ABSTRACT

Title of dissertation:	DEEP ANALYSIS OF BINARY CODE TO RECOVER PROGRAM STRUCTURE
	Khaled ElWazeer, Doctor of Philosophy, 2014
Dissertation directed by:	Professor Rajeev Barua Department of Electrical and Computer Engineering

Reverse engineering binary executable code is gaining more interest in the research community. Agencies as diverse as anti-virus companies, security consultants, code forensics consultants, law-enforcement agencies and national security agencies routinely try to understand binary code. Engineers also often need to debug, optimize or instrument binary code during the software development process.

In this dissertation, we present novel techniques to extend the capabilities of existing binary analysis and rewriting tools to be more scalable, handling a larger set of stripped binaries with better and more understandable outputs as well as ensuring correct recovered intermediate representation (IR) from binaries such that any modified or rewritten binaries compiled from this representation work correctly.

In the first part of the dissertation, we present techniques to recover accurate function boundaries from stripped executables. Our techniques as opposed to current techniques ensure complete live executable code coverage, high quality recovered code, and functional behavior for most application binaries. We use static and dynamic based techniques to remove as much spurious code as possible in a safe manner that does not hurt

code coverage or IR correctness. Next, we present static techniques to recover correct prototypes for the recovered functions. The recovered prototypes include the complete set of all arguments and returns. Our techniques ensure correct behavior of rewritten binaries for both internal and external functions.

Finally, we present scalable and precise techniques to recover local variables for every function obtained as well as global and heap variables. Different techniques are represented for floating point stack allocated variables and memory allocated variables. Data type recovery techniques are presented to declare meaningful data types for the detected variables. Our data type recovery techniques can recover integer, pointer, structural and recursive data types. We discuss the correctness of the recovered representation.

The evaluation of all the methods proposed is conducted on SecondWrite, a binary rewriting framework developed by our research group. An important metric in the evaluation is to be able to recompile the IR with the recovered information and run it producing the same answer that is produced when running the original executable. Another metric is the analysis time. Some other metrics are proposed to measure the quality of the IR with respect to the IR with source code information available.

DEEP ANALYSIS OF BINARY CODE TO RECOVER PROGRAM STRUCTURE

by

Khaled ElWazeer

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2014

Acknowledgments

First and foremost, all praise and thanks are due to ALLAH almighty. He is the one who blessed me with the ability to achieve this success. During hard times in the course of my PhD, I always return to him and ask him in my prayers and he always answers my prayers and supplications.

I'd like to thank my advisor Professor Rajeev Barua for all his enormous help and support along the PhD years. His continuous advice has always been shedding the light in my way.

I also thank my dissertation committee: Prof. Manoj Franklin, Prof. Shuvra Bhattacharyya, Prof. Donald Yeung and Prof. Jeff Foster for accepting to serve on my committee and for their valuable feedback.

I'd like to thank my research group members including Kapil, Aparna, Matt, and Jim. Without their hard work, help and support, this dissertation would not have been a reality.

I'd like to thank my father Dr. Mohamed ElWazeer and my mother Dr. Mona Ibrahim for all what they did for me. Being a homeschooled kid, my parents were my only teachers in almost all pre-college studies. They were very patient with me and stood behind me until I got to this point in my life. No words can describe my gratitude to them. I'd like also to thank my brother Ammar, my sister Salma and her husband Ahmed for their continuous support and prayers.

Special thanks to my wife Soha. From my first day in US, we were together. We faced all the challenges together. Without her being beside me and providing all kinds of care and support, this dissertation would not have been possible. She always takes away from her time and duties to make sure I have the suitable environment to excel in my work. I will never forget all what she did for me.

Special thanks to all my friends at the ‘Tauba’ community. The weekly gathering we have been holding since I came to US affected my life greatly. Special thanks to Dr. Hisham Abdullah, Dr. Tamer ElSayed and his wife for their immediate support during the first year of my PhD. Special thanks to my dear friend Mohamed Kashef for all what he did for me during my PhD years.

Table of Contents

List of Figures

Chapter 1: Introduction

There has been tremendous amount of ongoing work on program analysis and understanding on the source code level. Many tools as well as research efforts have taken place to analyze source code programs for variety of reasons. Many advanced tools exist for source level bug detection, vulnerabilities detection, model checking, verification, memory analysis, debugging and code optimization.

