies which is discussed in detail in Chapter 5. The algorithm uses a reaching definition algorithm and iterative algorithm which is explained in section 3.1 along with the examples.

1.5 Thesis Overview

The organization of the remainder of the thesis is summarized as follows. Chapter 2 provides background on the concept of the binary executable, assembly language, and stack memory management. It also focuses on the work contributed by others in this field. It introduces the University of Idaho's static tool for binary analysis called JIMA and how it is being used in the development of the algorithm. Chapter 3 introduces the algorithm developed in detail. The approach includes reaching definition analysis (RDA) and iterative algorithm which is explained in detail with the approach, actual implementation and examples along with the dataflow diagrams. It states how RDA and iterative algorithm integrated in the algorithm. Chapter 4 presents all the experiments and test cases run on stripped binaries and showcased the results with the success rate defined in terms of precision, recall and F1 score for both GCC and Clang. There were around 650 test cases out of which some gave false results due to the limitations and compiling techniques. Chapter 5 draws attention to the challenges, limitations and drawbacks of analyzing indirect memory locations statically. In this chapter, the shortcomings of the algorithm have been discussed for how and why the algorithm presented is not complete. Finally, chapter 6 provides a conclusion, research contribution and what can be done for future work and the ways to tackle it.

Chapter 2 : Background

A file containing a machine or low-level code that can be run directly by a computer's processor is called a binary executable. It is sometimes referred to as an executable file or a binary file. A compiler or an assembler transforms source code into a computer executable machine code. An executable could be a group of many files or libraries merged and linked together by a linker that creates a final executable. Binary executables are used to run applications, operating systems, and other software on a computer. Additionally, there are binary data files, which are collections of bytes, words, or even arrays. When a binary executable is executed, the computer's processor reads and interprets the machine code in the file and executes the instructions. Binary files are different from text files as they are not available for human reading whereas text files are specific to the character set. Also, binary executables are usually platform-specific, meaning they are compiled for a particular type of computer or operating system. For example, due to compatibility issues, an executable file which is compiled for Windows will not run on a Mac or Linux computer without some kind of portability layer or virtual machine. However, binary files can have text strings which could be either ASCII or Unicode. Generally, some programs or files with extensions such as .bin or .exe are identified as executable.

2.1 Fundamental Concepts of Binary Executable

Executables are a sequence of bytes of data written in low-level code. Although programs written in high-level languages are convenient to read for humans, the binary executable is fastest in interacting with the hardware as it is in the binary digit form, ones and zeroes. These files are often stored in a container format like *'Executable and Linkable format'* (ELF) or *'Portable Executable'* (PE). An executable is generally divided into sections - .text (executable code), .data (initialized global and static variables), and .rodata (read-only data, such as constants and strings) as shown in Figure 2.1. They are essential for system calls and tasks which are necessary at runtime.

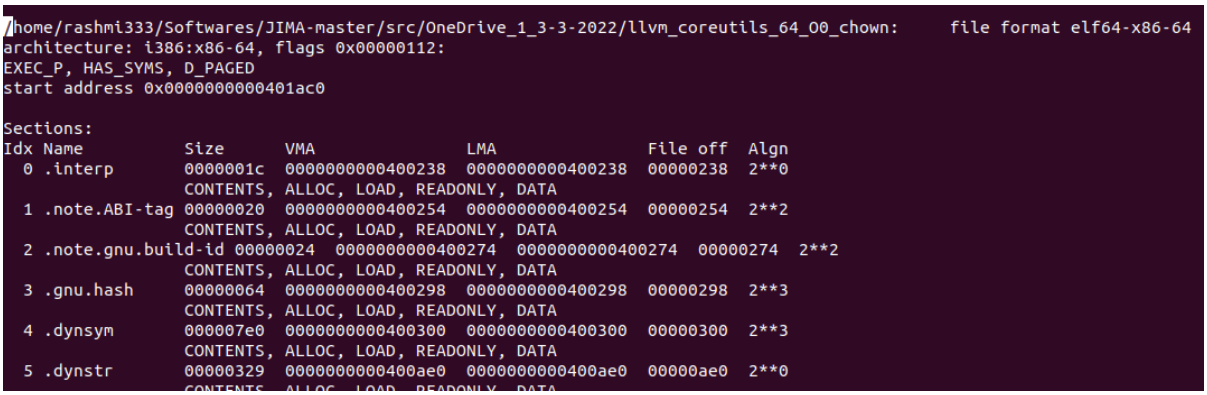

Figure 2.1: An example of a binary file (ELF format).

From high-level source code, machine-level instructions can be generated which are often referred to as assembly language. Often, dynamically, there is only a single execution path which does not cover all the dataflow. Let us walk through the importance of reverse engineering and static binary analysis in detail.

2.1.1 Reverse Engineering and Static Binary Analysis

Reverse engineering is the process of analyzing, dissecting and comprehending how a program or application works by looking at its parts, functions, and design. In software, reverse engineering often involves examining compiled code to understand how a program works or to modify it for a specific purpose. It can also involve analyzing malware or other software to understand how it functions and how it can be defended against. Reverse engineering is an important process, as shown in Figure 2.2, for understanding complex systems, improving products, identifying security vulnerabilities, and protecting intellectual property. There are several reasons why reverse engineering is necessary:

- **Understanding complex systems**: Reverse engineering can be used to understand complex systems that are difficult to comprehend by simply examining their external behavior. This is especially useful when working with legacy systems or when trying to integrate different systems together.

  Product design and improvement: By reverse engineering a product or system, engineers can gain insight into the design choices made by the original creators which paves the way for innovative design.

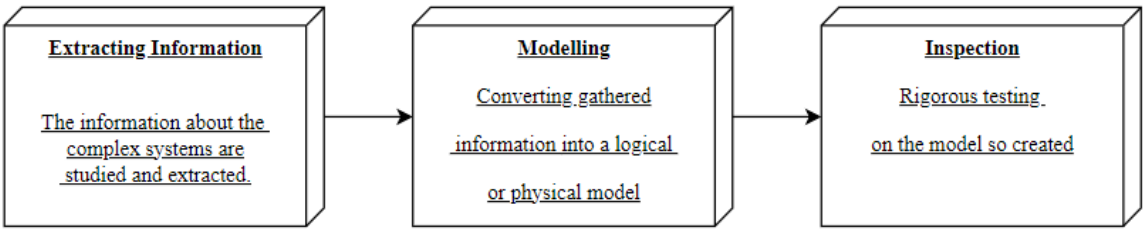

Figure 2.2: Basic process of reverse engineering.

- **Security analysis**: Reverse engineering can be used to identify vulnerabilities and weaknesses in software and hardware systems. It can be applied to identify security issues with a product. Instead of learning about these flaws at the distribution phase, reverse engineering allows for the detection of all such problems during the research process itself. This is important for protecting systems against attacks, as it allows developers to understand how an attacker might try to exploit a system.

- **Intellectual property protection**: Reverse engineering can also be used to identify intellectual property violations, such as copying of software designs.

Static binary analysis is a type of reverse engineering and software analysis that involves examining the binary code of a program without actually running it. In static binary analysis, the binary code is analyzed using specialized tools and techniques that are designed to detect and identify certain types of issues, such as buffer overflows, race conditions, and other types of bugs. This analysis can also help identify areas of the code that might be susceptible to exploitation by attackers or that could pose a risk to the security of the system. Some of the tools used in the static binary analysis include disassemblers, decompilers, and debuggers, as well as more specialized tools for analyzing specific types of vulnerabilities or code structures. The results of the analysis can then be used to improve the security of the software, to identify areas for optimization and performance improvements, or to provide insights into the inner workings of the program. Some of the static binary tools are JIMA [59], IDAPro [25], etc. In both static binary analysis and reverse engineering, the goal is to understand the behavior of compiled code. Reverse engineering often involves static binary analysis as a critical step in the process of reconstructing the original source code or design specifications. Similarly, static

binary analysis often involves reverse engineering techniques to identify and understand the behavior of compiled code, including analyzing its control flow, data flow, and calling conventions.

2.1.2 Basics of Assembly Language

Since static binary analysis requires knowledge of low-level or assembly language, this section talks about the assembly language required for understanding the stack memory management and analysis of a function for finding the parameters in detail. Since our algorithm works for Intel 32-bit and 64-bit executables, this section is based on understanding that the Intel ISA [45] has a specific instruction set. Assembly languages frequently act as bridges, enabling the development of more complicated programming languages, which can increase a developer's efficiency. For the development and testing of the algorithm, we have used ‘C’ and ‘C++’ as a high-level language converted into a binary executable and stripped binaries as binutils, findutils and coreutils to work on. The components are described in the coming sections.

2.1.2.1 Registers

Intel processors are backwards compatible and thus from 32-bit and 64-bit, the bits were added. With prefix ‘E’ in 32-bit, EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are the core eight general purpose registers. Similarly, having ‘R’ as a prefix in 64-bit, the eight registers are named as RAX, RBX, etc. The other eight are named numerically r8 through r15. The summary of Intel register naming convention is given in Figure 2.3.

2.1.2.2 Memory and Addressing Modes

Declaration of Static Data Region: Data declarations should be preceded by the .DATA directive and have the locations labeled with names for later reference like variables. For example,

Z DD 1, 2, 3 ; Declare three 4-byte values, initialized to 1, 2, and 3. The value of location Z + 8 will be 3.

str DB 'hi',0 ; Declare 2 bytes starting at the address str, initialized to the ASCII character values for ‘*hello*’ and the null (0) byte.

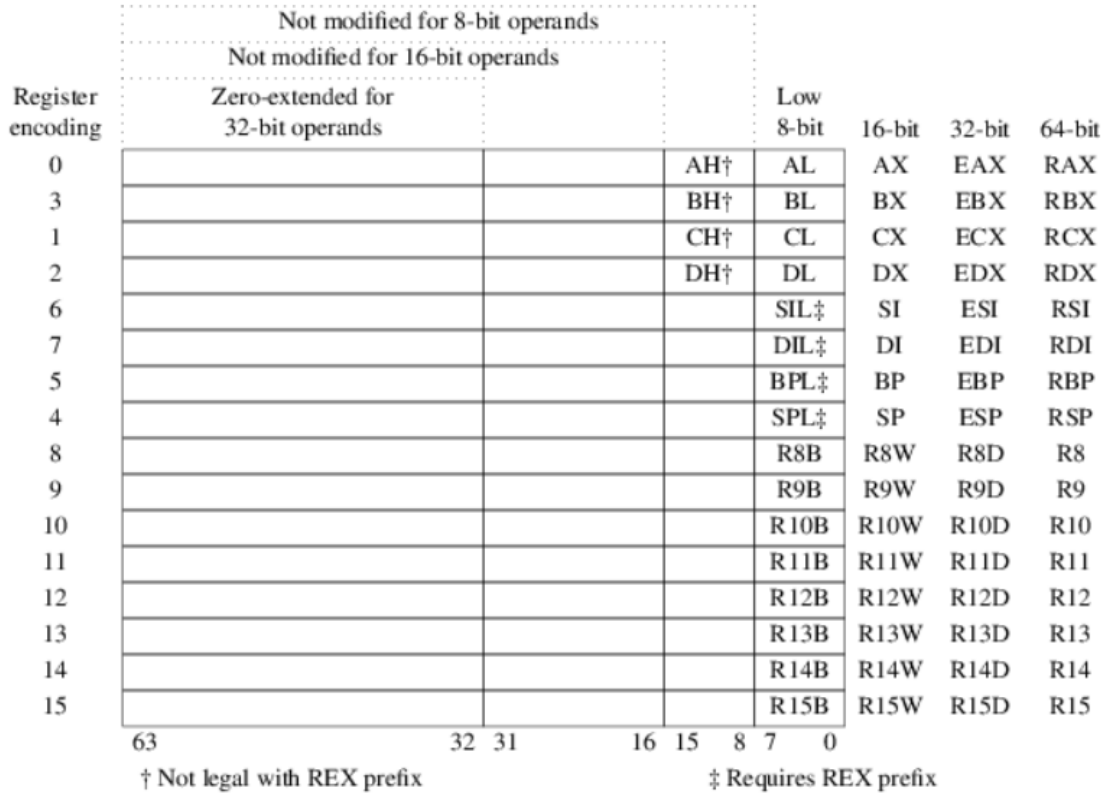

Figure 2.3: General purpose register in x86. (Adapted from Intel's guide manual)

- **<u>Addressing Memory Access</u>:** Modern x86-compatible processors are capable of addressing up to 2^N bytes of memory: memory addresses are N-bits wide. For example,

  mov %cl, (%esi,%eax,1) ; Move the contents of CL into the byte at address ESI+EAX

  mov (%esi, %ebx,4),%edx ; Move the 4 bytes of data at address ESI+4*EBX into EDX

- **<u>Size Directives:</u>** In general, the assembly code instruction in which the data item is addressed can be used to determine the intended size of the data item at a specific memory address. For example,

  movb $2, (%edx) ; Move 2 into the single byte at the address stored in EDX.

  movl $2, (%ebx) ; Move the 32-bit integer representation of 2 into the 4 bytes starting at the address in EBX.

2.1.2.3 Instructions

The instructions can be divided into three categories – control flow, arithmetic & logic and data transfer. Examples of each type of instructions are as follows:

- **Data transfer:** This includes instructions like mov, lea, push, pop, etc. where the actual data or content is getting transferred. For example,

  movb $3, 0x08445 — store the value 3 into the byte at location having memory address as *0x08445*

  pushl %ebp — pushing register ebp on stack.

  leaq 0xe5d(%rip),%rdi — the address of (%rip)+0xe5d and store in %rdi.

- **Arithmetic and Logical Instructions**: These perform some logical or arithmetic operations that include increment & decrement, addition, subtraction, integer multiplication, and, or, xor, nor, shl etc. For example,

  addl $0x1,-0x4(%rbp) ; adding 1 to the value stored in %ebp-0x4.

  shll $1, %eax — shift all the bits in %eax by 1 spot left effectively.

- **Control Flow Instructions**: An instruction pointer (IP, EIP) register is used to track where the current instruction starts in memory. After an instruction has been executed, it ordinarily increments to point to the following instruction in memory. The given control flow instructions update the IP register implicitly rather than allowing direct manipulation. This type includes jump, conditional jump, compare, return, call, etc. Examples are –

  jmp 11d1 <displayString+0x48> ; jump to a target address

  callq 1070 <putchar@plt> ; call a target outside the function

2.1.2.4 Calling Conventions

A protocol describing how to call and return from routines is known as the calling convention. For instance, a programmer need not look up a subroutine's definition to know how parameters should be handed to it if there are rules for calling conventions. Furthermore, high-level language compilers can be designed to adhere to a set of calling convention rules, enabling

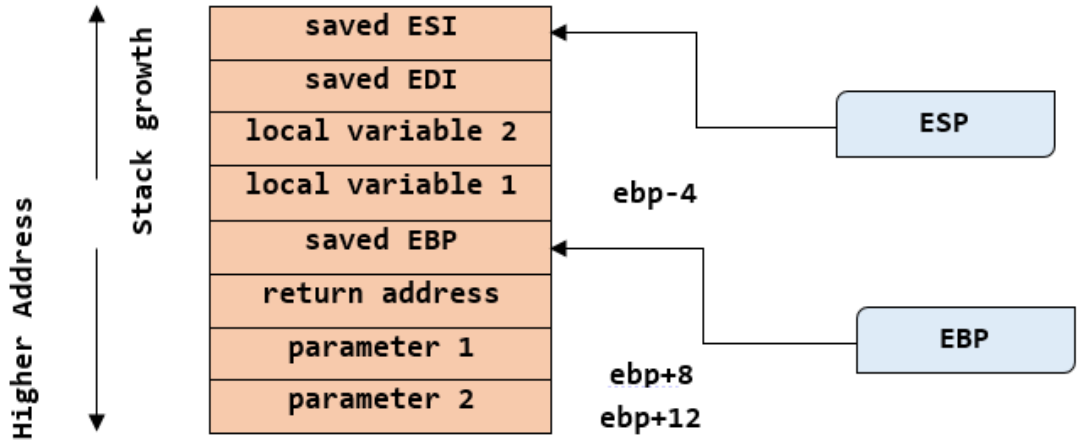

Figure 2.4: Stack during subroutine call (Reference taken from x86 Assembly Guide)

hand-coded assembly language routines and high-level language routines to call one another. The utilization of the hardware-supported stack forms a significant part of the C calling convention. Instructions like push, pop, call and ret form the basis of this convention. The stack is used to pass parameters to subroutines. The stack serves as a storage location for registers and local variables utilized by subroutines. There are two sets of rules for the calling convention. The caller of the subroutine follows the first set of rules, and the writer of the subroutine abides by the second set of rules (the callee). The calling convention should be implemented with extreme caution in subroutines since failure to follow these guidelines can soon lead to severe software problems because the stack will be left in an inconsistent state. Figure 2.4 shows how a stack looks during a subroutine call with two parameters and two local variables being executed. Each memory location is 4 bytes apart and the base pointer is 8 bytes away from the first parameter. The return address is placed below the base pointer and above the parameters, the call instruction has an additional 4 bytes of offset. To exit from the procedure, ret instruction will reach the return address saved on the stack. The way function parameters are passed, return values are returned, and the stack is maintained during function calls are all governed by calling conventions. There are different types calling conventions used in programming and systems architecture, including:

- **CDECL (C Declaration):** The programming languages like C and C++ frequently make use of this calling technique. The caller is in charge of clearing the stack after a

function call according to the cdecl protocol, which requires that function parameters be passed on the stack in reverse order (the rightmost parameter is pushed first).

- **STDCALL (Standard Call):** Some Windows APIs and the Microsoft Visual C++ compiler both use this calling convention. According to the stdcall protocol, the callee is in charge of clearing the stack before returning, and function parameters are passed on it in reverse order.

- **FASTCALL:** Some compilers and architectures, like x86, use this calling pattern to enhance function call performance. Faster function calls without having to touch the stack are possible with the fastcall convention, in which function parameters are passed in registers (often in the ECX and EDX registers).

- **THISCALL:** In object-oriented programming languages like C++, this calling convention is used to handle member function calls that have a concealed this pointer as the first parameter. The *'this'* pointer is normally supplied in a register (ECX on an x86 platform, for example) or on the stack when using the thiscall convention, and the remaining parameters are also passed on the stack.

- **SYSTEM V AMD64 ABI:** Systems running on 64-bit x86 architecture, including Linux and macOS, use this calling convention. The callee is in charge of clearing the stack before returning under the System V AMD64 ABI. Function parameters are supplied in registers (such as RDI, RSI, R9) before spilling onto the stack if necessary.

2.2 Stack Memory Management

Stack memory management is a process by which a software or application allocates and deallocates memory from the stack data structure to store and retrieve data during the execution of the program. This memory allocation takes place on contiguous blocks of memory. The compiler creates and maintains the stack and activation record called stack frame. Most programming languages uses a stack to manage data for function calls. Therefore, a developer need not worry about stack allocation or deallocation. Generally, the information includes

argument values (if not passed in registers), the return address, callee-saved register values, and local variables. Every function call if not all will have at least a return address, which is the location in the calling function to return to when the function call is finished.

In stack memory, the first item popped off of the stack is the last item pushed onto the stack that means it follows the '*Last In First Out'* (LIFO) data structure. In modern microprocessors, it is assumed a stack grows down. This also means that the last item pushed is in a lower memory address. The stack often stores local variables, function parameters, calling functions, and return addresses. A stack memory management involves the following two main operations:

- **Pushing:** Pushing is the process of adding data to the top of the stack. When a function is called, its local variables and parameters are pushed onto the stack. The return address of the function is also pushed onto the stack, which allows the program to return to the calling function after the execution of the current function is completed.