In practice, often times users need to apply the above analyses on the executable level instead of applying them on the source code level. There are good reasons for this. Most of the applications used on a daily basis are IP protected with no access to the source code. In other cases, applications utilize third party software and components with no access to their source code. Sometimes, the software that needs to be analyzed is a legacy software system with no available source code. In all these scenarios and others, users are left only with executables to analyze.

Even if the source code of software applications is available, it is usually not a good representative of what actually happens during the binary execution. There is a well-known phenomenon called: *What You See Is Not What You Execute (WYSINWYX)* [7]. Compilers translating source code into binaries often do modifications on the source code by introducing new variables, defining a memory layout for the program and doing whole program transformations. These changes can cause some vulnerabilities not existing in source code to start appearing in binary executables. In fact, researchers started realizing that some compiler optimizations might not be safe and might cause security breaches to come up in the optimized code [72].

Sometimes, the binaries do not represent source code because of modifications that happen after the compilation process took place. For example, dynamic instrumentation might be inserted into the binary after being compiled to monitor certain program behavior. Another example is bad code injection and malware.

Even if all of the above scenarios do not happen, source code analysis might be difficult. One reason is because of having a code base written in more than one source language each with different syntax and semantics.

Reverse engineering executable code is also becoming extremely important in the cyber security domain. Recently, the rate of cyber-attacks on vulnerable application code increased significantly. In 2010, the federal government observed an average of 15,000 attacks per day [27]. Most of the attacks were utilizing vulnerabilities in application code.

Because of all the above, analyzing executable code is very essential and it is commonplace today. Agencies as diverse as anti-virus companies, security consultants, code forensics consultants, law-enforcement agencies and national security agencies routinely try to understand binary code.

Unfortunately, current tools and research for handling executables is not going at the same pace as the development of source level analysis tools. The executable analysis is usually harder because of the limited amount of information available in executables compared to source code. This also results in less precise results of the same analyses done on binary executables compared to source code.

In this dissertation, we aim at converting executables to an intermediate representation (IR) similar to the IR obtained from source code. By doing this, we directly enable the reuse of all current advanced source level tools for analyzing executables without the need to develop new custom tools for this purpose.

To make the maximum benefit of the source level analyses running on the IR recovered from executables, we identify four main properties that should exist in any tool recovering IR from executables. Our goal in this dissertation is to have all the four properties in our system. Unfortunately, current existing executable level tools converting binaries to IR cannot achieve all combined four properties. The four properties are:

1. **Functionality** The recovered IR should fully represent all aspects of the input binary. For this to happen, we define the functionality property of the IR to be the ability to recompile the IR producing a rewritten binary that resembles the input binary for all input data sets. This is a strong guarantee of IR correctness. This also makes it easier for applications like binary debugging where users can update the IR with print statements, addition, or removal of code and examining the effect of that on the binary behavior. In general, code updates are possible only if the IR is functional. Any kind of compiler passes and static/dynamic or even hybrid analyses can run on the recovered IR and the results of such passes are guaranteed to be correct if the IR is functional.

2. **Quality** The IR should be of a high quality. We will use the term high-level IR to refer to the IR quality as well during the course of this dissertation. The IR quality

here means that the IR contains the same kind of information that are available in source code. For example, the IR should have functions, function APIs, variables and data types. The more features the IR has, the higher quality the IR is. This enables better code understanding by users as well as better outputs from source code compiler passes that might be run on the recovered IR.

3. **Precision** The recovered information about the binary that is represented in the IR should be precise. The conversion to IR process should recover the same information that is presented in the original source code of the binary without missing some information or adding extra too many false positives. As an example, if a function takes only two arguments in the original binary, we should recover only two arguments. If we recover less than that, we might fall into a functionality issue since the recovered IR might not work in all cases. On the other side, if we recover six arguments, the IR will probably be working fine, but we will have a less precise IR that is harder to analyze and understand. The added false positive arguments might introduce fake side effects and data flow edges that might make any analysis running on the IR less accurate and effective. Users will have less readable recovered code from the IR.

4. **Analysis Scalability** The analyses used to recover information from the IR should be scalable. The system should support arbitrary large binaries without taking too long time to analyze those. This makes the system practical to use.