- **Popping:** Popping is the process of removing data from the top of the stack. When a function returns, its local variables and parameters are popped off the stack. The return address is also popped off the stack, allowing the program to return to the calling function.

The stack is managed automatically by the compiler, which keeps track of the stack pointer with a register that stores the memory address of the top of the stack. Usually, it is referred to as ESP (Extended Stack Pointer) on 32-bit Intel systems and RSP (64-bit Stack Pointer) on 64-bit Intel systems. To access the memory address of the top of the stack, we can simply read the value of the ESP or RSP register. This value represents the memory address of the last item pushed onto the stack. When we push a value onto the stack, the ESP or RSP register is decremented by the size of the pushed value, so that it points to the start of that value on the stack. The stack grows from higher memory addresses downward towards lower memory addresses. During the pushing and popping operations, the stack pointer is changed to guarantee that the data is saved and retrieved in the proper sequence. Usually, the first two instructions in a function save the current activation record pointed by (%rbp) on the stack, i.e.

("pushq %rbp") and then copies the stack pointer to *'%rbp'*, i.e. ("movq %rsp, %rbp") as shown in Figure 2.5. The frame pointer is *'%rbp'* for the current function call. Thus, all references to local variables and to parameters or arguments in memory are made using this indirect memory addressing with either *'%ebp'* or *'%rbp'* which was mentioned in section 1.2.3. Space for local variables is created on the stack by subtraction from *'%esp'* or *'%rsp'* and then local variable is accessed with a negative offset from *'%rbp'*. The syntax for a local variable looks like *'-0x28(%rbp)'* in assembly in AT&T syntax which is an indirect memory address obtained by subtracting 0x28 as an offset from the address value in *'%rbp'*. Since stack memory management uses only a few machine instructions to push and pop data, for example, Intel has optimized memory access using *'%esp'*, *'%ebp'*, so it is fast and efficient presenting several advantages, including:

- **Efficiency and Convenience:** Because the program or operating system handles memory allocation and deallocation automatically, stack memory management is a very effective method of managing memory. This can save time and resources because there is no longer a need to manually allocate or deallocate memory for local variables.Writing and debugging code is made simpler by the program's automatic memory management of local variables and function calls.

- **Security**: Stack memory management provides a secure way of managing memory because the memory allocated on the stack is only accessible within the current function or block. This means that other parts of the program cannot access the memory, which helps to prevent data corruption and security vulnerabilities.

- **Speed:** Stack memory management is typically faster than other types of memory management because the memory is allocated and deallocated using a simple pointer manipulation. This means that programs that use stack memory can run faster and be more efficient than programs that use other types of memory management.

However, the amount of stack memory available to a program is limited, and exceeding the available stack space can cause a stack overflow error, which can lead to program crashes and other issues. Figure 2.6 shows a program having stack overflow. For the function *'foo'* it does

```
0000000000001169 <getAverage>:
    1169:    endbr64
    116d:    pushq  %rbp
    116e:    movq   %rsp,%rbp
    1171:    movq   %rdi,-0x28(%rbp)
    1175:    movl   %esi,-0x2c(%rbp)
```

Figure 2.5: A function start showing stack and base pointer making space

not include any base condition for its recursive call, i.e. calling the function *'foo'* again. This loop will never end and results in the overflow of the stack to store more values. Hence, whenever the stack memory gets completely filled, a stack overflow error occurs. In summary, stack memory management is an essential part of the execution of computer programs, allowing them to store and retrieve data in a temporary and efficient manner. The management of the stack is automatic and handled by the compiler, which uses the stack pointer to keep track of the data stored on the stack.

2.3 Related Work

Disassembly Static analysis techniques for extracting data from binary executables have received a lot of attention in the past few decades [39, 47, 9, 48, 46, 36, 40, 27]. A lot of efforts have been made towards static analysis of a binary code [4, 5, 6] to develop tools to find bugs and security vulnerabilities. Many techniques have been proposed to improve the recovery of data types [10, 11, 12], code structure [10, 13, 14, 15], and even exact syntactic identity [10, 16]. Many tools like ATOM [26], EEL [27], Vulcan [29], and Phoenix [28] evaluate executables with the help of symbol-table or debugging information. It is now possible to find some information even without the debugging information when descriptors represent collections of memory configurations [24]. Scharwz et al. combined linear sweep and recursive traversal techniques and created a hybrid algorithm where they tested the output of both disassembly and if it matched then it was considered as correct [42]. Orso et. al created a disassembly method for use on obfuscated code [43] relied on CFG checks to verify disassembly correctness. Some of the tools like GHIDRA, IDAPro, BAP and JIMA are examples of static binary analysis tools.

```
#include<stdio.h>

void foo(int a) {
   if(a== 0)
    return;
   a += 1;
   printf("\n");
   printf("%d",a);
   foo(a);
}

int main() {
   int a = 3;
   foo(a);
}
```

Figure 2.6: Stack Overflow example

Decompilation As decompilation is an important step towards analyzing stripped binaries by static analysis, there are developments in this direction like Cifuentes's [46] concentration on the recovery of expressions from instruction sequences and control flow. Also, Chang et al. created a framework for converting assembly code to high-level languages like C++ and Java [53] and makes use of a group of similar decompilers. Some decompilers were chained together to create a mechanism for a lower-level decompiler to communicate to a higher-level compiler and vice-versa. It is done by interfacing different decompilers with the modularized framework. However, this process is very complicated as the locals of a procedure need to be identified prior to decompilation.

Memory Accesses To determine information about a program's memory accesses, Rugina and Rinard [54] used a combination of pointer and numeric analysis. It was not widely adopted as their approach assumed that the local and global variables of the program are known prior to analysis. The collection of "allocation blocks" included the local and global variables and dynamic-allocation sites of the program. There was no information about alignment and stride information even though they were able to determine the range information. Pointer and numerical analyses are carried out independently even though they can be intertwined. A static-analysis method is presented by Bergeron et al. [55] to determine whether an executable that contains debugging information complies with a user-specified security policy. According to Min'e [56], a combined data-value and points-to analysis computes an over-approximation of

the values in the cells where program variables are divided into groups based on how they are accessed at each program point. His implementation does not enable study of heap-allocated storage-using apps. Furthermore, his methods are unable to draw conclusions from loop access patterns. Similarly, instead of using global reasoning about complete heap abstractions, Hackett and Rugina [57] offer an approach that employs local reasoning about specific heap areas. They essentially employ an independent-attribute abstraction, wherein each "tracked place" in a concrete memory configuration is tracked independently of the others.

Value Set Analysis Attempts have been made towards accessing the indirect memory to recover intermediate representation (IR). Ramalingam et al. [2] made an attempt by creating Value Set Analysis (VSA) which analyzes and reason about the values that program variables can take during program execution. VSA has been successful due to its precision, scalability, flexibility, and soundness properties, which make it a powerful and effective technique for analyzing program values and reasoning about their behavior during program execution. Debray et al. [36] also made a similar attempt by proposing an alias algorithm which is highly related to VSA. This alias determines an over-approximation of the range of values that each register can hold at each program point. They only maintain the tracking of the addresses' low-order bits in their analysis, which involves approximating a set of addresses using a set of congruence values. Unlike VSA, it did not attempt to track values that are not in registers. So, whenever a load from memory occurs, they lose a significant amount of precision. A few others which are closely related are Amme et al. [39] who created an algorithm for data-dependence analysis and Guo et al. [40] who proposed a low-level intermediate representation for pointer analysis which tracks registers in a flow-sensitive manner. It is ambiguous whether the Amme et al. technique completely accounts for dependencies between memory regions since it only does an intraprocedural examination. Guo et al. algorithm's [40] solely tracks registers in a flow-sensitive manner while treating memory addresses in a flow-insensitive manner. The technique achieves context-sensitivity by using partial transfer functions. Unknown initial values (UIVs) are used as parameters for the transfer functions, although it is unclear if the algorithm takes into consideration the risk of called processes altering the memory locations that the UIVs represent. Moreover, Xu et al. [49] developed a system to determine whether or not specific memory-safety features applied to SPARC executables. The untrusted program's

initial inputs were typestate and linear constraint indicated. This was based on traditional theorem-proving methods: the typestate-checking algorithm employed Omega [51] to determine Presburger formulas and the induction-iteration method [50] to create loop invariants. To find aliases in assembly code, Brumley and Newsome [52] offer an approach based on Datalog programs. The goal is to translate each assembly statement into a predicate in Datalog. All of the alias associations would be present in the resulting saturated database. To find out if any two memory accesses are aliases, tools would query the database. Although their method is intriguing, it is unclear whether it would be useful. To ensure that the analysis is completed in an acceptable amount of time, for example, they lack the concept of widening for loops. The approach described in this thesis, on the other hand, aims to recover data from an Intel x86 executable that enables the generation of intermediate representations comparable to those that can be constructed for a program written in a high-level language.

2.4 Introduction to JIMA Tool Kit and JIL file

JIMA is a binary analysis toolkit developed at University of Idaho. This section utilizes some of the research group's description of JIMA. The JIMA tool suite is written in Python3. Although the use of Python may limit performance, it also enables collaboration, portability and extensibility. The JIMA binary analysis and repair activities center on the JIL data format, a custom data structure that maintains information about the contents of a binary, an intermediate representation. The JIMA Intermediate Language (JIL) representation is a large python dictionary. This dictionary contains entries that may themselves be dictionaries, lists or sets. The use of dictionaries and sets to speed up access is at the cost of more memory used. The JIMA tool suite includes several functions and utilities to examine, evaluate and modify the JIL data structure. For reuse, JIMA stores the JIL data structure in a pickled (compressed) file. For the purposes of this thesis, we use JIL file of the JIMA tool that gives instructions, size, targets, indices etc. like the objdump file.

2.4.1 JLift

The JIMA system starts with the JLift tool. This tool is responsible for extracting information

```
def processFile(fileName):
   instructs._init_()
   with open(fileName,'rb') as fn:
      jil=pickle.load(fn)

   ins = jil['ins']
   funcs=jil['functions']
   addresses={}

   for funcId in funcs:
       addresses[funcs[funcId]['startAddr']]=funcId

   for addr in sorted(list(addresses.keys())):
      func=funcs[addresses[addr]]

       for insId in range(func['startIndex'],func['endIndex']+1):
         inst = ins[insId]
         printInst(inst,jil,sys.stdout)
```

Figure 2.7: Python Program to read a JIL file

from an ELF executable file. JLift reads the ELF file, collecting information about the file structure, individual sections, exception handlers, dynamic symbols and data sections. For disassembly of instructions, the tool invokes objdump and stores the result in a file. That file is then read and parsed. A future improvement of the tool will integrate direct calls to a dissembler, avoiding the need to invoke objdump and the corresponding extra overhead of text file writing and parsing. JLift also directly reads the ELF file collecting additional information not directly available from objdump. JLift populates the several fields that are part of the Jil data structure, such as *'startAddr'*, *'jumpPtrs'*, *'jumpedBy'*, etc.

2.4.2 Detection of Explicit Calls and Jumps

During disassembly, JIL categorizes each instruction. If the instruction is an explicit call or jump (has a hardcoded address), it records the source and target addresses as well as the reverse in data. It also keeps track of the returns. Initially all calls and jumps are a mapping from the source address to a single target address. However, with jump pointers and call pointers, the targets may be one of a list of addresses. The 'calledBy' and 'jumpedBy' data structures map the target address to a list of source addresses. These data structures are used by JIMA when detecting function starts and function boundaries. Note that in Intel 32-bit position independent

```
startAddr  = 0xbda
startindex = 111
endAddr    = 0xca9
endindex   = 164
len = 208 (0xd0)
secid      = 13
sec name   = .text
Has True Return
```

Figure 2.8: Output generated by running code in Figure 2.7.

code, a compiler can generate a call to the next instruction, which then pops the value into a register, obtaining the current instruction counter. JIMA detects this behaviour and does not include these calls in the preceding list of calls. Modern compilers use a function *get_thunk* that provides this capability.In that case it is a real function, and JIMA detects that as well. Intel 64-bit code has a separate instruction pointer register and does not need to use this trick.

2.4.3 Exception Handler Analysis

ELF data files contain two exception handling sections, eh_frame and eh_frame_hdr, which contain information about the exception handlers. The details of these exception handling sections is beyond the scope of this thesis, but it is sufficient to say that they contain tables that map regions of instructions to specific exception handlers. When an exception occurs, the tables are searched to find the region containing the current instruction pointer address and to get the appropriate handler (and possible pre-processing code). The compilers analyzed here (i.e., gcc, icc, clang) append the exception handlers to their parent functions and include them in the size of the function. Normal control flow analysis will not detect these handlers as part of the parent function, since they are not normal. This is true for linear and recursive analysis and both static and dynamic analysis (unless the dynamic analysis forces an exception to occur). As an added heuristic, JIMA decodes these tables to correctly include the handlers within the parent function’s boundaries. The other algorithmic tools appear to do this as well, but not the machine learning and neural network tools.

2.4.4 Static Binary Analysis

The JIMA tools are designed to support binary analysis. With all of the information about the executable stored in the JIL data structure, we have a collection of library routines to support

analysis and expansion of the capabilities of JIMA. A simple bit of code can be used, with the library, to start processing the JIL file. For example python code in Figure 15 can be used to generate the output as shown in Figure 16. Programmers can walk through the functions, extract information about them along with the function instructions. JIMA's opcodes library contains additional information about each instruction classifying it as math, control flow, etc. It keeps track when the opcode modifies flags and source and destinations, since some opcodes will access or modify implicit registers. All of this information is implicit in most binary analysis systems, and the purpose of JIMA is to make it queryable and usable by programmers, so that they can build additional analysis tools. The original version of Jima, used for the CGC, includes control-flow and data-flow analysis modules, has the ability to determine which values were relevant for the conditions in a jump, and detects loops, loop parameters, and estimates the numbers of arguments and local variables. We are in the process of porting all of these analysis functions to the python-based version of JIMA. A companion project to this thesis is using JIMA data to detect variadic functions in binaries, while another is using it as an interface to GDB to generate lists of breakpoints for code coverage analysis. Exception handlers exist to allow the program to execute code that is outside the normal control flow. These can cause great difficulty for binary analysis. The binary analyzer has to tread carefully with exception handlers and the data they provide but must be able to use them.

2.4.5 Binary Rewriting

The final portion of JIMA subsystem is the binary rewriting section which is not used in this thesis. Using results from the analysis, and user provided guideline, JIMA can generate a representation of the binary in two ways. This most common way is to generate assembly code, with appropriate labels and tags and pragmas, along with a build configuration file, to allow the binary to be assembled and relinked. The assembly also can include a large number of comments about the file to assist in manual inspection as shown in Figure 2.8. The second way is to provide a translation of the file into another analysis form. We currently do this in a PROLOG data format for one of our projects. We plan to allow translation into other forms, such as LLVM, as needed. We are also involved in a project to generate 'C' code from the data file and using it to guide selective editing of the raw ELF file.

Chapter 3 : The Algorithm

As we have established that analyzing stripped binaries is quite challenging, the major aim of this thesis is to improve a portion of that analysis by finding function formal parameters, local variables and parameters used as a pointer. This algorithm is implemented in several phases and uses the JIL file as an input. The algorithm is written in Python and is compatible with the 3.x version.

3.1 Initial Analysis – A Reaching Definition

Since assembly language does not use the higher-level variable names, it is not immediately clear which value a register is storing. To solve this, it is mandatory to determine the data flow of every instruction for every path possible. Hence, initially, we need data flow analysis which will converge to reach the definition. The sequence of steps starts with intraprocedural analysis that leads to reaching definition analysis and finally implements data flow equations [8]. These steps are necessary prerequisites to develop the algorithm.

3.1.1 Intraprocedural Analysis

An intraprocedural analysis is the analysis of each instruction within a single function or procedure. The analysis is defined by several formal functions. Each function consists of a list of instructions, labelled by an instruction id. To be clear, we will list multiple statements separated by a semicolon, even though the semicolon is not used in assembly language. The first function from this set has the information about the true entry called the 'Initial' label to the block and the second has information about the true exit called the 'Final' label [8]. Let us define certain terms with the help of examples stated in assembly language:

- Initial label: This returns the initial label of every function.

  $\boldsymbol{init}: \mathbf{Stmt} \rightarrow \mathbf{Lab}$

  Examples: $init(\ell: (movq \;\; \%rdi, -0x18(\%rbp)) = \ell$

- Final label: This returns the final instruction or statement of the function.

  $$\boldsymbol{final}: \mathbf{Stmt} \rightarrow \mathbb{P}(\mathbf{Lab})$$

  Examples: $final(\ell: [retq]) = \ell$

  $final(\ell: [nopl\ (\%rax)]) = \ell$

- Block: In these functions, Blocks constitute an important part of the analysis which are instructions or statements. In this thesis, we use the terms 'statements' or 'instructions' interchangeably. This function is used to access the statements or tests of a label in a program. It is of the form –

  $$\boldsymbol{block}: \mathbf{Stmt} \rightarrow \mathbb{P}(\mathbf{block})$$

  Example: $block(\ell: [addq\ \ \%rdx, \%rax]) = \{\ \ell: [addq\ \ \%rdx, \%rax]\ \}$

- Labels: Set of labels in a program is defined as –

  $$\boldsymbol{labels}: \mathbf{Stmt} \rightarrow \mathbb{P}(\mathbf{Lab})$$

  where, $\boldsymbol{labels}(\boldsymbol{S}) = \{\ \boldsymbol{\ell} \mid \boldsymbol{\ell}: [\mathbf{B}] \in \mathbf{Stmt}(\boldsymbol{S})\ \}$

Thus, we can say that, $\boldsymbol{init}(\boldsymbol{S})\boldsymbol{and}\ \boldsymbol{final}(\boldsymbol{S}) \in \boldsymbol{labels}(\boldsymbol{S})$; $\boldsymbol{Block}(\boldsymbol{S}) \subseteq \boldsymbol{labels}(\boldsymbol{S})$, such that 'ℓ' is the address or index of the current statement, 'Ᵽ' is the program, 'a' is a constant.

3.1.2 Reaching Definition Analysis

The *Reaching Definition Analysis* or reaching assignments analysis determines for each instruction the source instruction of that value stored in the registers. To conduct this analysis, at every instruction, we determine a set of two functions called RD_{entry} and RD_{exit} i.e. RD = (RD_{entry} , RD_{exit}). This is the current RD working set. An assignment of the form ℓ: movl %eax, %edi] *may* reach a certain program point (usually an entry or exit of an elementary block) along some path if there is an execution of the program where %edi was assigned a value at ℓ when the program point is reached [8]. For example, consider the code snippet shown in Figure 3.1. The label is shown at 4, 4: [movl 0x10(%ebp), %eax] called RD_{entry} reaches to 5: [movl %eax, -0x4(%ebp)] RD_{exit} which means (%eax,4) reaches the entry point at 5. Implementation

```
1:      pushl  %ebp
2:      movl   %esp,%ebp
3:      subl   $0x10,%esp
4:      movl   0x10(%ebp),%eax
5:      movl   %eax,-0x4(%ebp)
6:      movl   -0x4(%ebp),%eax
7:      movl   (%eax),%eax
8:      testl  %eax,%eax
```

Figure 3.1: A code snippet to explain Reaching Definitions

of RDA requires an understanding of a few more functions [8]. These are –

- $\boldsymbol{kill_{RD}}: \boldsymbol{Blocks_*} \rightarrow \mathbb{P}(\boldsymbol{Var_*} \times \boldsymbol{Lab_*^?})$

  This function produces the set of variables (either memory locations or registers) and labels (indices) of assignments that are destroyed or removed from the RD working set for the instruction. It can happen when a variable is overwritten by a new value. $\boldsymbol{Lab_*^?}$ is the union of all the labels in a set with initialization or no value for a particular statement or set of statements (a block) i.e. $\boldsymbol{Lab_*^?} = \boldsymbol{Lab_*} \cup \{?\}$.

  Examples:

  $$kill_{RD}(\ell: [\text{movl } \ 0x10(\%ebp), \%eax]) = \{(\%eax, ?)\} \cup \{(\%eax, \ell')\}$$
  $$kill_{RD}(\ell: [\text{subl } \ \$0x10, \%esp]) = \emptyset$$

- $\boldsymbol{gen_{RD}}: \boldsymbol{Blocks_*} \rightarrow \mathbb{P}(\boldsymbol{Var_*} \times \boldsymbol{Lab_*^?})$

  The *generator* function produces the set of pairs of variables and labels of assignments generated by a block (statement or set of statements).

  Examples:

  $$gen_{RD}(\ell: [\text{movl } \ 0x10(\%ebp), \%eax]) = \{(\%eax, \ell)\}$$
  $$gen_{RD}(\ell: [\text{pushl } \%ebp]) = \emptyset$$

Further, the analysis that has a pair of functions (RD_{entry} , RD_{exit}) maps the labels to the set of variables and indices (labels of assignment block).

$$\boldsymbol{RD_{entry}}, \boldsymbol{RD_{exit}}: \boldsymbol{Lab_*} \rightarrow \mathbb{P}(\boldsymbol{Var_*} \times \boldsymbol{Lab_*^?})$$

Example 3.1.2 As shown in Figure 3.1, for every instruction kill, gen or no function is applied which leads to the final analysis at every index. For example, the gen_{RD} function is used at indices 2, 4, 5, etc. Function gen_{RD}(4: [movl 0x10(%ebp),%eax]) = { ('%eax',4) } and the assignments are continuously added to the final analysis set of every instruction. Thus, by analyzing the block from the start, we will get the analysis of instruction at index 4 as {('%eax', 4), ('%esp', 3), ('%ebp', 2)}. Similarly, $kill_{RD}$ can be defined at 6 and 7 such that $kill_{RD}$([movl -0x4(%ebp),%eax][6]) will overwrite the value of '%eax' at index 4. Hence, the final analysis at index 6 is {('-0x4(%ebp)', 5), ('%eax', 6), ('%esp', 3), ('%ebp', 2)}. Finally, full analysis of the code snippet is –

final_analysis {1: set(), 2: {('%ebp', 2)}, 3: {('%esp', 3), ('%ebp', 2)}, 4: {('%eax',4), ('%esp', 3), ('%ebp', 2)}, 5: {('-0x4(%ebp)', 5), ('%eax',4), ('%esp', 3), ('%ebp', 2)}, 6: {('-0x4(%ebp)', 5), ('%eax',6), ('%esp', 3), ('%ebp', 2)}, 7: {('-0x4(%ebp)', 5), ('%eax',7), ('%esp', 3), ('%ebp', 2)}, 8: {('-0x4(%ebp)', 5), ('%eax', 7), ('%esp', 3),

3.2 Implementation

As discussed before, the implementation of this algorithm is currently limited to Intel's ISA in AT&T syntax. The theory behind finding the function parameters in 32-bit is that *%ebp* or *%rbp* act as a reference on the stack. Going towards the positive offset gives us the formal parameters and the negative offset determines the local variables. And in 64-bit, we keep track of the sources which have not been used before as destinations or are unique to sources. This has been discussed in Sections 2.2 and 2.3 in detail. For finding the parameter used as a pointer, we require RDA as it will determine the flow where the pointer was last accessed, initialized, or changed which is discussed in Section 3.1. The pointer is dereferenced and to compare or check the value from where it is coming, the algorithm traverses back through the path of data flow for all the instructions. If the pointer is coming from the assignment of the parameter which we discover in the first part, then we say that the parameter is used as a pointer at some instruction with index ℓ. Before the actual implementation, a few steps are performed which are common for all the major findings like finding sources, destinations for each instruction or

```
char * get_archive_member_name_at (struct archive_info *arch, unsigned long
offset, struct archive_info *nested_arch)
{
  size_t got;
  if (fseek (arch->file, offset, SEEK_SET) != 0)
    {
      error (_("%s: failed to seek to next file name\n"), arch->file_name);
      return NULL;
    }
  got = fread (&arch->arhdr, 1, sizeof arch->arhdr, arch->file);
  if (got != sizeof arch->arhdr)
    {
      error (_("%s: failed to read archive header\n"), arch->file_name);
      return NULL;
    }
  if (memcmp (arch->arhdr.ar_fmag, ARFMAG, 2) != 0)
    {
      error (_("%s: did not find a valid archive header\n"),
            arch->file_name);
      return NULL;
    }
  return get_archive_member_name (arch, nested_arch);
}
```

Figure 3.2:‘get_archive_member_name_at’ function written in C language from gcc_binutils_32_O0_elfedit file.

statements. Also, we need to create a flow list or working list which states the flow of each statement to the next in order to get the final analysis of the whole function. Then, finally solving the data flow equation forms the basis of the algorithm. To understand all the steps in detail, we use an example of a C function given in Figure 3.2, converting it into a binary code as shown in Figure 3.3 and finally obtaining the JIL file which will serve as an input to the algorithm. Further, by showing the output at every step, we will demonstrate how we fetched the function parameters, local variables and the parameters used as pointers for a function.

3.2.1 Approach

We need to install the JIMA toolkit and then set the home variable in the JIMA environment. The function given in Figure 3.2 is a function of a program written in C language to swap two integers. It is done for Intel ISA and compiled by 32-bit GCC compiler which gives us a binary file in ELF format. Further, we execute JIMA and save the JIL file as an output in the current

```
0804b7ff <get_archive_member_name_at>:
 804b7ff:    pushl  %ebp
 804b800:    movl   %esp,%ebp
 804b802:    pushl  %ebx
 804b803:    subl   $0x24,%esp
 804b806:    movl   0xc(%ebp),%edx
 804b809:    movl   0x8(%ebp),%eax
 804b80c:    movl   0x4(%eax),%eax
 804b80f:    movl   $0x0,0x8(%esp)
 804b817:    movl   %edx,0x4(%esp)
 804b81b:    movl   %eax,(%esp)
 804b81e:    calll  8048ba0 <fseek@plt>
 804b823:    testl  %eax,%eax
 804b825:    je     804b84e <get_archive_member_name_at+0x4f>
 804b827:    movl   0x8(%ebp),%eax
 804b82a:    movl   (%eax),%ebx
 804b82c:    movl   $0x804d400,(%esp)
 804b833:    calll  8048c40 <gettext@plt>
 804b838:    movl   %ebx,0x4(%esp)
 804b83c:    movl   %eax,(%esp)
 804b83f:    calll  804a378 <error>
 804b844:    movl   $0x0,%eax
 804b849:    jmpl   804b8fb <get_archive_member_name_at+0xfc>
 804b84e:    movl   0x8(%ebp),%eax
 804b851:    movl   0x4(%eax),%eax
 804b854:    movl   0x8(%ebp),%edx
 804b857:    addl   $0x34,%edx
 804b85a:    movl   %eax,0xc(%esp)
 804b85e:    movl   $0x3c,0x8(%esp)
 804b866:    movl   $0x1,0x4(%esp)
 804b86e:    movl   %edx,(%esp)
 804b871:    calll  8048bd0 <fread@plt>
 804b876:    movl   %eax,-0xc(%ebp)
 804b879:    cmpl   $0x3c,-0xc(%ebp)
 804b87d:    je     804b8a3 <get_archive_member_name_at+0xa4>
 804b87f:    movl   0x8(%ebp),%eax
 804b882:    movl   (%eax),%ebx
 804b884:    movl   $0x804d324,(%esp)
 804b88b:    calll  8048c40 <gettext@plt>
 804b890:    movl   %ebx,0x4(%esp)
 804b894:    movl   %eax,(%esp)
 804b897:    calll  804a378 <error>
 804b89c:    movl   $0x0,%eax
 804b8a1:    jmp    804b8fb <get_archive_member_name_at+0xfc>
 804b8a3:    movl   0x8(%ebp),%eax
 804b8a6:    addl   $0x6e,%eax
 804b8a9:    movl   $0x2,0x8(%esp)
 804b8b1:    movl   $0x804d426,0x4(%esp)
 804b8b9:    movl   %eax,(%esp)
 804b8bc:    calll  8048b60 <memcmp@plt>
 804b8c1:    testl  %eax,%eax
 804b8c3:    je     804b8e9 <get_archive_member_name_at+0xea>
 804b8c5:    movl   0x8(%ebp),%eax
```

```
804b8c5:    movl   0x8(%ebp),%eax
804b8c8:    movl   (%eax),%ebx
804b8ca:    movl   $0x804d42c,(%esp)
804b8d1:    calll  8048c40 <gettext@plt>
804b8d6:    movl   %ebx,0x4(%esp)
804b8da:    movl   %eax,(%esp)
804b8dd:    calll  804a378 <error>
804b8e2:    movl   $0x0,%eax
804b8e7:    jmp    804b8fb <get_archive_member_name_at+0xfc>
804b8e9:    movl   0x10(%ebp),%eax
804b8ec:    movl   %eax,0x4(%esp)
804b8f0:    movl   0x8(%ebp),%eax
804b8f3:    movl   %eax,(%esp)
804b8f6:    calll  804b5e3 <get_archive_member_name>
804b8fb:    addl   $0x24,%esp
804b8fe:    popl   %ebx
804b8ff:    popl   %ebp
804b900:    retl
```

Figure 3.3: Assembly code of C function in AT&T syntax given in Figure 3.2 obtained from objdump.

directory. Figure 3.3 is an assembly code of the function given in Figure 3.2. It could be considered as a general iterative algorithm for any distributive framework for a function ***f*** which could be of an application, program, or a dynamically executable file.

Analysis of each statement or block is required to compute the data flow algorithm. This analysis consists of *extreme labels* which could be the initial or the final index of a function *f* and a finite dataflow. The algorithm uses *analysis* as a dictionary that contains the input information up to any label. The input information for any label of the elementary block is the control flow analysis done by the reaching definition. This reaching definition includes all the assignments made to variables up until that label. The analysis dictionary is indexed by the labels through *gen* and *kill* functions which have already been discussed in section 3.1.2. This analysis of each label is created by the **worklist** which is the pair of the current and possible target's indices. By looping through every instruction using the JIL file, we create the list of **sources** and **destinations** also. Further, we calculate the final analysis using the data flow algorithm. With the sources and destinations, we find formal parameters and local variables if any register or memory offset used directly as a source can be considered as a parameter. For this, we can look for instructions that move values onto the stack directly. The rationale behind

INPUT:	A function *ƒ* having distributive framework of an application:	
OUTPUT:	parameters(), local variables(), pointer as parameter()	
METHOD:	***Step 1:***	calculating sources & destinations

		sources := nil; destinations := nil; final_result := nil;
		init := init(*ƒ*(block)); final := final(*ƒ*(block));
		for all inst in range(*ƒ*[init], *ƒ*[final] + 1)
		  if inst is call then check get_pc_thunk
		  if reg in sources then final_result.append(inst[0]);
		  else pass;
		  else if inst in *jump, return, push, lea, pop, shr, shl* then
		  destinations.append(inst[0]);
		  else if inst in *cmp, mov, add, sub, test* then
		  sources.append((inst[0]);
		  destinations.append(inst[1]);
		  else check len(inst[agrs])>1 then
		  sources.append(inst[0]);
		  destinations.append(inst[1]);

	Step 2:	calculating parameters

		for src in sources:
		  if src not in destinations:
		  final_result.append(src);
		parameters := set(final_result);

	Step 3:	creating flowlist

		flowlist := nil;
		for all inst in range(*ƒ*[init], *ƒ*[final] + 1)
		  if inst :== *ƒ*[init] then flowlist.append(ℓ,ℓ+ 1),
		  where ℓ is the label and ℓ’ = ℓ+1
		  else if inst is *cndJump* then
		  ℓ’ := inst.get('ctrlTarget');
		  flowlist.append(ℓ, ℓ’);
		  flowlist.append(ℓ, ℓ +1);
		  else if inst in *jump, call* then
		  if inst.get('ctrlTarget') is None then
		  flowlist.append(ℓ, ℓ + 1);
		  else
		  ℓ’ = jilLib.getInsIdByAddr(jil,
		  inst.get('ctrlTarget'))[1]
		  flowlist.append(ℓ, ℓ’)
		  else if inst is *ret* then *continue*;
		  else if inst is *jump and* ℓ’ ∉ range(*ƒ*[init], *ƒ*[final]) then continue;
		  else *flowlist.append(ℓ, ℓ + 1);*
		flowlist = flowlist.copy();

	Step 4:	computing analysis for each ℓ using RDA

Table 3.1 cont'd

		analysis := set(); // initialized with the null set for all ℓ
		if inst :== ƒ[init] then analysis[ℓ] := set([‘null’, ℓ])
		else if inst in mov, add, sub, lea then
		analysis[ℓ] := set(destination,ℓ) ∪ analysis[ℓ -1]
		else analysis[ℓ] := analysis[ℓ -1])
		for aly in analysis[ℓ].copy() do
		if aly[0] is null then remove
		analysis[ℓ].remove[aly]

	STEP 5:	using general iterative algorithm computing final analysis of every statement or instruction
		Iteration (updating worklist and Analysis)

		worklist = flowlist.copy(); final_analysis = {}
		while worklist ≠ nil do
		ℓ := worklist[0][0]; ℓ’:= worklist[0][1];
		worklist := worklist.pop(0);
		final_analysis[ℓ] = do kill & gen (initial_analysis[ℓ])
		if final_analysis[ℓ] ⊄ analysis[ℓ’] then
		analysis[ℓ’] := analysis[ℓ’] ∪ final_analysis[ℓ]
		ℓ’’ := value for item,value in flowlist if item == ℓ’;
		for all ℓ’’ with (ℓ’, ℓ’’) in flowlist do
		worklist.append((ℓ’, ℓ’’));

	STEP 6:	Fetching parameter used as pointer

		temp1 := inst[0]; temp2 := inst[1];
		local_argument :=nil; pointer_as_parameter := nil;
		if (temp1 ≠ nil or temp2 ≠ nil) and sources[ℓ] in (parameters or local_argument) then
		local_argument.append(temp1 or temp2);
		else for all item,val in flowlist if val :== ℓ do
		for all address in final_analysis(item) do
		if sources[ℓ] :== address and sources[ℓ] in (parameters or
		local_argument) then
		pointer_as_parameter.append(temp1 or temp2);

Table 3.1: Pseduo-code of the algorithm

this approach is that the number of arguments is the references to EBP + (n>=8) within a given function. Also, the negative offset to all references of EBP can be stated as local variables, i.e., EBP – (n>=8). After fetching formal parameters, we need to check for the instructions that access the memory pointed to by the pointer parameter. By looping through all the instructions one by one and using sources, destinations, worklist, initial analysis, final analysis, parameters,

and local variables, which we calculated above, we find the parameters used as a pointer by dereferencing every memory location. This location is either checked directly in the parameters list or traced back by the backward analysis to get the flow which tells if the value of the location is flowing from a parameter. There are many rules and limitations associated with the approach which are discussed in later sections. The implementation is done in Python3 and initially aimed at Intel processors [45] which can be generalized later for other ISAs. The pseudo-code of the algorithm is presented in Table 3.1 step by step with the description. In the coming sections, we will understand the steps to develop the algorithm in detail along with the dataflow diagrams (DFDs).

3.2.2 Finding Sources and Destinations

The first step is to create a list of sources and destinations of every instruction despite the flow of data or control. This is done by looping through all the instructions within the range of the *first index* (f[init]) and *end index +1* (f[final]) of a function by checking the *start* and *end* addresses. First, we check that the instructions we are working on have arguments which means they either have a source or a destination. This is done by examining the value of the '*args'* key of the JIL instruction dictionary. Otherwise, the loop continues and checks for the type of instruction which plays a key role in identifying the source and destination. For example, '*mov'* has the first register or memory offset as the source and the second as the destination. In Figure 3.3, instruction (movl 0xc(%ebp),%edx) has *'0xc(%ebp)'* as source and *'%edx'* as destination. Similarly, (pushl %ebp) only has a destination. This segregation has been done based on semantics. Once getting the instruction, we check whether the instruction involves registers or indirect memory addressing. This is done by checking the value of that argument's key 'type' of that particular argument. If the type is 'reg' then it is a register and if it is of type *'memOffsetBase', 'memBase'*, etc., we consider it as indirect referencing. Moreover, a check is made to see if the base, offset, or index has a null value before dereferencing the address in order to determine the memory location to assign as a source or a destination. A source can be an immediate or a constant value also. This is true if the value of *'type'* is *'imm'*. It should be noted that some checks were also made to consider the registers to be the same as 32-bit and 64-bit in the final code, which is discussed in detail in the 3.3 rules section of this chapter. For

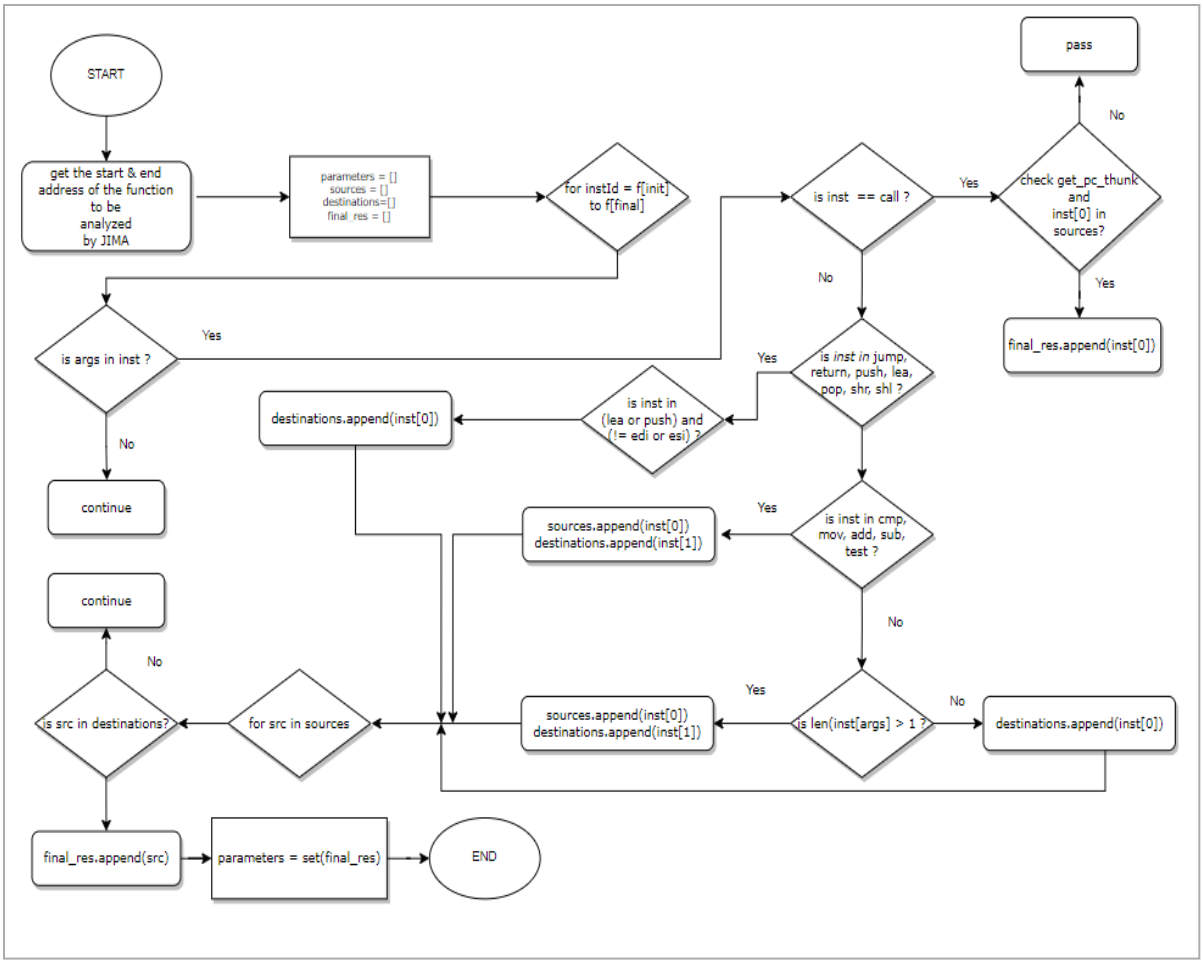

Figure 3.4: Data Flow diagram for calculating sources, destinations, and parameters.