Achieving all of the above four goals is very hard if we want to handle any executable in the world. In this dissertation, we show that for most of the compiled application code that is used now, all these properties combined can be achieved. We do this by laying out specific assumptions and developing techniques that can always work for binaries satisfying these assumptions. These assumptions are usually valid in most application code as we show in the dissertation.

This dissertation discusses various techniques that are necessary for any system recovering IR from binaries. We discuss the recovery of functions, function boundaries, function APIs, variables, and data types from executables. This dissertation is not claiming to discuss all aspects of the IR recovery process. There are some aspects and challenges that this dissertation does not present solutions to and are presented in other dissertations and published work.

In the next few sections we describe briefly the challenges while recovering these specific IR aspects mentioned above and present why the state of the art techniques before ours fail to achieve all the four properties stated above. We do not present a complete literature review in this chapter. Every chapter will be followed by a detailed literature review related to the topic of that chapter.

1.1 Functions Recovery

Current binary analysis and rewriting tools cannot combine the four properties described above while recovering functions from stripped executables (those without any relocation, symbolic or debugging information). This is because of the challenging trade-offs the analyses face.

In binaries, distinguishing code from data buried inside the code section, such as literal constants, jump tables, and literal tables, is difficult. Techniques such as recursive traversal [62, 67] that track direct control flow paths from the binary entry point fail to detect all possible code sequences in a binary because of the existence of indirect control flow paths.

Other techniques like [31] [30] [55] heuristically detect only some code entry points by observing certain function prologue patterns which leads to non-guaranteed code coverage. We call such techniques *best effort techniques*.

Speculative disassembly was proposed to achieve completeness by disassembling all portions of the binary including portions that could be either code or data. The cost of that is sacrificing *accuracy* by having extra recovered spurious code. The output *functionality* is also sacrificed since disassembling all possible code may result in having conflicting code sequences. In addition, the existence of many spurious functions has two more negative consequences: (i) it is difficult to manually comprehend a spurious code; and (ii) it makes code analysis inaccurate since spurious code introduces non-existent dataflow relations and side effects.

In chapter 2, we present techniques that can recover accurate function boundaries from executables with guaranteed code coverage. We present novel techniques that identify likely spurious functions not only based on their prologue patterns 'like most of the related work', but also based on what these functions do in practice 'their semantics'. We also present dynamic based techniques to further enhance the spurious code detection process.

Our function recovery methods are based on speculative disassembly. Instead of disassembling the executable from every possible byte offset, we only disassemble from code addresses that appear as constants in any of the executable segments. This will cover all functions and all executable code in the binary provided that the binary does not use computed addresses. We keep all conflicting code around, and use function inlining to merge split functions together. We identify likely spurious functions by examining the semantics of such functions and identifying invalid behaviors like accessing out of bound memory or using conditional flags in an inconsistent way. Such identified functions are hidden from the user, but they are included in the rewritten binaries for safety since they cannot be proven not to execute at run time.

Our techniques prove to be very effective in practice. We are able to identify function boundaries with almost 100% detection accuracy. We detect almost 96% of the spurious functions in the IR. We never miss any actual function that can be executed in the binary during our recovery process. None of our identified spurious functions gets executed at run time which proves that our techniques are robust and reliable.

1.2 Function API Recovery

Recovering function APIs usually mean recovering arguments and returns as well as the function calling convention. In this dissertation we focus on recovering register allocated arguments and returns of our recovered functions. We also describe how existing techniques to recover memory arguments and returns are affected by having possible inaccurate function boundaries from our function boundaries recovery techniques.

The first challenge behind identifying accurate and yet complete register arguments information is that callee-saves registers should not be counted as being arguments or returns only because they have been saved and then restored back in a function. Statically tracking which registers are used as callee-saves is not trivial. Some techniques identify callee-saves based on dynamic analyses [10]. The problem with such techniques is that they might miss arguments if the input data set is such that these arguments are not accessed. In this dissertation, we present a static technique that can recover such callee-saves accurately without missing any of them.

To recover an accurate set of register arguments and returns, we accurately and completely identify callee-saved registers. We define a callee saved register to be a register saved to a certain memory stack offset and then restored back from the same stack offset. The save operation should dominate the restore operation and the stack memory location used for the save operation should not be read or written to inside a certain function. We use a modified version of the value set analysis technique by Balakrishnan and Reps [5] to accurately track the memory stack.