example, *%eax* is the same as *%rax* and so is *%ax*. The output of the sources and destination for Figure 3.3 is:

sources {3095: '%ebp', 3096: '%esp', 3097: '%ebx', 3098: '$0x24', 3099: '0xc(%ebp)', 3100: '0x8(%ebp)', 3101: '0x4(%eax)', 3102: '$0x0', 3103: '%edx', 3104: '%eax', 3106: '%eax', 3108: '0x8(%ebp)', 3109: '(%eax)', 3110: '$0x804d400', 3112: '%ebx', 3113: '%eax', 3115: '$0x0', 3117: '0x8(%ebp)', 3118: '0x4(%eax)', 3119: '0x8(%ebp)', 3120: '$0x34', 3121: '%eax', 3122: '$0x3c', 3123: '$0x1', 3124: '%edx', 3126: '%eax', 3127: '$0x3c', 3129: '0x8(%ebp)', 3130: '(%eax)', 3131: '$0x804d324', 3133: '%ebx', 3134: '%eax', 3136: '$0x0', 3138: '0x8(%ebp)', 3139: '$0x6e', 3140: '$0x2', 3141: '$0x804d426', 3142: '%eax', 3144: '%eax', 3146: '0x8(%ebp)', 3147: '(%eax)', 3148: '$0x804d42c', 3150: '%ebx', 3151: '%eax', 3153:

destinations {3095: 'null', 3096: '%ebp', 3097: 'null', 3098: '%esp', 3099: '%edx', 3100: '%eax', 3101: '%eax', 3102: '0x8(%esp)', 3103: '0x4(%esp)', 3104: '(%esp)', 3105: 'null', 3106: '%eax', 3107: 'null', 3108: '%eax', 3109: '%ebx', 3110: '(%esp)', 3111: 'null', 3112: '0x4(%esp)', 3113: '(%esp)', 3114: 'null', 3115: '%eax', 3116: 'null', 3117: '%eax', 3118: '%eax', 3119: '%edx', 3120: '%edx', 3121: '0xc(%esp)', 3122: '0x8(%esp)', 3123: '0x4(%esp)', 3124: '(%esp)', 3125: 'null', 3126: '-0xc(%ebp)', 3127: 'null', 3128: 'null', 3129: '%eax', 3130: '%ebx', 3131: '(%esp)', 3132: 'null', 3133: '0x4(%esp)', 3134: '(%esp)', 3135: 'null', 3136: '%eax', 3137: 'null', 3138: '%eax', 3139: '%eax', 3140: '0x8(%esp)', 3141: '0x4(%esp)', 3142: '(%esp)', 3143: 'null', 3144: '%eax', 3145: 'null', 3146: '%eax', 3147: '%ebx', 3148: '(%esp)', 3149: 'null', 3150: '0x4(%esp)', 3151: '(%esp)', 3152: 'null', 3153: '%eax', 3154: 'null', 3155: '%eax', 3156: '0x4(%esp)', 3157: '%eax', 3158: '(%esp)', 3159:

where 119, 120 and so on are the indices of the function. Also, if the instruction is a *'call'*, we need to check whether the call has been made to a function called *get_pc_thunk* which is used for position-independent code on x86. It loads the position of the code into a called register (for example, if we have instruction - **call 0xf60d2f47 <__i686.get_pc_thunk.bx>,** %bx in instruction is a register used as source) which allows global objects (which have a fixed offset from the code) to be accessed as an offset from that register. Position-independent code can be loaded and run at several addresses without any changes. Code that will be linked into shared libraries must take this into consideration because those libraries may be mapped at various addresses in separate processes. However, an equivalent call is not required on x86-64, because that architecture has IP-relative addressing modes (that is, it can directly address memory locations as an offset from the current instruction location).

3.2.3 Finding Formal Parameters

Further, the list of sources is looped through, and each source is compared with every destination. If the source is not in the destination, then it is a unique value coming directly onto the stack externally. This is considered to be a parameter since it has been directly used as a source without any assignment. It is usually in the form of EBP + (n>=8). It is done at step 2

in the pseudocode shown in Table 3.1. The idea is based on stack memory management and the calling convention, which has already been discussed in Chapter 2. While looping, the value is appended in the *final_res* list as shown in Figure 3.4. Then the unique values are fetched by the 'set' operation to avoid getting the same value twice from a particular iteration within the loop. Further, if a call is made to the function *get_pc_thunk*, then the address of the next instruction is stored in the called register which is also added in the *final_res*. By taking the length of the set of *final_res* we get the number of parameters and their values as output from the algorithm which is:

no. of parameters is 3 {'0xc%ebp', '0x10%ebp', '0x8%ebp'}

3.2.4 Creating Flowlist

As shown in Table 3.1, we create a flowlist in step 3. This will determine that at every instruction, all the expressions, which must have already been computed up until that instruction, may not get modified later for any of the data flow paths. This is used to avoid re-computation and is an important step in the development of the algorithm where at each point(instruction), we are fetching the address of all the possible data flow targets. Thus, flowlist states all the possible data flow paths and in our case, it is the list of sets of all possible initial and target indices (ℓ, ℓ') of every instruction. We can say that flowlist **W** is a list of pairs where each pair is an element or possible indices of the flow of the function ƒ which is an element of the set of functions **F** in a program. The presence of a pair in the worklist indicates that the analysis has changed at the exit of (or entry to – for backward analysis) the statement labelled by the first component. Thus, it must be recomputed at the entry to (or exit from) the block labelled by the second component. In this process also, the types of instructions play an important role. For the range of function *start* (f[init]) and *end* (f[final]), we again check for different instructions. If the label is the first instruction, we move to the next label by adding 1 to the value of the label which gives us the next instruction. The idea is that usually, a function starts either with *'ebp'* stack based or with no operations operands like *nop, endbr32, endbr64,* etc. Then we check the type of instruction and take the value of *'ctrltarget'* of every source

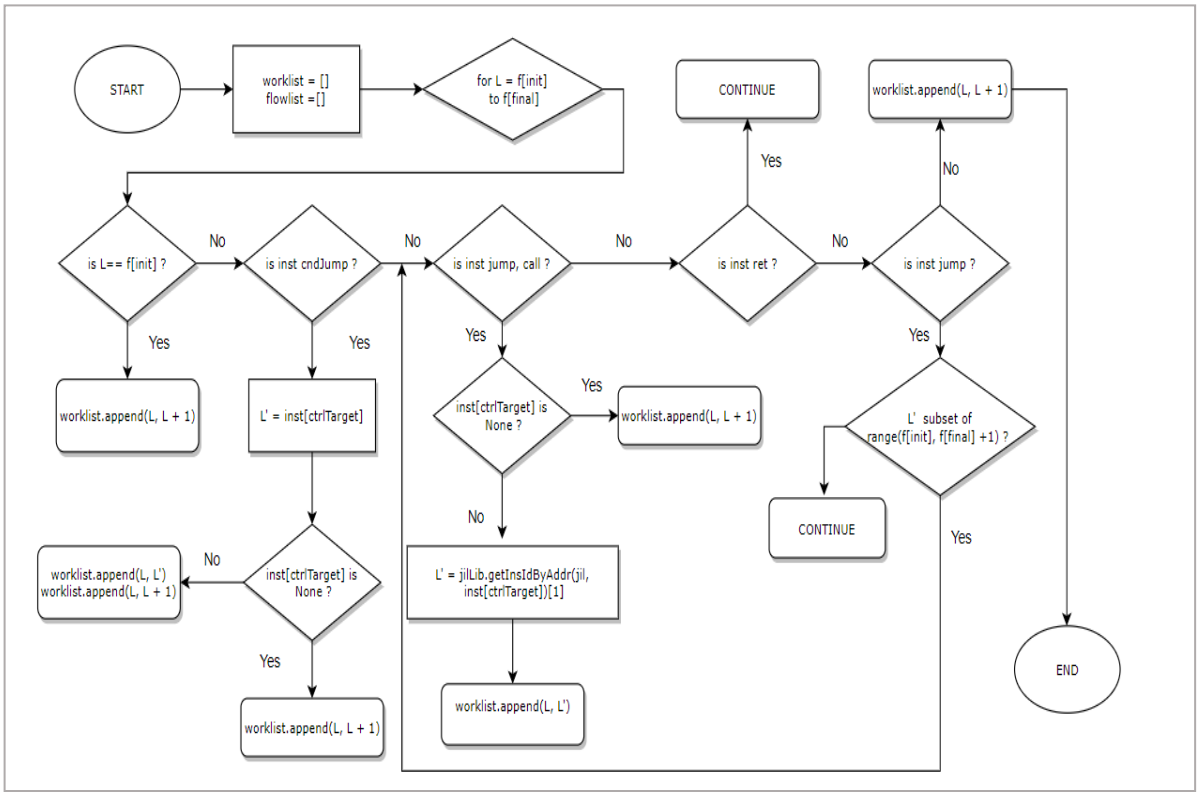

Figure 3.5: Data Flow diagram for creating a worklist.

and destination as we did in step 1. If the instruction is '*jump'*, *'call'* or *'cndjmp'*, we need to verify the value of the ℓ'. This ℓ' could be either the next instruction or any other memory location out of the scope of that function, and thus we either add the data flow path in our *'worklist'* or *continue* with the next instruction in the loop. For example, in Figure 3.3, for a conditional jump at label ℓ 3107, it has two possible data flow paths. It can jump either to the label ℓ' 3117 or to 3108 depending on the condition set being true or not. If the condition is true or values are equal, it will go to 3117 which we get from the value of *'ctrltarget'* of the label 3107 and if it is not equal, it will simply go to the next instruction label ℓ' 3108. Further, if the instruction is a call like at label 3132 which calls the function at ℓ' 85 which is out of the scope of the function boundary level. Hence, we simply continue with the very next instruction. Similarly, if the instruction is a *'ret'*, then we directly continue through the next label as shown in Data flow diagram in Figure 3.5. Hence, instructions like *'jump'*, *'cndjmp'*, *'call'*, etc. change the control flow of the program and capture all possible states of the program. The worklist we got from the algorithm for the program given in Figure 3.3 is –

flow list is [(3095, 3096), (3096, 3097), (3097, 3098), (3098, 3099), (3099, 3100), (3100, 3101), (3101, 3102), (3102, 3103), (3103, 3104), (3104, 3105), (3105, 55), (3106, 3107), (3107, 3117), (3107, 3108), (3108, 3109), (3109, 3110), (3110, 3111), (3111, 85), (3112, 3113), (3113, 3114), (3114, 1510), (3115, 3116), (3116, 3160), (3117, 3118), (3118, 3119), (3119, 3120), (3120, 3121), (3121, 3122), (3122, 3123), (3123, 3124), (3124, 3125), (3125, 64), (3126, 3127), (3127, 3128), (3128, 3138), (3128, 3129), (3129, 3130), (3130, 3131), (3131, 3132), (3132, 85), (3133, 3134), (3134, 3135), (3135, 1510), (3136, 3137), (3137, 3160), (3138, 3139), (3139, 3140), (3140, 3141), (3141, 3142), (3142, 3143), (3143, 43), (3144, 3145), (3145, 3155), (3145, 3146), (3146, 3147), (3147, 3148), (3148, 3149), (3149, 85), (3150, 3151), (3151, 3152), (3152, 1510), (3153, 3154), (3154, 3160), (3155, 3156), (3156, 3157), (3157, 3158), (3158, 3159), (3159, 2923), (3160, 3161), (3161, 3162), (3162, 3163)]

3.2.5 Final Analysis for Every Label

In this section, we will find the analysis and then the final analysis of every label ultimately up until that index within the range of the function starts and that instruction itself using the flowlist, as shown in example 3.1.2. We also create worklist which is a copy of flowlist and it will keep on changing during the iteration.This is done by using the RD analysis algorithm, explained in detail in section 3.1.2, which helps us to solve the $kill_{RD}$ and gen_{RD} functions while iterating through the statements. This constitutes steps 4 and 5 of Table 3.1. The method employs ‘analysis’ as a dictionary data structure that holds the input data for any label. Control flow analysis conducted by reaching definition serves as the input data for each label of an elementary block. In order to find the analysis, we will initialize all the labels with an empty set. By looping through, we check that if ‘label’ is the first index then we will set the analysis of that as (‘null’, ℓ). For the code in figure 3.3, label 3095 is set as (‘null’,3095). The analysis of a label is the values assigned to a variable up until that particular instruction. It serves as input or known information for the next label. For example, when ℓ is equal to 3108, then ℓ’ as per the worklist would be 3109. Initially, the analysis of 3108 would be an empty set() as discussed in the dataflow diagram. Then, we move further to verify whether it is any instruction out of *mov, add, sub,* or *lea*. If it is any one of the *mov, add, sub,* or *lea*, then we create a set

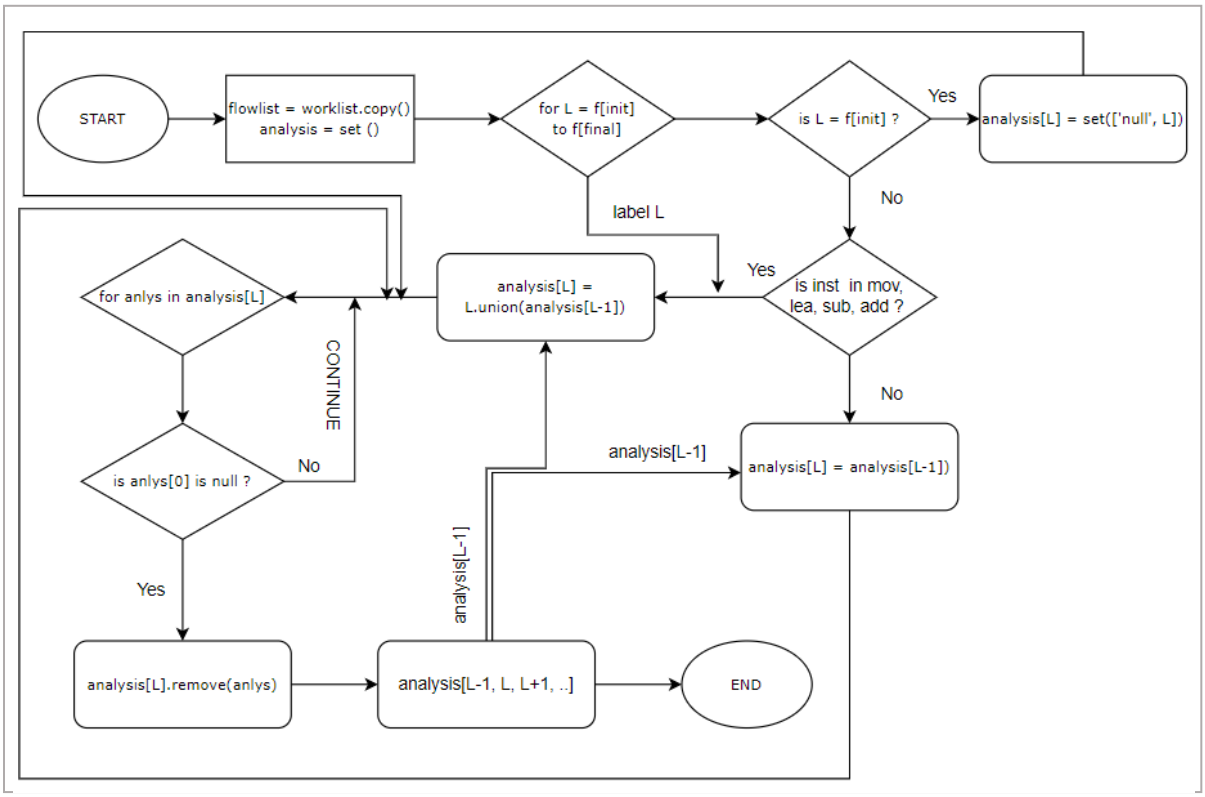

Figure 3.6: Data flow Diagram to calculate analysis of every label.

by doing a union of the current label and analysis of the previous label. This is shown in the dataflow diagram in Figure 3.6. Thus, for the label *'804b827'* in Figure 3.3, we have analysis as **{('0x4(%esp)', 3103), ('%eax', 3108), ('%ebp', 3096), ('%esp', 3098), ('(%esp)', 3104), ('%eax', 3101), ('0x8(%esp)', 3102), ('%edx', 3099), ('%eax', 3100)}**. If it is not any one of the *mov*, *add*, *sub*, *lea* instructions as mentioned before, then the analysis of the current label will be the same as the previous one. The instruction with label 3109 is *'movl'*, where the source is a pointer, as the type of this argument is *'mem'* and the destination is a register *'%ebx'*. Hence, we do the union of the set *('%ebx',3109)* and the previous analysis, i.e. of label 3108. Thus, we finally get the analysis of 3109 as **3109: {('0x4(%esp)', 3103), ('%ebx', 3109), ('%eax', 3108), ('%ebp', 3096), ('%esp', 3098), ('(%esp)', 3104), ('%eax', 3101), ('0x8(%esp)', 3102), ('%edx', 3099), ('%eax', 3100)}**. The number of sets keeps on increasing for the newer labels with the iterations. In this step also, the type of value is important to fetch the exact register or memory offset as discussed in section 3.2.2. All the labels are defined in the dictionary data structure which has key as labels and values as the list

of these sets. After iterating the function for the first time, we then loop through the analysis to see if there are any null sets. In the analysis[3109], label 3095 has a null value and hence, we will remove it since the null set means no information is present about that label and is of no use to keep for future purposes. It will make the analysis redundant, complex, and time-consuming to perform. The fragment of the analysis from the algorithm is –

analysis {3095: {('null', 3095)}, 3096: {('%ebp', 3096)}, 3097: {('%ebp', 3096)}, 3098: {('%esp', 3098), ('%ebp', 3096)}, 3107: {('0x4(%esp)', 3103), ('%ebp', 3096), ('%esp', 3098), ('(%esp)', 3104), ('%eax', 3101), ('0x8(%esp)', 3102), ('%edx', 3099), ('%eax', 3100)}, 3108: {('0x4(%esp)', 3103), ('%eax', 3108), ('%ebp', 3096), ('%esp', 3098), ('(%esp)', 3104), ('%eax', 3101), ('0x8(%esp)', 3102), ('%edx', 3099), ('%eax', 3100)}, 3109: {('0x4(%esp)', 3103), ('%ebx', 3109), ('%eax', 3108), ('%ebp', 3096), ('%esp', 3098), ('(%esp)', 3104), ('%eax', 3101), ('0x8(%esp)', 3102), ('%edx', 3099), ('%eax', 3100)}}

Next, the analysis calculated previously leads to the introduction of the data flow equation which served as the basis of our algorithm. Thus, we require RDA to develop the algorithm of data flow equations. The general iterative algorithm for the data flow path states that for any framework (in our case it is a function) we can find the analysis at each label (index) by doing forward and backward analysis of the data flow. It states all the possible values that a register or a memory address can hold up till that instruction. If any value gets modified in any of the future statements, the analysis also gets updated at that label. Forward analysis means the forward flow of statements(S_*). This is step 5 of the pseudo-code and uses both worklist and analysis as input to calculate final_analysis for every label. It starts by iterating the worklist until it is not empty. Firstly, the first element of the worklist set is popped from which we get our first set of labels, i.e. *ℓ and ℓ'.* As an example, we have *ℓ and ℓ'* as 3109 and 3110. Then iterating over the analysis[ℓ], a condition is checked where the current value of the loop (which is an element of analysis[ℓ]) is compared with the destinations[ℓ]. In this condition, we fetch every element *'anlys'* of analysis[3109] and, we check if the destination[3109] is equal to that element or not. If they are not equal, then we add the element from the current element from

analysis[ℓ] to final_analysis[ℓ]. Otherwise, we need the $kill_{RD}$ function to remove that value. Finally, we add the new set of elements having values (destinations[ℓ], ℓ) by using gen_{RD} function. We find that at label 3109, the value '%eax' register is holding gets reassigned again by a parameter *'0x8(%ebp)'*. This means that at index 3109, a new value is generated for the register *'%eax'* and is not overwritten for any previous value assigned to the register '%eax' anywhere before in the control flow. Hence, we get the 'if' condition true and a value is generated at label 3109 in the analysis by the generator function. Similarly, we either add or remove the values if generated or overwritten, respectively, by $kill_{RD}$ and gen_{RD} functions for the remaining analysis of the labels iterating with the help of with the control flow analysis and storing it in in a new dictionary data structure called final_analysis.

Further, we remove the null set elements from the final_analysis dictionary having key as the current label in the current loop of the worklist since we might encounter these null values again due to the repeated iteration. At this stage, we get the final_analysis of 3109, however, we are still calculating the analysis and final_analysis of ℓ' which is 3110. Then, as described in the dataflow diagram in Figure 3.7, we will check if the final_analysis[ℓ] is a subset of analysis[ℓ'] or not. If it is not a subset, we update the analysis[ℓ'] by the union of it with the final_analysis[ℓ]. Here, we will compare whether the final_analysis[3109] ⊆ analysis[3110]. The main idea is to verify that the analysis of 3110 has all the reaching definitions up until the instruction label 3109, including the analysis of the label 3110 itself. If it is not a subset, it means we are missing some information up until that label. Therefore, we update the set of analysis[3110] by assigning the union of the final_analysis[3109] and analysis[3110]. Thus, we obtain the final_analysis in the next iteration where the ℓ will be 3110. If the last condition is true, then we find the list of all possible ℓ'' (all possible paths for ℓ') by iterating over the flowlist which is the copy of the worklist initially obtained at step 3. Finally, we update the worklist by appending all the ℓ''. Here ℓ'' is 3111 and we generally append this in the worklist. However, in our case, final_analysis[3109] is a subset analysis[3110] and it will not get appended in the worklist. Thus, the next set which gets appended in the worklist is **working list after appending [(3111, 85), (3112, 3113), (3113, 3114), (3114, 1510), (3115, 3116), (3116, 3160), (3117, 3118), (3118, 3119), (3119, 3120), (3120, 3121), (3121, 3122), (3122, 3123), (3123, 3124), (3124, 3125), (3125, 64), (3126, 3127), (3127, 3128), (3128, 3138),**

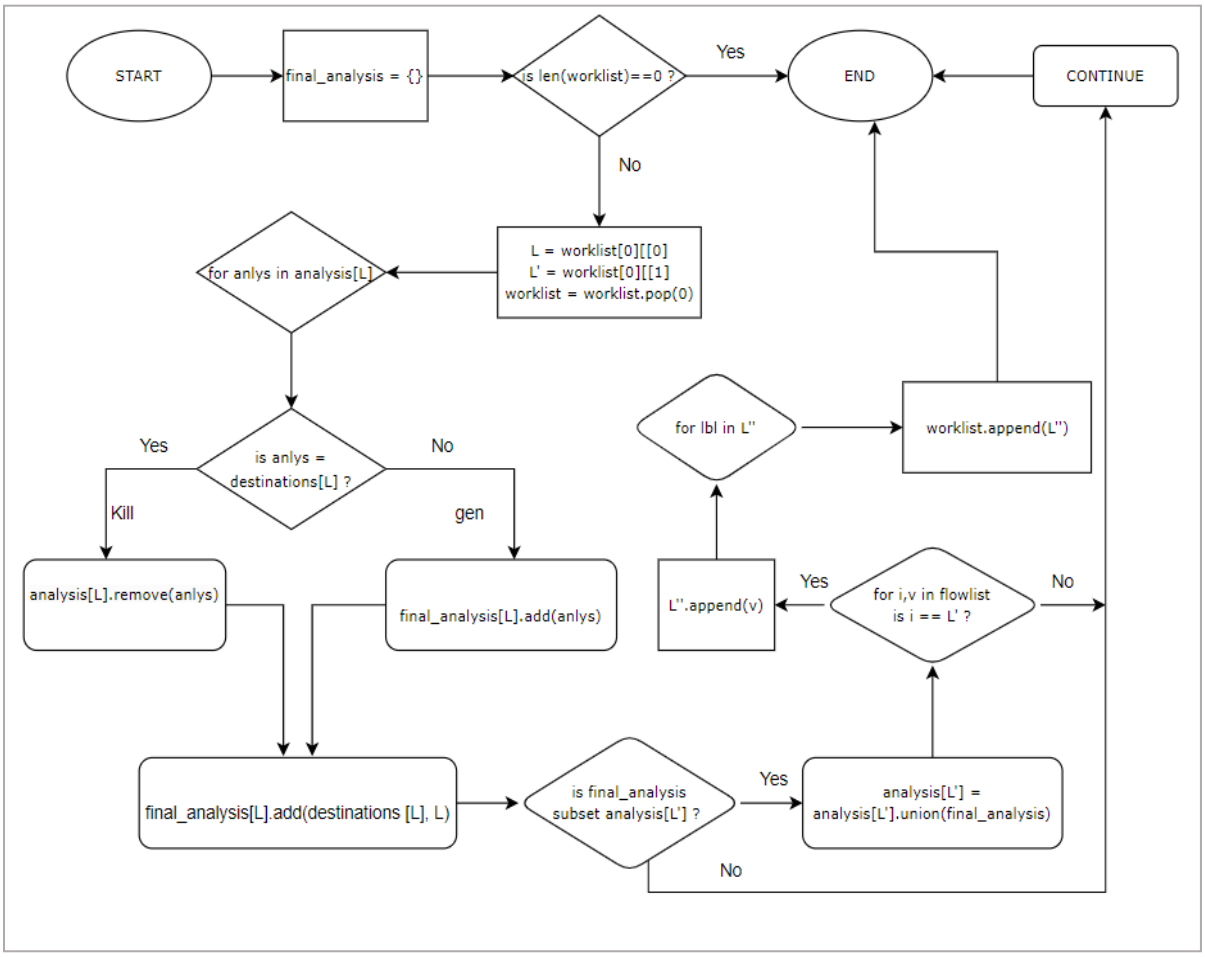

Figure 3.7: Data flow Diagram to calculate final analysis of every label.

(3128, 3129), (3129, 3130), (3130, 3131), (3131, 3132), (3132, 85), (3133, 3134), (3134, 3135), (3135, 1510), (3136, 3137), (3137, 3160), (3138, 3139), (3139, 3140), (3140, 3141), (3141, 3142), (3142, 3143), (3143, 43), (3144, 3145), (3145, 3155), (3145, 3146), (3146, 3147), (3147, 3148), (3148, 3149), (3149, 85), (3150, 3151), (3151, 3152), (3152, 1510), (3153, 3154), (3154, 3160), (3155, 3156), (3156, 3157), (3157, 3158), (3158, 3159), (3159, 2923), (3160, 3161), (3161, 3162), (3162, 3163), (3097, 3098), (3105, 55), (3107, 3117), (3107, 3108), (3117, 3118), (3108, 3109), (3111, 85)] for ℓ as 3110, ℓ' as 3111 and ℓ'' as 85. Therefore, the worklist gets modified and (ℓ',ℓ'') gets appended. This whole process continues until the worklist is empty. At last, we get the final_analysis of all the labels within the range(f[init], f[final+1). In general, we can say that the final analysis of a label ℓ so obtained after the completion of step 4 & 5 is the information of all the variables which is changed or

modified by some operation in the possible dataflow paths. The fragment of the final analysis of the function as shown in Figure 3.3 is –

final_analysis {3095: set(), 3096: {('%ebp', 3096)}, 3107: {('0x8(%esp)', 3102), ('%edx', 3099), ('0x4(%esp)', 3103), ('%eax', 3106), ('%esp', 3098), ('%ebp', 3096)}, 3108: {('0x8(%esp)', 3102), ('%eax', 3108), ('%edx', 3099), ('0x4(%esp)', 3103), ('(%esp)', 3104), ('%esp', 3098), ('%ebp', 3096)}, 3109: {('0x8(%esp)', 3102), ('%eax', 3108), ('%edx', 3099), ('0x4(%esp)', 3103), ('(%esp)', 3104), ('%esp', 3098), ('%ebx', 3109), ('%ebp', 3096)}, 3110: {('0x8(%esp)', 3102), ('%eax', 3108), ('%edx', 3099), ('0x4(%esp)', 3103), ('(%esp)', 3110), ('%esp', 3098), ('%ebx', 3109), ('%ebp', 3096)}, 3111: {('0x8(%esp)', 3102), ('%eax', 3108), ('%edx', 3099), ('0x4(%esp)', 3103), ('(%esp)', 3110), ('%esp', 3098), ('%ebx', 3109), ('%ebp', 3096)},…. }

3.2.6 Finding Parameter Used as Pointer

To find the parameter used as a pointer, we first need to check for the instructions having ‘args’, i.e. source or destination. In order to fetch a pointer, we only need the statements that involve indirect accessing of the memory to dereference the memory address by an offset. For this, we iterate over the function's range to get the value of the desired memory address. The statements that only have arguments of type "mem" are tested to see if the condition is true. Let us call this "*mem*" argument as “*temp*”, a location which could be either a source or destination of the current instruction where both are tested individually. We then determine whether any argument's value out of both is present in parameters; if so, we can treat that argument as a local argument. Otherwise, looping through all the ℓ’ in the flowlist and considering it as the current instruction label ℓ, we can traverse back to fetch the index from where we got ℓ' initially. We can call this label ℓ^*. This is called backward analysis and the pair of labels now are (ℓ^*, ℓ) which is equivalent to (ℓ, ℓ’) in the forward analysis, as discussed in section 3.1, where ℓ^* is the last, ℓ is the current and ℓ’ is the next instruction. By traversing through the final_analysis of every ℓ^*, we compare the source or destination of that label

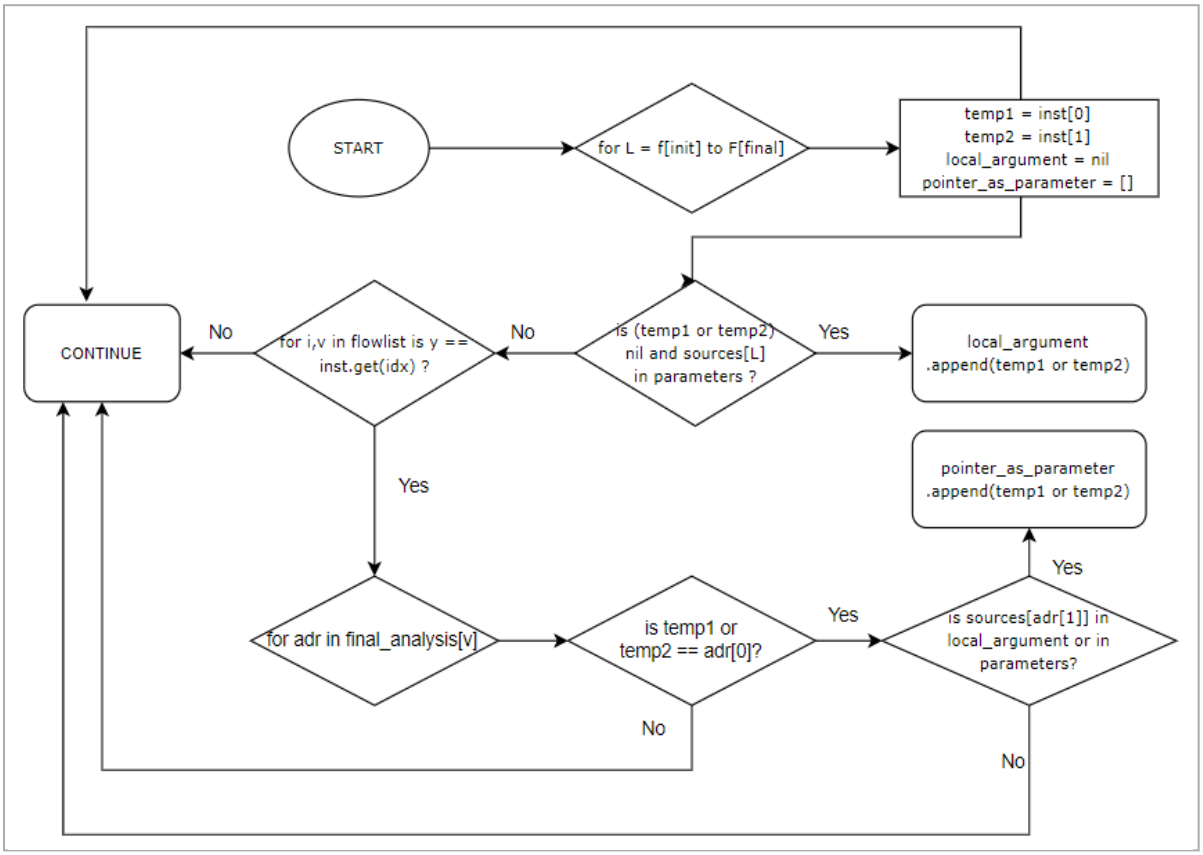

Figure 3.8: Data flow diagram for finding the pointer as parameter used as pointer.

with the value in final_analysis for the key ℓ* up until the first instruction. This means we test whether the sources[ℓ*] or temp_location is equal to final_analysis[ℓ*]. If it is true and this temp exists in the list of parameters or local arguments, then we say that it is a parameter used as a pointer. In Figure 3.3, the instruction having label ℓ as 3109 at '804b82a' accesses memory indirectly at the source argument. This means the temp is '%eax', ℓ* is 3108 and the source of 3109 is equal to the final_analysis of 3108. Hence, the last value assigned for '%eax' is coming from the source of ℓ*(3108), i.e., 0x8(%ebp) which is a parameter and hence, is a pointer.

Therefore, we have finally performed the binary static analysis on the function given in Figure 3.3. We found the parameters as *{'0x8%ebp', '0x10%ebp', '0xc%ebp'}* which are ***arch**, **offset** and ***nested_arch** respectively; local parameters *{'-0xc(%ebp)'}* which is **size_t got**; finally, at 3109, 3130 and 3147, we get *(%eax)* as parameters used as a pointer.

Note: In this example, out of the two parameters as pointers which are **arch* and **nested_arch*, only **arch* is used as a pointer inside the function, whereas **nested_arch* was used as a parameter to another function **get_archive_member_name** and is not used as **pointer.** Thus, the only parameter used as pointer inside the function is ***arch** which we are able to identify.

Similarly, while finding pointers used as parameter, we find local variables by backward analysis. Since the local arguments are calculated in the form of EBP - (xN>=8) for O0 levels and of the form ESP ± xN where Xn< stack depth for O2 level. This is done by looking at the source and destination of that instruction individually. If the origin of the any source or destination of any *'mov'* instruction is a parameter or a local variable (we do the back tracing as we did while finding the pointers) then it is a local variable. We can see this in step 6 in Table 2. The output we get from the algorithm is:

local arguments are 1 {'-0xc(%ebp)'}

3.3 Rules

In the development of this algorithm, there are a few rules which need to be adhered to while coding, otherwise, it will lead to discrepancies in the final result.

- Consider all the registers having similar functionality to be the same for 16-bit, 32-bit and 64-bit registers. Since these registers have different names, the algorithm will evaluate the same registers differently and results in discrepancies in the final output. For example, *%eax* and *%rax* are the same but since the names are different, they might be considered different registers which leads to wrong results. Hence, during the code development, we created a register list having the same functionality. Before assigning the value of a given register to any variable, it is checked in the list which tells us if the same or an equivalent register name is used before. For example, the lists created in the code are:

```
same_reg1 = ['%ax', '%al', '%eax', '%ah', '%rax']
same_reg2 = ['%bx', '%bl', '%ebx', '%bh', '%rbx']
same_reg3 = ['%cx', '%cl', '%ecx', '%ch', '%rcx']
same_reg4 = ['%dx', '%dl', '%edx', '%dh', '%rdx']
 same_reg5 = ['%edi','%rdi','%di']
```

- Every time an instruction argument is checked or calculated; it is checked in three ways based on its type. If they are ‘reg’ means it is a register, ‘imm’ gives an immediate value and if it has ‘mem’ keyword in the type, it states a memory location and needs to be dereferenced. Also, if it is ‘mem’ type, then the calculation of an offset is different based on the null values in ‘base’, ‘scale’ or an ‘index’.

```
if inst.get('args')[0].type == 'reg' and reg not in not_reg:
   sources.append(inst.get('args')[0].reg)
elif inst.get('args')[0].type == 'memOffsetBase':
   if str(inst.get('args')[0].base) not in not_base_index:
       sources.append(hex(inst.get('args')[0].offset) +
       inst.get('args')[0].base) if  inst.get('args')[0].offset != 'None' and
       int(inst.get('args')[0].offset)>0 else
       sources.append(inst.get('args')[0].base)
  elif str(inst.get('args')[0].index) not in not_base_index:
       sources.append(hex(inst.get('args')[0].offset) +
       inst.get('args')[0].index) if inst.get('args')[0].offset != 'None' and
       int(inst.get('args')[0].offset)>0 else
       sources.append(inst.get('args')[0].index)
else:
       pass
```

- We created a list where we should not consider certain registers as a part of the analysis as they are more of general-purpose registers such as a base pointer, instruction pointer, etc. Even though they are actual registers, in order to be aligned with the approach of our algorithm, they are not considered to be useful and hence we skipped them.

```
not_reg = ['%ebp', '%rbp', '%esp', '%rsp', '%rip', '%eip']
 not_base_index = ['None', '%eip', '%rip','%esp','%rsp']
```

- A JIL file created from a C source file has many functions involved. We need to loop through all the potential functions one by one and then this algorithm is applied. However, for testing purposes, we can pass the direct ranges or addresses of a function

with *start* and *end* addresses also. But the JIL file is not directly in human-readable format and hence, we need to either use *pprint* library or parse the file itself which is discussed in section 2.4.