The second challenge is recovering external function APIs and ensuring their correct execution while passing all needed arguments. Apart from standard libraries which have a known prototype, external function prototypes are not known to the static analyzer and needs special handling. We present a static technique that ensures that for any external function following a known compiler calling convention, all arguments are passed correctly and the execution of the recompiled IR is correct. We use a trampoline function that gets executed at run time and passes all needed arguments. This trampoline function adjusts the memory stack such that it appears to the called external function the same way it appears in the original input binary.

Finally, having possible inaccurate function boundaries present in the IR makes current memory allocated argument recovery techniques not completely functional. Such techniques usually assume there is a return address on top of the stack of a function which is not always true if function boundaries are inaccurate. For example, if a single binary function is split into two parts in the IR (because of inaccurate boundaries), this makes the second part of this function in the IR having no return address on top of the stack. Current published memory allocated arguments recovery techniques have to account for these situations. We present the necessary changes to these techniques to account for the inaccurate function boundaries problem. The return address is always abstracted in the recovered stack array in the recovered IR such that this problem is avoided.

1.3 Variables and Data Types Recovery

Current tools recovering variables and data types from executables cannot have all the combined four properties *functionality*, *scalability*, *precision* and *quality*. Some of them have very high precision at the cost of no scalability [4]. Others are scalable but with low precision [31]. Many of the current tools cannot recover functional IR while recovering variables and data types.

Existing tools recovering variables from executables often miss the special types of variables like the ones allocated on the x86 floating point stack. Such variables are very important to recover and will render the IR incomplete if not recovered. IDA Pro [31] is the only tool known to us that can recover x86 floating point variables in some cases with some sort of a heuristic that might fail and hence the recovered code is not always functional.

In chapter 4 of this dissertation, we present a sound technique to recover all variables allocated on the floating point stack in x86 architectures. The techniques presented build a linear system of equations based on the low level operations done on the binary level to determine the floating point stack height at all program points including program points located after indirect control transfer instructions whose targets are usually not known statically.

For memory and register allocated variables, current tools recovering such variables are either imprecise [31] or recover precise information with no *scalability*. For example, DIVINE [4], the most precise variable identification tool proposed in the literature spends two hours while analyzing programs of the order of 55,000 assembly instructions.

Current work on type analysis from binaries has the preciseness problem. Many type recovery tools cannot track data flow through memory which limits their type recovery capabilities especially for multi-level pointers and recursive data structures. [51] [22] [68].

Precise type analyses that can detect multi-level pointers have a scalability problem. TIE [43] is the state of the art type recovery technique from binaries which is very precise, unfortunately it is built on top of DIVINE [4] which has a well-known scalability issue.

In chapter 5 of this dissertation, we present novel techniques that can recover variables with data types 352X faster than current techniques with almost the same precision. The techniques presented are also completely functional. The basic intuition of the techniques presented is to use a non-sound pointer analysis that is very fast while maintaining the memory layout of the original binary in the recovered IR to maintain correctness. We show that a non-sound pointer analysis gives almost the same precision as a sound one while achieving linear scalability and not sacrificing the output correctness.

1.4 SecondWrite

Figure 1.1 presents an overview of SecondWrite [63], [23], [3]; the executables analysis and rewriting framework we use to implement the techniques presented in this dissertation. SecondWrite translates the input x86 binary code to the intermediate format of the LLVM compiler [46]. The disassembler along with the binary reader translate every x86 instruction to an equivalent LLVM instruction.