- For 32-bit, there is a need to adhere to a special function called *'get_pc_thunk'* which is already incorporated in the algorithm since it uses a register where the next instruction value is stored. Initially, when it was not incorporated, it led to some erroneous results.

- For every call, the return value is in register '*%eax'* and for O2 levels also, *'%eax'* serves as special instruction for *'ret'* instruction.

- This algorithm only works for Python > 3.x version. If the algorithm ran on python version less than 3.x, then it will call the sys.exit function.

- Apart from the common instructions like '*mov, push, add, sub'*,etc., there are other commands also. They are taken care of by having an '*else'* clause in every branch of the algorithm. The approach for these commands is similar and since the testing is still going on, we continue to update the algorithm if any different scenario arises.s

Thus, the implementation of this algorithm requires certain considerations to get the correct output. Note, these are limited to Intel only as other ISA may require some different strategies or rules. And since we are limiting ourselves with the use of JIL data structure file while analyzing objdump file or any other might not involve these checks or limitations.

Chapter 4 : Experiments and Results

In Chapter 3, the algorithm was presented and explained in detail with the help of an example. In this chapter, we will give the details of the experimental results for detecting the parameters, local variables and pointer used as parameter of a function for binutils, coreutils and findutils stripped binaries compiled on GCC, Clang compilers. The results were verified manually and have been tested in over 650 test cases. In our analysis, we sometimes found discrepancies in the source code function declaration and function definition. Similarly, our analysis has sometimes overapproximated the final results by considering a 64-bit parameter into two 32-bit parameters which results in few false positives too. These kinds of issues provide us with limitations which have been talked about in detail in Chapter 5.

4.1 Ground Truth

The ground truth for each stripped binary and its function has been obtained by the data in ground truth file which consists of start and end addresses of a function, size and the function prototype, etc., of the source code. We have compared the ground truth of our results with the source code as well ground truth file generated in our JIMA toolkit. To get this ground truth file, we need to run some packages individually which are a part of the JIMA. These are py_dwarf.py and makeDwarfSym.py that gives us the header file, text file, JSON, etc. Further, it is cross verified manually by checking the source code of that particular file. Considering this file as a ground truth, the results are evaluated. Since the results we get from this algorithm is compared with the ground truth, it is mandatory to note that there can be discrepancies in the ground truth as well.

Example of Discrepancy: In stripped binaries, we do not have any debugging information and hence source code makes the ground truth. There are times when the listed function prototype or declaration has a different number of formal parameters as compared to the actual number of parameters used in the function definition. Some of the arguments are not even used in the function definition and are redundant to pass, which creates discrepancies in defining the actual formal parameters. In our analysis too, we encountered a similar issue in which the

```
bfd_boolean section_iterator_callback (bfd* abfd, asection* s, void* data)
```

Figure 4.1: Function declaration of a source code from gcc_binutils_32_O0_ld-new

```
08052679 <section_iterator_callback>:
 8052679:       pushl  %ebp
 805267a:       movl   %esp,%ebp
 805267c:       subl   $0x10,%esp
 805267f:       movl   0x10(%ebp),%eax
 8052682:       movl   %eax,-0x4(%ebp)
 8052685:       movl   -0x4(%ebp),%eax
 8052688:       movl   (%eax),%eax
 805268a:       testl  %eax,%eax
 805268c:       je     805269f <section_iterator_callback+0x26>
 805268e:       movl   -0x4(%ebp),%eax
 8052691:       movl   $0x1,0x4(%eax)
 8052698:       movl   $0x1,%eax
 805269d:       jmp    80526ac <section_iterator_callback+0x33>
 805269f:       movl   -0x4(%ebp),%eax
 80526a2:       movl   0xc(%ebp),%edx
 80526a5:       movl   %edx,(%eax)
 80526a7:       movl   $0x0,%eax
 80526ac:       leavel
 80526ad:       retl
```

Figure 4.2: Binary code definition of the function declaration given in Figure 4.1

static analysis of the source code could not identify unused attributes. However, when the binary static analysis was done for the function shown in Figure 4.2, it was found that fewer parameters were actually used or called than the number of parameters which were passed in the definition of source code as shown in Figure 4.1. The function given in Figure 4.2, is the objdump obtained from stripped binary '*gcc_binutils_32_O0_ld-new*' which is in the ELF format. As the name suggests, it was compiled on a 32-bit GCC compiler with an optimization level of O0. As per the function definition, we can see that three parameters have been passed to a function but after doing the static binary analysis, it was found that only two parameters were actually used. After the analysis, only two parameters, at locations *0x10(%ebp)* and *0xc(%ebp)*, which are **data* and **s*, respectively, were found to be actually used. As we see, these types of issues cannot be identified using source code analysis alone and also create discrepancies with the ground truth. We see that binary analysis works better in determining

the relevant ground truth. However, little effort was made in the direction of binary analysis and even commercial tools, which analyze binary code, require debugging information and depend on symbolic execution. Thus, the algorithm developed, as explained in detail in the later sections, serves as the ultimate step in finding the function parameters.

The results are finally presented by the precision, recall, and F1 after comparing it with the ground truth for each of the operation, i.e. detecting the parameters, local variables and pointer used as parameter of a function. To calculate them, we need true negative, false positive, false negative, and finally true positive results of the parameters, local variables and parameter used as pointer in a function identification. These metrics are defined as follows for each of the operation:

- **True Negative** (TN) means the source code has not output for any of the operations of a function and even the algorithm presented it with no output.

- **False Negative** (FN) means that the source code has the correct number of outputs for any of the operations of a function but the algorithm does not detect it**.**

- **False Positive** (FP) means the source code has no output for any of the operations of a function but the algorithm detects some output**.**

- **True Positive** (TP) means that the source code has the correct number of output for any of the operations of a function and the algorithm also detects the correct output.

Precision (P): For comparing the ground truth with the output of our algorithm, we need to know when the algorithm has correct results. Precision means the percentage of reported operations on a funcion that are correctly reported and are 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.

$$\boldsymbol{P} = \frac{|\mathbf{TP}|}{|\mathbf{TP}| + |\mathbf{FP}|}$$

Recall (R): Recall denotes the percentage of true or correct operations on a function detected

by the algorithm. The recall is the ratio of the true positive as the numerator and the sum of true positive and false negative results as the denominator.

$$\boldsymbol{R} = \frac{|\mathbf{TP}|}{|\mathbf{TP}| + |\mathbf{FN}|}$$

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 algorithm and the ground truth, addressing both false positives and false negatives.

$$\boldsymbol{F1} = \frac{\mathbf{2} * \mathbf{P} * \mathbf{R}}{\mathbf{P} + \mathbf{R}}$$

4.2 Datasets & Results

Different stripped binaries are chosen to obtain the results by finding precision, recall and F1 score. These are GCC 32-bit and 64-bit, both compiled by O0 and O2 optimization levels. The results for finding parameters and correctly identifying the number of parameters used as pointers are tested by the script giving the output in CSV format. This CSV file has the function start address, in both hexadecimal and decimal format, the number of parameters obtained from the algorithm, the function end address, the name of the parameters, the number of local variables, the name of the local variables and list of pointers as parameter which is a set of the name of the memory location and the address where it is found. The CSV is then compared with the '.h' file, i.e. the header file for the parameter and pointers. But, for the local variable, it has been done manually, since we need to look for the actual number of local variables, which is not possible without the source code. Table 4.1 and Table 4.6 show the number of information about the data used to perform the experiments.

4.1.1 GCC 32-bit and 64-bit O0

The results from algorithm were tested on 144 functions compiled on 32-bit and 100 functions compiled on 64-bit for O0 optimization level. Table 4.2 shows the summary of the percentages of precision, recall and F1 score for finding the parameters, local variables and parameter used as pointer. As we can see in the Table 4.2, the precision obtained in 32-bit and 64-bit is 96.53%

	Data Set			
	ELF 32-bit		ELF 64-bit	
	gcc O0	gcc O2	gcc O0	gcc O2
Number of Binaries	3	2	2	2
Number of Functions	145	100	100	100
Avg. Number of Functions	48	50	50	50
Size of Stripped Binaries (MB)	6	6.9	6	8

Table 4.1: Characteristics of GCC Dataset

	Data Set % Precision, Recall and F1 for GCC O0 optimization level					
	ELF 32-bit			ELF 64-bit		
	Precision	Recall	F1	Precision	Recall	F1
Parameters	96.53	100	98.23	100	100	100
Local variables	91.17	96.24	93.6	96.20	95	95.81
Parameter used as a Pointer	92.60	83.4	88.1	98.53	93.05	95.71

Table 4.2: Summary of percentages of precision, recall and F1 for finding parameters, local variables and parameter used as a pointer in GCC O0 optimization level

and 100%, respectively for finding parameters which means that the algorithm gives the correct number of parameters for around 96.53% for 32-bit and 100% for 64-bit of the time. Similarly, the precision is calculated as 91.17% and 96.20% for local variables and 92.60% and 93.05% for parameter as pointer for 32-bit and 64-bit, respectively. Further, the recall is calculated as

	Data Set % Precision, Recall and F1 for GCC O2 optimization level					
	ELF 32-bit			ELF 64-bit		
	Precision	Recall	F1	Precision	Recall	F1
Parameters	99	100	99.5	97.92	96.91	97.41
Local variables	63.42	88.14	73.76	29.55	41.94	34.67
Parameter used as a Pointer	86.67	81.25	83.87	56.37	93.94	70.46

Table 4.3: Summary of percentages of precision, recall and F1 for finding parameters, local variables and parameter used as a pointer in GCC O2 optimization level.

100% for getting number of parameters in both 32-bit and 64-bit. Having 100% recall means how accurately the algorithm is able to find the number of parameters. It is also called true positive rate or sensitivity. In the similar manner, recall is also calculated for local variables as 96.24% and 95% and 83.4% and 93.05 for parameter used as pointer. Finally, F1 score is also calculated as 98.23% and 100% for finding parameters which means the tradeoff between precision and recall. Also, we got 93.6% and 95.81% for local variables and 88.1% and 95.71% for parameter used as pointer.

4.1.2 GCC 32-bit and 64-bit O2

The same calculations is done for the functions compiled for O2 optimization level on both 32-bit and 64-bit. The experiments were performed on 200 functions in total, accounting for 100 each for both 32-bit and 64-bit. Again, the precision, recall and F1 score are calculated for finding the parameters, local variables and parameter used as pointer. Since O2 is highly

Type	TN	TP	FN	FP	Precision	Recall	F1
Parameter	0	239	0	5	98	100	99
Local Variables	52	169	11	12	93.37	93.88	93.62
Parameter as Pointer	102	119	17	6	95.2	87.5	91.4

Table 4.4: Average % of different measures for different types for GCC O0 optimization level

Type	TN	TP	FN	FP	Precision	Recall	F1
Parameter	0	194	3	3	98.47	98.47	98.47
Local Variables	43	70	25	62	53.03	73.68	61.67
Parameter as Pointer	82	77	11	30	72	87.5	80

Table 4.5: Average % of different measures for different types for GCC O2 optimization level

optimized, we did not achieve very good results as compared to O0. Due to the high false positives rate which is observed in O2, especially for the local variables, the values of different measures are also dropped significantly as shown in Table 4.3. The reason behind high false positives is discussed in detail in the section 5.2. We see that the different measure values are highest in finding the parameters in both 32-bit and 64-bit. The precision is 99%, the recall is 100% and the F1 score is 99.5% for finding parameters in 32-bit. In 64-bit, these measures got dropped as compared to the 32-bit. Similarly, for local variables, the values were least for the different measures due to many reasons like padding variables, a 64-bit value gets divided into two 32-bit, some global variables, etc,. We observed the precision as low as approx. 64% and

	Data Set			
	ELF 32-bit		ELF 64-bit	
	clang O0	clang O2	clang O0	clang O2
Number of Binaries	2	2	2	2
Number of Functions	50	50	50	50
Avg. Number of Functions	25	25	25	25
Size of Stripped Binaries (MB)	1.3	0.94	1.4	1.2

Table 4.6: Characteristics of Clang Dataset

30% only, whereas the recall values are better which means we have more false positives than false negatives. We got 88.14% and 41.94% recall values for local variables for 32-bit and 64-bit, respectively. This also means that in 64-bit, we see that the optimizations are much higher due to enough memory. Further, we calculated parameter used as a pointer and found the precision, recall and F1 score values as 86.67%, 81.25% and 83.87 respectively for 32-bit whereas, 56.37%, 93.94% and 70.46%, respectively for 64-bit.

4.1.3 Final Analysis of GCC O0 and 02 Optimization Level

Table 4.4 shows the confusion matrix for each of the types we evaluated. Thus, the average precision, recall and F1 score were calculated for finding parameters, local variables and parameters as pointers for the O0 optimization level. We get average precision, recall and F1 score for parameters as 98%, 100% and 99% overall. Similarly, for local variables, 93.4%, 93.9% and 93.6% are observed; for parameters as pointers, 95%, 88% and 92% approximately are observed. We get the total functions on which the evaluation is done by adding true negatives, true positives, false negative and false positives which were defined before. Thus, the values are highest for finding parameters among the other two.

	Data Set % Precision, Recall and F1 for CLANG O0 optimization level					
	ELF 32-bit			ELF 64-bit		
	Precision	Recall	F1	Precision	Recall	F1
Parameters	96	100	98	100	100	100
Local variables	73.34	73.34	73.34	74.35	96.67	84.05
Parameter used as a Pointer	91.67	81.5	86.28	83.34	50	62.50

Table 4.7: Summary of percentages of precision, recall and F1 for finding parameters, local variables and parameter used as a pointer in Clang O0 optimization level

In a similar manner, Table 4.5 shows the confusion matrix for each of the types evaluated for O2 optimization level to calculate the average precision, recall and F1 score. We observe the F1 score is highest for finding the parameters and least for local variables which is the case with O0 level too.

4.1.4 Clang 32-bit and 64-bit O0

The results from algorithm were tested for LLVM compiler on 50 functions compiled on 32-bit and 50 functions compiled on 64-bit for O0 optimization level. Table 4.7 shows the summary of the percentages of precision, recall and F1 score for finding the parameters, local variables and parameter used as pointer. As we can see in the Table 4.7, the precision obtained in 32-bit and 64-bit is 96% and 100%, respectively for finding parameters which means that the algorithm gives the correct number of parameters for around 96% for 32-bit and 100% for 64-bit of the time. Similarly, the precision is calculated as 73.34% and 74.35% for local

variables and 91.67% and 83.34% for parameter as pointer for 32-bit and 64-bit, respectively.

	Data Set % Precision, Recall and F1 for Clang O2 optimization level					
	ELF 32-bit			ELF 64-bit		
	Precision	Recall	F1	Precision	Recall	F1
Parameters	94	100	97	97.87	93.87	95.82
Local variables	54.05	80	64.51	50	26.5	34.64
Parameter used as a Pointer	75	66.67	70.59	94.5	65.38	77.3

Table 4.8:Summary of percentages of precision, recall and F1 for finding parameters, local variables and parameter used as a pointer in Clang O2 optimization level.

Further, the recall is calculated as 100% for getting number of parameters in both 32-bit and 64-bit. Having 100% recall means that almost always the algorithm is able to find the number of parameters accurately. In the similar manner, recall is also calculated for local variables as 73.34% and 96.67% and 81.5% and just 50% for parameter used as pointer. Finally, F1 score is also calculated as 98% and 100% for finding parameters which means the tradeoff between precision and precision and recall. Also, we got 100% and 84.05% for local variables and 86.28% and 62.50% for parameter used as pointer compiled on 32-bit and 64-bit, respectively. The statistics clearly shows that the algorithm performed better on GCC O0 than Clang O0 optimization level. Also, the algorithm performed better in GCC O0 than Clang O0 as so many false positives are observed as clang considers every structure element as a new local variable.

4.1.5 Clang 32-bit and 64-bit O2

The same calculations is done for the functions compiled for Clang O2 optimization level on

Type	TN	TP	FN	FP	Precision	Recall	F1
Parameter	0	98	0	2	98	100	99
Local Variables	21	52	9	18	74.28	85.3	79.40
Parameter as Pointer	49	32	15	4	88.89	68.08	77.92

Table 4.9: Average % of different measures for different types for Clang O0 optimization level

both 32-bit and 64-bit. The experiments were performed on 100 functions in total, accounting 50 each for both 32-bit and 64-bit. Again, the precision, recall and F1 score are calculated for finding the parameters, local variables and parameter used as pointer. Since O2 is highly optimized, we did not achieve very good results as compared to Clang O0. Also, as compared to GCC O2, the algorithm did not performed well for Clang O2 especially for fining local variables. This is again attributed to the compiling technique used by clang especially with struct data structures which is discussed in section 5.2. Due to the high false positives rate which is observed in O2, especially for the local variables, the values of different measures are also dropped significantly as shown in Table 4.8. We see that the different measure values are highest in finding the parameters in both 32-bit and 64-bit and least for local variables. The precision is 94%, the recall is 100% and the F1 score is 97% for finding parameters in 32-bit. In 64-bit, these measures got dropped as compared to the 32-bit. Similarly, for local variables, the values were least for the different measures due to many reasons like padding variables, a 64-bit value gets divided into two 32-bit, some global variables, etc,. We observed the precision and recall as low as approx. 50% and 26% which drops the value of F1 score to 34.64% in local variables for 64-bit. This is because of the highly optimized compiler.

4.1.3 Final Analysis of Clang O0 and 02 Optimization Level

Table 4.9 shows the confusion matrix for each of the types we evaluated. Thus, the average

Type	TN	TP	FN	FP	Precision	Recall	F1
Parameter	0	93	3	4	95.9	96.9	96.39
Local Variables	14	30	30	26	53.57	50	51.72
Parameter as Pointer	49	31	15	5	86.12	67.4	75.62

Table 4.10: Average % of different measures for different types for Clang O2 optimization level

precision, recall and F1 score were calculated for finding parameters, local variables and parameters as pointers for the O2optimization level. We get average precision, recall and F1 score for parameters as 95.9%, 96.9% and 96.4% overall. Similarly, for local variables, 53.57%, 50% and 51.72% are observed; for parameters as pointers, 86.12%, 67.4% and 75.62% approximately are observed. We get the total functions on which the evaluation is done by adding true negatives, true positives, false negative and false positives which were defined before. Thus, the values are highest for finding parameters among the other two.

In a similar manner, Table 4.10 shows the confusion matrix for each of the types evaluated for O2 optimization level to calculate the average precision, recall and F1 score. We again observe the values are highest of finding the parameters and least for local variables which is the case with O0 level too. Also, the measures of local variables are the least among others.

Chapter 5 : Challenges and Limitations

This chapter discusses the challenges faced while developing the algorithm and the limitations of it. Since the objective of this research is to make a generalized approach for finding the number of parameters, local variables and parameter used as pointer, it is a challenge to incorporate all the requirements of different compilers for both 32-bit and 64-bit for different optimization levels.

5.1 Challenges

Numerous challenges were presented while analyzing parameters, local variables and pointers to develop the algorithm due to the several reasons.

Compiler Architecture: The way a compiler generates binary code can be influenced by its architecture, which might make it more challenging to search for local variables and arguments in stripped binaries. Due to this difference in a compiler's architecture, it becomes difficult to address all the conditions when a modification is made because one compiler might hamper the correct output when compiled on another platform. The registers used by various compilers to store local variables and function parameters may also vary. While evaluating stripped binaries, it can be difficult to determine which registers are being used for which purpose. Additionally, stack usage also plays an important role as depending on the compiler, different conventions are used for storing local variables and parameters on the stack. Due to this, it becomes very hard to understand how the stack is being used and which values belong to which variables. The calling convention used by a compiler can also impact the way a function parameter is given and how a return value is handled. Further, the optimizations by a compiler changes the way code is generated and thus, the structure of the code changes. This changed sequencing of instructions or code that seems superfluous also gets removed by compiler optimizations. This can cause unexpected behavior and make it more difficult to track the flow of data and the changed values. Therefore, it is very difficult to make a generalized approach for different compiler architectures in a single algorithm.

Some compilers will store local variables on the stack in the order as the source code while others may rearrange them. For instance, GCC and ICC write code utilizing various approaches and optimizations. As a result, they could handle local variables differently during code compilation. In order to assign registers for local variables, both of them employ various techniques, which results in various register usage patterns and performance characteristics of the produced code. Both of them have different optimization levels that can alter some inline small functions or remove unnecessary variables, resulting in different register usage patterns. Since they are different in architecture and the calling convention, the memory access can also vary as it depends on how the data structures are laid out in memory. It may use different data structure alignment and packing options, which can result in different memory usage patterns for local variables. The particular variation will depend on the specific code being generated and the optimization parameters employed by each compiler. To understand how various compilers are affecting local variable handling for a specific codebase, it is always a good idea to examine the resulting assembly code and run benchmarks. Due to the complexity of the differences in architecture, it is quite hard to identify the parameters, pointers as parameters, and local variables perfectly.

In both **32-bit and 64-bit compilers**, one of the main challenges is memory management during program execution. Since the stack size in a 32-bit compiler is restricted to 4GB, the memory for storing parameters and local variables is also a finite amount. This results in issues like programs that need a lot of memory. Sometimes, a 64-bit parameter passed in 32-bit becomes a challenge to recognize the actual parameter as it will get divided into two 32-bit values and thus, breaks the code. This results in many false positives while finding the number of parameters. On the other hand, 64-bit compilers may address up to 16 exabytes of memory, which is a substantially larger number (18,446,744,073,709,551,616 bytes) thus, making applications with more intricate data structures possible. However, due to enough memory, the parameters are directly used from the registers which makes it hard to capture the local variables and parameters on the stack during run-time correctly. Also, in 64-bit, *'%eax'* serves a special purpose where all the return addresses of calls made to different functions resides. Hence, this exception of *'%eax'* should also be handled carefully while designing the algorithm as it may result in many false positives. One such challenge occurred while analyzing the

pointers and the stack pointer is being referenced to the last time where *'%eax'* has assigned a value or has the return value from a function. Hence, during the analysis of pointers, it results in false positives.

Optimization levels: Different optimization level makes any analysis challenging because of how the compiler transforms and optimizes the code during compilation. The compiler produces code that closely resembles the original source code when optimization level O0 (i.e., no optimization) is used. This makes it simpler to recognize and locate parameters and local variables in the compiled code because the resulting code largely preserves the original structure and variable names. However, the compiler aggressively optimizes the code at optimization level O2 (i.e., high optimization), potentially affecting the code's structure and the way that parameters and local variables are used. It may be more challenging to identify and locate particular parameters and local variables in the produced code because of optimizations the compiler makes, such as inlining routines, removing unnecessary variables, and rearranging instructions. Moreover, the compiler may undertake aggressive register allocation and other optimizations at high optimization levels, which could result in variables being stored in registers rather than in memory. This can make identifying and locating parameters and local variables in the produced code much more difficult.

Furthermore, the **calling convention** of both optimization levels is different which makes it more complicated to develop a generalized algorithm. Since the compiler produces code that closely resembles the original source code at optimization level O0, it involves employing a straightforward calling convention where the caller is responsible for clearing the stack following the function call, such as the CDECL convention. This convention makes it simpler to debug the code because it is clear-cut and simple to comprehend. At optimization level O2, however, a complicated calling convention, such as the FASTCALL convention is used where some parameters are passed in registers rather than on the stack which later may be used by the compiler. This usually happens when the compiler knows the function is local and can optimize all the calls made to it. The code may run more quickly as a result, but the function call and parameter passing may not follow the expected pattern, making it harder to analyze. It is more difficult to understand the program's control flow and data flow because of the

function inlining at the O2 level, which completely eliminates the need for a function call by inserting the function code right into the calling code. This is because the way functions are called and how parameters are given get altered most of the time.

An example of this may be that in a 32-bit and 64-bit O0 optimization level, a compiler (especially the common ones like GCC, Clang, etc.) generally pushes the *'%ebp or %rbp'* first onto the stack followed by an instruction like *'movq %rsp,%rbp'* or *'movl %esp,%ebp'*. However, in the O2 optimization level, there is no general pattern that we observe at the start of a function in binary code. Generally, we see an instruction like *'subl $0x30, %esp'* meaning that it makes space on the stack by using *'%esp'* as a reference for local variables which includes some padding too. This padding gives some extra values which are not local variables compared with the ground truth and gives many false positives again. Other times, we see a direct push of *'%ebp'* or *'%rbp'* similar to in the O0 level. In addition to this, there are scenarios where we see *'push %ebx'* at the function start. It is uncommon that a function does not follow the *-fomit-frame-pointer* option by default. Hence, without a frame pointer, which generally happens in the case of optimizations, the compiler uses other registers to reference these values. O2 may use the *'%ebx'* register to reference the values within a function. But, in x86 systems, the *'%ebx'* register also gets utilized for position-independent code (PIC) that creates code that can be loaded at any address in memory. The O2 optimization level in GCC pushes the value of the *'%ebx'* register onto the stack at the start of a function and recovers it before returning to enable with PIC. By doing this, it is ensured that utilizing the *'%ebx'* register to reference values inside the function can be done without endangering the PIC. However, it is important to note that other compilers or optimization levels might not exhibit this behavior, which is unique to GCC's implementation of the O2 optimization level. In addition, depending on the particular code being produced, using the *'%ebx'* register may not always be the best option and alternative registers may be used in its place.

Finding parameters used as a pointer also presents some challenges especially in O2 level. One of the major challenge with stripped binaries is determining the places in memory where the pointers are placed are transferred between functions or allocated dynamically. Additionally, sometimes the current function calls another function which means that the caller

function does not necessarily executes the next instruction of the given function or the return address is stored in *'%eax'*. This disrupts the control flow of the current function and makes it harder to achieve the correct flow in the form of the workflow list. Therefore, it becomes hard to recognize the pointer used as parameter especially in the case where *'%eax'* is a parameter. Further, we found few instances where the base or frame pointer, *'%ebp'*, used as a regular register. For example, let us suppose the parameter has assigned value to the *'%ebp'* register, like in instruction *'movl 0x30(%esp), %ebp'*. Using this base register, the next instruction *'movzbl 0x7(%ebp), %ecx'* is used as a pointer to the parameter *'0x30(%esp)'*. Thus, this creates confusion in recognizing or analyzing the correct pointer since any value of the form *0xn('%ebp')* is used generally as parameters.

Therefore, analyzing the binary code statically is challenging as it depends on data flow analysis which can be limited in scope due to its complex nature. Since it is hard to decode the complex data structures or dynamic memory allocation, the algorithm presented sometimes gave false results as well.

5.2 Limitations

With respect to the development of the algorithm presented this section describes limitations in detail with the examples. Some of these limitations can be enhanced in future and are not implemented because of the time constraint.

- **Lack of Context:** Static analysis techniques do not execute the code, therefore they are unable to access the environment in which it is being used. It may be challenging to tell which functions are called in particular execution paths or how variables are used as a result. Factors such as input parameters, environment variables, system configuration, and the state of memory and file systems are included in the context of the program. For example, depending on its runtime context, a variable might be initialized to a particular value that isn't always clear from the code alone. Similarly, the behavior of a function may depend on the exact parameters provided to it, which may not be known

without executing the program. Due to a lack of context, findings in the analysis may contain false positives or false negatives. To alleviate this constraint, it may be essential to combine static analysis with other techniques, such as dynamic analysis, to acquire more full knowledge of the program's behavior.

- **Limitations due to Incomplete Code and Data Flow Analysis:** In stripped binaries, we may find some code that is executed only under specific conditions, certain type of inputs or system configurations. As a result, the insufficient code creates inaccurate data flow or control flow graphs, which may result in an incorrect output. As a result, the method becomes more prone to errors since code that employs sophisticated data structures or dynamic memory allocation may not be handled by the data flow analysis method.

- **No runtime or dynamic information:** Programs frequently allocate and deallocate memory at runtime using dynamic memory allocation techniques like malloc() and free(). Tracking of all the allocations and deallocations of memory makes it more difficult to understand how variables are used. Also, code is executed at runtime by programs using dynamic control flow techniques like function pointers or indirect jumps which makes it hard to understand how the code is being run and which functions are being called.

- **Code Obfuscation:** As with dynamic analysis, code obfuscation techniques can also hinder static analysis. Obfuscation can make it difficult to understand the code structure or to identify important functions and variables. We constantly require useful properties of a function that hold true across all executions. It usually gets combined with the complexity added by typical encryption and obfuscation methods for protecting binary code. The privacy-sensitive portions of an algorithm are typically encrypted to maintain privacy. Binary code is modified through obfuscation to make it more challenging to understand or deconstruct [7]. An example of obfuscation is shown in Figure 5.1 which shows a pseudo-code where obfuscation changes binary code to make it challenging to

```
const int constants1[32] = {...};
const int constants2[32] = {...};
int random = rand();
int c = 0;
for (int i = 0; i < 32; ++i) {
      if (some_func(random, i) == true) {
            c ^= constants1[i];
      } else {
            c ^= constants2[i];
      }
}
```

Figure 5.1: A pseudo-code showing obfuscation (Adapted from [7]).

understand or deconstruct. In the example, static analysis by itself is unable to detect the loop's execution route. To infer that integer *'c'* is not constant, we need an abstract representation of *'c'*.

- **Lack of Dynamic Analysis:** In dynamic analysis, the code is executed in a controlled environment setting and during the runtime, its behavior is observed. Memory leaks, runtime problems, and race situations are just a few examples of the faults that can be detected in this way but are challenging by using only static analysis. Moreover, dynamic analysis can be used to test the behavior of the software under various settings and scenarios, which can be used to validate the software's accuracy in a real-world setting. But, dynamic analysis tends to take a single path of execution flow using runtime information. Thus, it has its own restrictions which infers that having the hybrid strategy should be considered which includes the combination of both.

- **Architectural difference:** In our analysis, we found few cases in 32-bit compiled stripped binaries where a 64-bit parameter has been passed and gets divided into two 32-bit parameters. This gives some false positive results which is difficult to determine in the absence of the source code or debugging information. An example is shown in Figure 5.2 of a function named *'byte_put_little_endian'* from a standard library of binutils was tested. As we can see in the source code in Figure 5.3, this function has three parameters where the first one is a pointer too. But when we analyzed the function in the stripped binary, we found four parameters - {*'0x8(%ebp)', '0x14(%ebp)',*

```
void byte_put_little_endian (unsigned char * field, elf_vma value, int
size)
{
  switch (size)
    {
    case 8:
      field[7] = (((value >> 24) >> 24) >> 8) & 0xff;
      field[6] = ((value >> 24) >> 24) & 0xff;
      field[5] = ((value >> 24) >> 16) & 0xff;
      field[4] = ((value >> 24) >> 8) & 0xff;
      /* Fall through.  */
    case 4:
      field[3] = (value >> 24) & 0xff;
    case 3:
      field[2] = (value >> 16) & 0xff;
    case 2:
      field[1] = (value >> 8) & 0xff;
      /* Fall through.  */
    case 1:
      field[0] = value & 0xff;
      break;
    default:
      error (_("Unhandled data length: %d\n"), size);
      abort ();
    }
`
```

Figure 5.2: Source code where 64-bit parameter gets divided into two 32-bit parameters.

```
static int process_object (const char *file_name, FILE *file)
{
       long offset = ftell (file);
       if (! get_file_header (file))
              {
                     error (_("%s: Failed to read ELF header\n"),
              file_name);
                     return 1;
              }
       if (fseek (file, offset, SEEK_SET) != 0)
              {
                     error (_("%s: Failed to seek to ELF header\n"),
              file_name);
              }
       if (! update_elf_header (file_name, file))
              return 1;
       return 0;
}
```

Figure 5.3: An example of the limitation of intra-analysis

```
static int
elf_type (const char *type)
{

  if (strcasecmp (type, "rel") == 0)
             '0x4%esp'}
      return ET_REL.
  if (strcasecmp (type, "exec") == 0)
      return ET_EXEC;
  if (strcasecmp (type, "dyn") == 0)
      return ET_DYN;
  if (strcasecmp (type, "none") == 0)
      return ET_NONE;
  error (_("Unknown type: %s\n"),
type);
      return -1;
`
```

```
08049470 <elf_type>:
 8049470:    push1  %ebx
 8049471:    mov1   %eax,%ebx
 8049473:    sub1   $0x18,%esp
 8049476:    mov1
$0x804c411,0x4(%esp)
 804947e:    mov1   %eax,(%esp)
 8049481:    call1  8048c70
<strcasecmp@plt>
 8049486:    mov1   $0x1,%edx
 804948b:    test1  %eax,%eax
 804948d:    je     80494a8
<elf_type+0x38>
 804948f:    mov1
$0x804c415,0x4(%esp)
 .
```

Figure 5.4: An example code showing no local variables in source code (left side) and extra (false positives) local variables in binary code (right side)

0xc(%ebp)', '0x10(%ebp)'}. In the list, '0x8%ebp' is an extra since **field* parameter is a 64-bit parameter and got divided into 32-bit parameters. Thus, it gives extra parameters which leads to false positives. Similarly, there are few other functions such as byte_put_big_endian, byte_get_signed which comes under exceptional category.

Lack of Inter- function Analysis of a Pointer used as Parameter in a function: In this algorithm, we are dealing with the parameters which are used as pointers only in the scope, i.e., within a given function which is called intra-function analysis. But there were few cases where the parameter passed in the function was a pointer, however, it is not used as a pointer inside the scope of that function. The pointer parameter was passed to another function through a local variable and hence, we are not able to find this pointer parameter unless we do inter-function analysis. At this point of time, it is out of the scope of this thesis but is likely an enhancement in the future work. This non accessibility of inter-analysis gives us false negatives as we are not able to capture the pointer parameter even though it exists. As shown in Figure 5.3, the parameters '*file_name' and 'FILE *file' are passed either to the function 'ftell'

```
void freeargv (char **vector)
{
  register char **scan;

  if (vector != NULL)
    {
      for (scan = vector; *scan != NULL; scan++)
       {
         free (*scan);
       }
      free (vector);
    }
}
```

Figure 5.5: An example of limitation showing false negatives of local variables in case of faster access

or *'get_file_header'* directly and never really used as a pointer inside the function *'process_object'.* Hence, we need inter-analysis of functions where we can cater these types of limitations.

- **Extra Local Variables detected due to Optimizations and some Padding:** The compiler may carry out a number of optimizations to reduce the size and enhance the speed of the resultant binary when code is generated with optimization flags like O2. Register allocation is one optimization that the compiler might carry out that can speed up code execution, is the procedure by which the compiler assigns variables to particular hardware registers in the processor. The compiler might, however, occasionally need to allocate more registers than the hardware can support. To hold the values of the variables that cannot be kept in registers in this situation, the compiler may generate new local variables in the function. These new temporary local variables do not usually exist in the original source code, but are often generated by the compiler during compilation. In addition to that, sometimes compiled code does not do a push to pass to the next function. Instead, it preallocates space and uses it to reference the places to pass to the called functions. Also, some of these preallocated locations are due to the padding and as a result, in O2 optimization levels a high number of test cases resulted in extra local parameters. In Figure 5.4, the right side shows such a binary function where we see that stack is allocating space for function call variables, padding

```
static bfd_boolean
coff_read_word (bfd *abfd, unsigned
int *value)
{
  unsigned char b[2];
  int status;

  status = bfd_read (b,
(bfd_size_type) 2, abfd);
  if (status < 1)
    {
      *value = 0;
      return FALSE;
    }

  if (status == 1)
    *value = (unsigned int) b[0];
  else
    *value = (unsigned int) (b[0] +
(b[1] << 8));

  pelength += (unsigned int) status;

  return TRUE;
}
```

```
0804a6f3 <coff_read_word>:
 804a6f3:    pushl  %ebx
 804a6f4:    movl   %edx,%ebx
 804a6f6:    subl   $0x28,%esp
 804a6f9:    movl   %eax,0xc(%esp)
 804a6fd:    leal   0x1e(%esp),%eax
 804a701:    movl   $0x2,0x4(%esp)
 804a709:    movl   $0x0,0x8(%esp)
 804a711:    movl   %eax,(%esp)
 804a714:    calll  807e540
<bfd_bread>
 804a719:    testl  %eax,%eax
 804a71b:    jg     804a727
<coff_read_word+0x34>
 804a71d:    movl   $0x0,(%ebx)
 804a723:    xorl   %eax,%eax
 804a725:    jmp    804a74c
<coff_read_word+0x59>
 804a727:    cmpl   $0x1,%eax
 804a72a:    movzbl 0x1e(%esp),%edx
 804a72f:    jne    804a735
<coff_read_word+0x42>
 804a731:    movl   %edx,(%ebx)
 804a733:    jmp    804a741
<coff_read_word+0x4e>
```

Figure 5.6: An example showing false positives due to character array for local variables in GCC O2

or some extra variables to do optimizations, etc. Thus, we get the local variables as*{'0x8(%esp)', '0x4(%esp)'}* which are extra and give false positives, whereas there are no local variables defined in the source code as shown at the left-hand side of the figure.