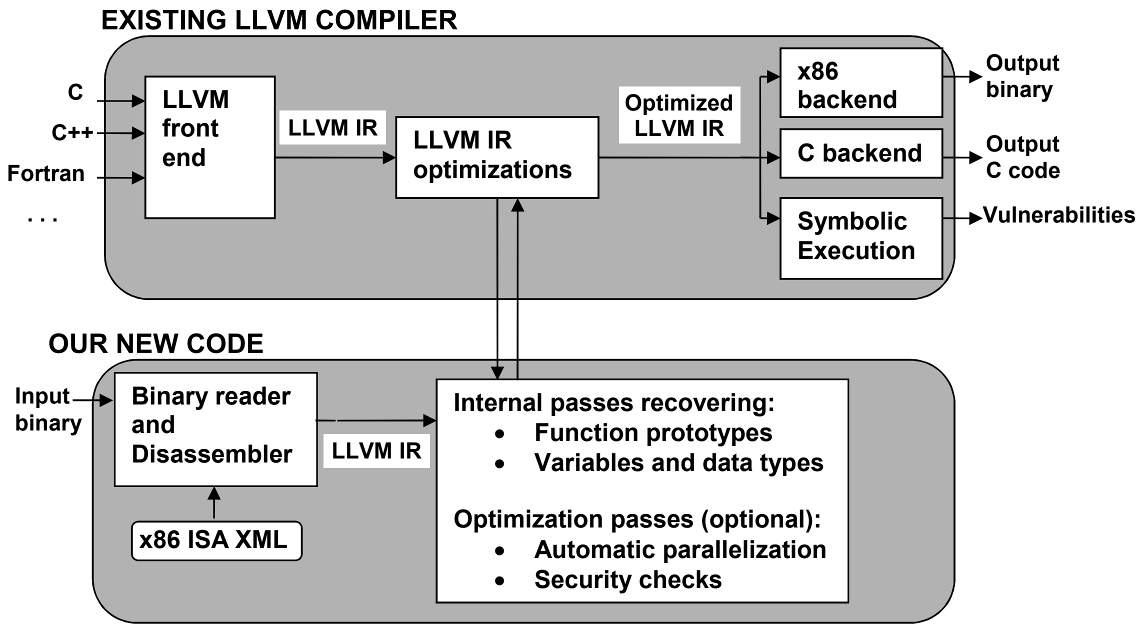


Figure 1.1: SecondWrite Flow

The disassembler implements the function boundaries recovery techniques presented in this dissertation. These techniques are essential for the whole system to recover high-level LLVM IR from stripped executables.

Once the initial LLVM IR is obtained, it is passed to SecondWrite internal passes to recover more information about the binary and enhance the quality of the IR. One of these internal passes is our function APIs recovery techniques. Another internal pass recovers variables and data types and emit them into the IR. Some other internal passes exist and not discussed in this dissertation including physical to abstract stack conversion [3], call translation, call back handlers [63] and others.

The recovered LLVM IR is then fed to the LLVM compiler core. The IR can be further optimized or analyzed. Finally, the backend of LLVM is used to generate either an output rewritten binary, or an output C code using the C backend of LLVM. The IR could be used in tools like Klee [11] to do symbolic execution analyses.

Without the techniques of this dissertation, SecondWrite cannot work correctly. If the earlier published techniques to be used (like the techniques present in IDAPro and other tools), the functionality and quality objectives of our system will be sacrificed. This is because existing tools like IDA Pro cannot guarantee complete code coverage and recover less precise information.

Our techniques can still be used by any other binary analysis and rewriting framework. They do not need to run on SecondWrite to be effective. The ideas presented in this dissertation can be applied directly while analyzing any binary (using IDA Pro tool for example). Our techniques will automatically give the functionality guarantees if applied. Hex-Rays for example can use our variables and data type recovery techniques while recovering C code from binaries.

1.5 Comparison with previous work in the SecondWrite project

The techniques presented in this thesis represent an essential component of SecondWrite [3, 23, 63]. As we discussed in the previous section, SecondWrite is a binary analysis and rewriting system that translates the input x86 stripped binary code to the high level intermediate format of the LLVM compiler [46]. The resulting IR can be used in program analyses and understanding purposes. It can also be used in many other analyses like symbolic analysis, automatic parallelization and symbolic execution. The LLVM backend can be used later to compile the resulting IR back into a rewritten binary which aids applications like binary translation and binary debugging.

SecondWrite uses the techniques described in our previous WCRE paper [63] to distinguish code from data during binary rewriting. Our previous WCRE paper presents code translation mechanisms for translating code addresses inside the rewritten binary and keeps a copy of the original binary segment in the rewritten binary to guarantee correct data accesses. The WCRE paper describes the binary characterization technique as a means to find an inclusive set of entry point code addresses inside the binary. The results presented in the WCRE paper are only related to the overheads of the translation mechanisms. There is no discussion about obtaining accurate function boundaries and eliminating spurious functions from the IR. The IR quality and correctness in the presence of functions that might be of inaccurate boundaries is not discussed in that paper.

The techniques we present in chapter 2 of this thesis completes the picture by showing that binary characterization by itself produces code with large number of spurious functions (up to 40% of the IR functions are spurious using the methods in the WCRE paper). In chapter 2, we show novel techniques that reduces the amount of IR spurious functions to less than 2%. This is essential for a high quality and readable IR. The techniques in chapter 2 has been submitted for publication in TOPLAS [25].

In this dissertation, there is no comparison with our previous EuroSys [3] techniques because they are solving a different problem in the system and because the techniques presented in this dissertation are essential for them to run correctly. The techniques presented in our EuroSys paper [3] transforms the physical stack present in the binary into an abstract stack array in the IR for every function. It also presents how symbols can be extracted from the recovered abstract stack. In chapter 3 of this dissertation, we relax one of the assumptions in our EuroSys paper [3] to enable handling spurious functions and functions with inaccurate boundaries correctly if they are ever executed in the rewritten binary. This relaxation is also part of our TOPLAS submission [25].

The remaining chapters of this dissertation build on the functions identified using the techniques presented in Chapter 2 to identify source code artifacts from binaries and represent them in the recovered IR correctly. The basic recovery of floating point stack variables with all indirect jumps are resolved, discovering all function arguments, returns, variables and data types was published in our PLDI 2013 paper [23]. We extended this work later to include discovering floating point stack variables with unresolved indirect jumps, rewriting external functions with unknown prototypes correctly, and detecting advanced data types like recursive data structures. These extensions to the work were submitted for publication to TOPLAS [24]. There is no previous work in the SecondWrite project that is concerned with solving these research problems other than our PLDI 2013 paper [23] and our submitted TOPLAS paper [24].

1.6 Organization of the Dissertation

This dissertation is composed of the following chapters:

- Chapter 2 presents methods to recover functions with accurate boundaries from executables
- Chapter 3 presents methods to recover register arguments and returns and hence function APIs
- Chapter 4 presents methods to recover the floating point stack allocated variables
- Chapter 5 presents our methods to recover memory allocated variables and data types and
- Chapter 6 concludes the thesis

Chapter 2: Recovering Functions with Accurate Boundaries

2.1 Introduction

The first step in any binary analysis and rewriting system that recovers IR from executables is to locate where the code exists and divide the code into procedures for higher quality IR.

Executables only have their entry point address visible and known in the file format. Other than that, it is up to every tool to analyze the binary and know exactly where all functions are located. This problem becomes trivial in simple executables where all control transfer instructions (CTI) like calls and jumps are direct ones. By direct we mean their target is known by examining the instruction itself. Indirect control transfers through registers or memory make the code discovery problem very challenging since an indirect call or jump can theoretically jump to any arbitrary location in the binary code.

In our work, as we mentioned in the introduction chapter, we want to achieve four properties in any IR we recover: *functionality*, *quality*, *accuracy* and *scalability*. For an IR to be functional, it has to fully represent all code in the binary which means that any disassembler we use has to have a 100% complete code coverage.

Achieving complete code coverage along with high quality IR and accurate function boundaries is very hard. The price paid while recovering all possible executable code is usually a large number of spurious functions (up to 40% according to the best technique recovering 100% live executable code by Smithson et al. [63]). We present novel techniques that reduce the number of spurious functions up to 2% which greatly enhances the readability and the quality of the recovered IR. Moreover, all our identified spurious functions are true spurious ones that never execute at run time which shows the strength of our techniques.

The main contributions achieved by the techniques presented in this chapter are as follows:

- **Accurate Function Boundaries:** We present disassembly techniques that can recover function boundaries that are as accurate as what is there in the debugging information of the input binary (debugging information is only used in testing) without sacrificing complete code coverage, thus aiding analysis precision and code readability.

- **Pruning Spurious Functions:** We present safe and sound hybrid static/dynamic techniques that can identify and delete spurious functions, aiding analysis precision and readability.

- **Marking Likely Spurious Functions:** We also present novel techniques that can identify functions that are spurious with high probability based on their semantics and move them to a separate file, thus further aiding human readability. Our results indicate that none of these functions gets executed in the rewritten binaries. In other words, for our input binaries compiled from source programs totaling over one million instructions, we never mis-identified a legitimate function as spurious

(although the opposite, which is much less harmful, did happen in a few rare cases.)

Our techniques in this chapter represent an essential component of the SecondWrite [3, 23, 63] system we use. The techniques presented in this chapter are implemented as part of the front end component of SecondWrite which reads the input binary and translates it into a working IR with accurate function boundaries.