- **False negatives in local variables due to use of registers:** We have encountered some cases where the local variable is declared with a register keyword for faster access. Since the local variable is now a register, it leads to confusion among a parameter or local variable or a return value register. Thus, in these types of cases, the algorithm does not locate the local variable correctly and gives false negatives. An example is shown in Figure 5.5 where we can see in the source code that the local variable named ‘scan’ is declared and is a pointer to a pointer to a character type. It also suggests to the compiler to store the variable in a CPU register for faster access. The "register"

```
enum elfclass
  {
    ELF_CLASS_UNKNOWN = -1,
    ELF_CLASS_NONE = ELFCLASSNONE,
    ELF_CLASS_32 = ELFCLASS32,
    ELF_CLASS_64 = ELFCLASS64,
    ELF_CLASS_BOTH
  };
static enum elfclass input_elf_class =
ELF_CLASS_UNKNOWN;
static enum elfclass output_elf_class =
ELF_CLASS_BOTH;

static enum d
elf_class (int mach)
{
  switch (mach)
    {
    case EM_386:
    case EM_IAMCU:
      return ELF_CLASS_32;
    case EM_L1OM:
    case EM_K1OM:
      return ELF_CLASS_64;
    case EM_X86_64:
    case EM_NONE:
      return ELF_CLASS_BOTH;
    default:
      return ELF_CLASS_BOTH;
    }
}
```

```
080492e0 <elf_class>:
 80492e0:    pushl  %ebp
 80492e1:    movl   %esp,%ebp
 80492e3:    subl   $0x1c,%esp
 80492e6:    movl   0x8(%ebp),%eax
 80492e9:    movl   %eax,-0x8(%ebp)
 80492ec:    movl   %eax,%ecx
 80492ee:    testl  %eax,%eax
 80492f0:    movl   %ecx,-0xc(%ebp)
 80492f3:    je     804936b
<elf_class+0x8b>
 80492f9:    jmpl   80492fe
<elf_class+0x1e>
 80492fe:    movl   -0xc(%ebp),%eax
 8049301:    subl   $0x3,%eax
 8049304:    movl   %eax,-0x10(%ebp)
 8049307:    je     8049353
<elf_class+0x73>
 804930d:    jmpl   8049312
<elf_class+0x32>
 8049312:    movl   -0xc(%ebp),%eax
 8049315:    subl   $0x6,%eax
 8049318:    movl   %eax,-0x14(%ebp)
 804931b:    je     8049353
<elf_class+0x73>
 8049321:    jmpl   8049326
<elf_class+0x46>
 8049326:    movl   -0xc(%ebp),%eax
 8049329:    subl   $0x3e,%eax
 804932c:    movl   %eax,-0x18(%ebp)
 804932f:    je     804936b
```

Figure 5.7: An example showing false positives for finding local variables in clang due to a presence of an enum or a struct; (source code at left side) and (binary code at right side)

it is not guaranteed that the compiler will do so, as the compiler may choose to ignore this suggestion. Overall, this statement declares a pointer to a pointer to a character type, and it may help optimize the program's performance by suggesting the compiler keep the variable in a register.

- **False positives in local variables due to character array or strcut elements:** There are cases where we have got extra local variables due to the presence of character array or struct where every element of the character array and struct was accessed individually and the algorithm encountered them as the individual normal local

variables as there is no way to differentiate between those temporary variables and actual variable. For example, as shown in the left of Figure 5.6 is the source code, in which we have defined char b[2]. This led to extra variables in the final result. We see at the right side of Figure 5.7 is the binary code obtained from objdump file. At address *'804a72a'*, we see that *'0x1e(%esp)'* is used as a variable for the first element b[0] of the array and similarly at '804a735' address, *'0x1f(%esp)'* is used as the second element b[1]. Since these both are less than the depth of the stack, they were considered as local variable. As the final list of local variables we got, **local arguments are 5 {'0x1e(%esp)', '0x1f(%esp)', '0x4(%esp)', '0x8(%esp)', '0xc(%esp)'}** which gives us false positives in GCC O2.

- **False positives due to unoptimized switch cases in Clang compiler:** Sometimes, due to a presence of struct or enum gives false positives in number local variables in the case of Clang compiler. As an example, in Figure 5.7 on the left-hand side we see the source code having *'enum'* class, i.e., enumerated and used for a function called *'elfclass'*. On the right-hand side of Figure 5.7 is the binary code where we can see at the addresses like '80492fe', '804932c ', etc., we got values like '-0xc(%ebp)', '-0x18(%ebp)' which usually serves as local variables and hence giving us the false positives. Although in this function, no actual variable is used as a local variable, we have got 7 local variables. These are some special cases which need to be worked upon.

Despite these limitations, static analysis can still be a useful tool for software development. It can identify possible problems and assist developers in finding bugs early in the development cycle by offering automated code analysis. However, it is crucial to be aware of the limits of static analysis and to incorporate it within a larger strategy for software development that also uses methods like human code reviews and testing. Furthermore, it is crucial to choose a static analysis tool that is appropriate for the programming language and development environment being used, to make sure the tool is configured correctly, and to be aware of the tool's limits. Developers can effectively and efficiently use static analysis to enhance code quality and lower the likelihood of bugs and vulnerabilities in their software by following these steps.

Chapter 6 : Conclusions and Future Works

As cyber-attacks continue to evolve and become more sophisticated, there is a need for static analysis tools for mitigating security risks and improving the integrity of software systems. For a better constructive approach, static analysis in conjunction with dynamic analysis, code reviews and reverse engineering paves the way. Static analysis tools can uncover potential security flaws before the code is executed or distributed, which can considerably lower the chance of a successful cyber-attack. Early detection of security flaws allows developers to correct them before they become more complicated and expensive to fix. The motivation behind reverse engineering and static analysis is to improve the quality, reliability, and security of software systems, by gaining insight into their behavior, identifying potential issues and vulnerabilities, and developing new tools, approaches, ideas, applications, etc. that work with the software. Hence, cyber-security has become an integral part of our lives and the major aspiration behind this thesis is to make reverse engineering simple, reliable and fast. Thus, in this thesis, we presented an approach to detect function parameters, local variables and parameters used as a pointer in a function in GCC and CLANG for both O0 and O2 on Intel 32-bit and 64-bit ISA architectures. This approach is an algorithm which is additional functionality to the University of Idaho's JIMA tool. Creating a new functionality for a static tool can be challenging and generally involves many aspects like designing, integrating, testing and the scope of future improvements.

6.1 Summary

Research communities in programming languages and compilers have generally adopted the maxim of increasing programmer productivity and software reliability. Research in reverse engineering is not something we encounter commonly and even if one did, it is highly limited to one architecture or optimization level only. To fill in the gap, adapting a generalized model is the main focus of this thesis. Thus, we created an algorithm for analyzing the stripped binaries, written in Python3.x. The algorithm is not complete and has limitations but there is a scope for improvement and further analysis which has been discussed in detail in section 6.3. Hence, this chapter concludes the method adopted and findings of the thesis and highlights

future work. By using measures of precision, recall, and F1, we demonstrate the accuracy and high success rate achieved by heuristics.

We began by presenting the advantages, challenges, recent developments and the motivation behind this thesis in Chapter 1. Then we discussed the fundamental concepts of binary analysis and assembly language where we presented reverse engineering, its tools and applications. Also, we discussed Intel's architecture for both 32-bit & 64-bit. Further, we talked about stack memory management during a subroutine call made to a function and the related work done in the same direction. This led to the introduction of the JIMA toolkit, where we discussed the JIL file and how it is used as input to our algorithm in Chapter 2. Afterwards, we introduced the algorithm itself and the approach behind it along with defining some new terms. We discussed the reaching definition analysis, control flow, worklist and iterative algorithm in detail. We also discussed the pseudo-code and every step of the algorithm in detail with some data flow diagrams. Then, with the help of an example we understand the output of the algorithm in terms of parameters, local variables, and pointers as parameters in Chapter 3. In chapter 4, we presented the experimental results for various bulk test cases along with the precision, accuracy etc. Finally, in Chapter 5, we outline the challenges and limitations encountered while developing the algorithm.

6.2 Research Contributions

Finding parameters, parameters used as pointers and especially, local variables in a function in stripped binaries that are without existing any source code or high-level information is very difficult. We started by analyzing the JIL file from JIMA tool to fetch the parameters by finding the sources which are getting used without any assignment. Then, we started looking for local variables which are of the form EBP- n form. But, in our analysis we find that this works only for O0 optimization level. In general, for O2 in GCC, it is the stack depth which defines the parameters and local variables. If for any source or destination, the value is greater than the stack depth then it is considered to be a parameter, otherwise, it is a local variable. However, since the O2 is highly optimized and directly deals with registers mostly, it uses extra local variables and hence, we get many false positives. This leads the drop in the F1 score of local

variables. Further, we introduce the concept of worklist, reaching definition analysis and iterative algorithm for the distributive framework which is function for case and finally developing the algorithm. This has been studied in detail in chapter 3 with the help of pseudo-code and dataflow diagrams.

We then tested the algorithm on Clang compiler too. We found that for O0 level for parameters, it performed similar to O0 GCC. However, for local variables the results dropped drastically since in clang, every element of a *struct* is considered as local variables which gave us many false positives. Once, analyzing and with the help of heuristics that some of the variables are coming because the parameters are first stored to registers and then to local variables to use later. We fixed it by ignoring those variables, however, in some cases we still got variables with the base registers '%eax', '%edx', '%ecx', etc. This is hard to avoid in generalized algorithm since the same results are obsereved for the actual local variables in 64-bit. Thus, it resulted in less accurate results for local variables. This has a direct impact on the parameters used as pointers. Due to the parameters being stored in registers sometimes and then in local variables, it gave some false results which indirectly referencing the parameters. To cater for it, we need to ignore those registers which then again used in the function later. Since we are ignoring those, in some special cases, these gave false negatives too. Hence, there is a huge difference in parameters used as pointers in both GCC and Clang.

As algorithm is designed to statically recognize parameters, local variable and pointers which means it does not need to execute the program to recognize these types and it considers all the possible data flow paths and has a full coverage of the code. The functions in the library packages like binutils, coreutils and findutils are used as datasets compiled on both 32-bit and 64-bit for O0 and O2 optimization levels for GCC and Clang. After evaluation, we discovered the algorithm worked best for 64-bit GCC O0 level where we got 100% F1 score and out of all, it also did best in finding the parameter as pointer where the F1 score is 95.7%. As we move to O2 levels, the algorithm did not perform well due to many limitations already discussed in section 5.2. We also observe that 32-bit presents some challenges like having a 64-bit divided into two 32-bit values, thus leading to many false positives which explains the difference of accuracy between 32-bit and 64-bit. Although the Clang compiler presented other

issues due to its compiling technique, the algorithm detected parameters correctly almost 100% of the time even for the O2 level. It performed worst for detecting correct local variables.

6.3 Future Work

As discussed previously, the limitation of the algorithm states some future enhancement and gives the scope to extend this research. As research is an ongoing process and doing more investigation may yield additional significant findings. Additionally, it prepares for automated inspection and maintenance. The following are some future works:

- Firstly, working towards all the limitations stated in section 5.2 and doing rigorous testing on those cases. Since these functions are a part of standard libraries compiled on both GCC and Clang compilers, these are some test cases should give true positives or true negatives for the algorithm so developed.

- Automating the testing results to compare it with ground truth which is already in the development phase.

- The analysis presented in this thesis is a intra-function which means it only works within the range of a function. It lacks the inter-analysis of a function which leads to the full coverage of the code and gives better insights and results. The results from this type of analysis will make more sense as we get more accurate CFGs and DFGs.

- Knowing about the parameters, local variables and parameters used as pointers is a step towards reverse engineering. We also require the remaining functionalities, such as obtaining type inference, identifying signed or unsigned numbers, arrays, etc., to ensure complete coverage of the code. Overall, this means to know everything about the function signature.

- The goal is to reverse engineer and find the vulnerabilities in stripped binaries. To do so, we need to generate the structure like the source code. Thus, finding a *'struct'*, and fetching all the variables and values used is another dimension of this research.

- Currently, the algorithm works for GCC and CLANG. To make the algorithm a more generalized model, there is a need to add different ISA architectures such as ICC compiler, ARM architecture, etc. Also, we need to leverage different optimization levels like O1, O3 and so on since currently it works for O0 and O2.

- Since the code presents different scenarios which can occur in a function depending on the compiler and optimization, many conditions are required to be checked. Therefore, there is a need for further optimization and refinement. The algorithm needs to be optimized and refined to improve accuracy, speed, and performance.

- Currently, this algorithm is based on a heuristic approach. With the hybrid strategy of applied machine learning and heuristics, it is possible to get a faster and better output. This could help in increased accuracy or investigate parallel processing to speed up the processing of big codebases.

As a result, there are many possible applications for this technique in the field of reverse engineering, where it can improve software analysis and comprehension through the development of new features.

References

[1] Juan Caballero and Zhiqiang Lin, 2015. Type Inference on Executables. *ACM Comput. Surv. V, N, Article A, January 2015*. DOI: http://dx.doi.org/10.1145/0000000.0000000.

[2] G. Balakrishnan, T. Reps, D. Melski and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. *ACM Trans. Program. Lang. Syst., vol. 32, no. 6, pp. 1-84, 2010.*

[3] Bin Zeng. Static Analysis on Binary Code. *A Tech. rep., 2012 (cit. on pp. 17, 22, 23), 2012.*

[4] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software–Practice And Experience, 30:775–802, 2000*.

[5] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. *In Conf. on Comp. and Commun. Sec. (*CCS), pages 235–244, November 2002.

[6] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. *In proceedings of Network and Dist. Syst. Security, Vol. 20. No. 0., February 2000.*

[7] X. Chen, H. Tan and K. Liu. Binary Code Analysis. *A journal, Computer, vol. 46, no. 08, pp. 60-68, 2013.* DOI: 10.1109/MC.2013.268

[8] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. In: Principles of Program Analysis, pp. 35–81. Springer, Heidelberg (1999). https://link.springer.com/book/10.1007/978-3-662-03811-6%20

[9] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. *In 22nd Annual Computer Security Applications Conference (ACSAC'06). 269–278, 2006.*

[10] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The Convergence of Source Code and Binary Vulnerability Discovery -- A Case Study. *In ACM on ASIA CCS '22, pages 602-615.* DOI: https://doi.org/10.1145/3488932.3497764

[11] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled reverse engineering of types in binary programs. *2011.*

[12] M. Noonan, A. Loginov, and D. Cok. Polymorphic type inference for machine code. *In Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI, pp. 27-41, 2016.*

[13] A. Gussoni, A. Di Federico, P. Fezzardi, and G. Agosta. A Comb for Decompiled C Code. *In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (pp. 637-651) CCS, 2020.*

[14] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. *In IEEE Symposium on Security and Privacy (SP) (pp. 158-177), 2016.*

[15] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. *In NDSS, 2015.*

[16] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Loginov. Evolving exact decompilation. *In BAR, 2018.*

[17] Alan Grosskurth and Michael W. Godfrey. A case study in architectural analysis: The evolution of the modern web browser. *Software Maintenence and Evolution: Research and Practic. EMSE, 2007.*

[18] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. *In Proceedings of the 12th ACM conference on Computer and Communications Security, CCS '05, pages 340-353, 2005.*

[19] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. A theory of secure control flow. *In ICFEM, pp. 111-124, 2005.*

[20] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *In Proceedings of the fourteenth ACM symposium on Operating systems principles, SOSP'93, pp. 203-216, 1993.*

[21] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. *Technical Report UCB/EECS-2009-133, EECS Department, University of California, Berkeley, Oct 2009.*

[22] Michael Howard. Some bad news and some good news. *October 2002*. http://msdn.microsoft.com/enus/library/ms972826.

[23] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *In Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11, pp. 283-294, 2011.*

[24] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. *In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL), pp. 238-252, 1977.*

[25] IDAPro disassembler. http://www.datarescue.com/idabase/.

[26] A. Srivastava and A. Eustace. ATOM - A system for building customized program analysis tools. *In Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (PLDI), (pp. 196-205), 1994.*

[27] J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. *In Conf. on Prog. Lang. Design and Implementation (PLDI), pp. 291–300, 1995.*

[28] Phoenix. http://research.microsoft.com/phoenix/.

[29] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. *Tech. Rep. MSR-TR-2001-50, April 2001.*

[30] Chris Wysopal. Veracode: Binary Static Analysis. *A tech. rep., Introduction to Computer Security, March 7, 2012.*

[31] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. *In Compiler Construction: 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, ,(pp. 5-23), pp. 5–23, March 29-April 2, 2004.*

[32] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-Aware Malware Detection. *In Proc. of the 2005 IEEE Symp. on Security and Privacy, pp. 32–46, 2005.*

[33] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. *In Proc. of the 2005 IEEE Symp. on Security and Privacy, pp. 226–241, 2005.*

[34] C. Kruegel, W. Robertson, and G. Vigna. Detecting Kernel-Level Rootkits Through Binary Analysis. *In Proc. of the Annual Computer Security Applications Conf. (ACSAC), pp. 91–100, Tucson, AZ, December, 2004.*

[35] F. Nielson, H. R. Nielson, and C. Hankin. Principles of Program Analysis. *Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.*

[36] S. Debray, R. Muth, and M. Weippert. Alias Analysis of Executable Code. *In 25th Proc. ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 98), ACM, 1998, pp. 12-24.*

[37] T. Wang et al. IntScope: Automatically Detecting Integer Overflow Vulnerability in x86 Binary Using Symbolic Execution. *In Proc. Network Distributed System Security Symp. (NDSS 09), Internet Society, 2009.* www.isoc.org/isoc/conferences/ndss/09/pdf/17.pdf.

[38] Á. Kiss et al. Interprocedural Static Slicing of Binary Executables. *In Proc. 3rd IEEE Int'l Workshop Source Code Analysis and Manipulation (SCAM 03), IEEE CS, 2003, pp. 118-127.*

[39] W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. *International Journal of Parallel Programming, 28, 2000.*

[40] B. Guo, M.J. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D.I. August. Practical and accurate low-level pointer analysis. *In 3nd International Symposium on Code Generation and Optimization, pp. 291-302, 2005.*

[41] J.R. Larus and E. Schnarr. Eel: Machine-independent executable editing. *In Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (PLDI), pp. 291-30, June 1995.*

[42] B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of executable code revisited. *In IEEE Ninth Working Conference on Reverse Engineering, pp. 45-54, October 2002.*

[43] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. *In 13th USENIX Security Symposium, Vol. 13, pp. 18-18, August 2004.*

[44] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. *In Annual Computer Security Applications Conference, pp. 421-430, 2007.*

[45] Intel Corporation. Intel 64 and ia-32 architectures software developer's manual, *2009.*

[46] C. Cifuentes. Reverse compilation techniques. *PhD thesis, Queensland University of Technology, p. 56, July 1994.*

[47] J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng., pp. 184-189, 79, 2001.*

[48] C. Cifuentes and A. Fraboulet. Interprocedural data flow recovery of high-level language code from assembly. *Technical Report 421, Univ. Queensland, 1997.*

[49] Z. Xu, B. Miller, and T. Reps. Typestate checking of machine code. *In European Symp. on Programming (ESOP), volume 2028 of Lec. Notes in Comp. Sci., pp. 335–351. Springer-Verlag, 2001.*

[50] N. Suzuki and K. Ishihata. Implementation of an array bound checker. I*n Proc. Principles of Programming Languages (POPL), pages 132–143, 1977.*

[51] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *In Supercomputing, pp. 4–13. IEEE/ACM, 1991.*

[52] D. Brumley and J. Newsome. Alias analysis for assembly. *Technical Report CMU-CS-06-180, Carnegie Mellon University, School of Computer Science, December 2006.*

[53] B.-Y. E. Chang, M. Harren, and G. C. Necula. Analysis of low-level code using cooperating decompilers. *In Proc. Static Analysis Symposium (SAS), pp. 318–335, 2006.*

[54] R. Rugina and M.C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *Trans. on Prog. Lang. and Syst. (TOPLAS), pp. 182-195 2005.*

[55] J. Bergeron, M. Debbabi, J. Desharnais, M.M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng., pp. 184-189, 79, 2001.*

[56] A. Min´e. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. *In Languages, Compilers, and Tools for Embedded Systems, 2006.*

[57] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. *In Proc. Principles of Programming Languages (POPL), pp. 310–323, 2005.*

[58] Cristina Cifuentes and Mike Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. *In Proceedings of the 7th International Workshop on Program Comprehension (IWPC '99). IEEE Computer Society, 192.* https: //doi.org/10.5555/520033.858247

[59] Jim Alves-Foss and Jia Song. Function boundary detection in stripped binaries. *In Proceedings of the 35th Annual Computer Security Applications Conference (pp. 84-96), December 2019.*