Function information is needed in all our work presented in the following chapters. The techniques in the following chapters need function information to define function APIs and declare local variables. In addition, the high-level IR recovery techniques like the one by Anand et al. [3] need function information to detect memory arguments and be able to do interprocedural symbolic analysis.

Our techniques can still be used by any other binary analysis and rewriting framework. They do not need to run on SecondWrite to be effective. The ideas presented in this chapter can be applied directly while disassembling any binary (using IDA Pro tool for example). They will automatically give the complete code coverage guarantees if applied.

We tested our methods inside SecondWrite on the SPEC 2006 benchmark suite compiled using two different compilers (GCC and Visual Studio) on two different platforms (Linux and Windows). All rewritten binaries work correctly and give the same results as the original binaries. We show in the results section of this chapter that the recovered IR procedures using our techniques are 99% accurate compared to the original binary procedures. The 1% inaccurate procedures do not sacrifice the IR correctness. We show detailed results about SecondWrite relevant to this chapter including IR procedures accuracy and readability metrics, disassembler scalability, and the performance of some of our individual heuristics in the overall procedures recovery process.

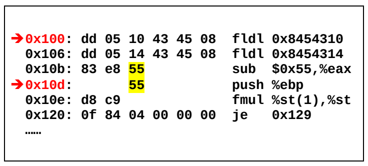
We recognize that recovering source-level function information is impossible in many cases. *We stress that this is not our target in this chapter*. We are trying to recover the same function information as what is in the debugging information of the input binary to our system; *not* in the original source code (we never use, and do not need, debugging information in our analysis). Source code functions can be significantly changed during compilation using compiler transformations like function inlining or CPS transformations in functional languages compilers. When we refer to the accuracy of the recovered functions in this chapter, we compare what we recover to function boundaries in the function symbol table of the original binary (in a control binary with debug information; the actual binary we rewrite does not have such a function table). We do not compare with the original source code.

2.2 Background and Motivating Example

The problem of recovering accurate function information from binaries and ensuring their correctness is challenging. Binaries are composed of code and data segments. Data can be embedded into the code segment and it is not known what portions of the code segment might contain this embedded data. This results in non-valid recovered code if such embedded data is disassembled or rewritten by mistake. Even among the code, since the targets of indirect calls and branches are generally not known, the structure of the code is hard to determine especially with variable length instructions (like in x86) where multiple overlapping instruction sequences are possible for the same code, depending upon which byte we disassemble from.

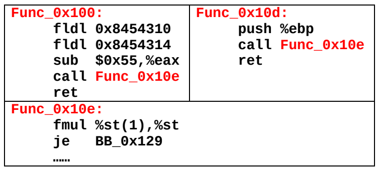
The first attempt to discover code was recursive traversal disassembly [62, 67], in which disassembly starts from the entry point of the binary (which is known according to the binary format). To recover a code portion, there has to be a direct control flow path (through direct calls and jumps) from the entry point to that portion. This leaves large portions of the binary that we cannot prove is code. As an example, let us assume the code example in Figure 2.1 is only reachable using indirect calls (through function pointers). Recursive traversal will not be able to discover any portion of this code.

Improving on that, some best-effort techniques [31, 55] add more entry points to the recursive traversal by examining known function entry prologue patterns like allocating stack, initializing the frame pointer or saving registers. The problem with such techniques is that they cannot guarantee completeness. As shown in Figure 2.1, the code example does not have any known entry point pattern and hence will not be discovered by these best-effort techniques.



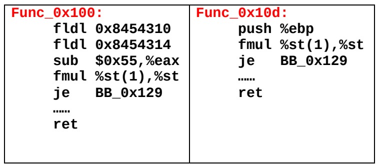
```
➔0x100: dd 05 10 43 45 08   fldl 0x8454310
 0x106: dd 05 14 43 45 08   fldl 0x8454314
 0x10b: 83 e8 55            sub  $0x55,%eax
➔0x10d:       55            push %ebp
 0x10e: d8 c9               fmul %st(1),%st
 0x120: 0f 84 04 00 00 00   je   0x129
 ……
```

Figure 2.1: Conflicting CFGs (input)



```
Func_0x100:                  Func_0x10d:
     fldl 0x8454310               push %ebp
     fldl 0x8454314               call Func_0x10e
     sub  $0x55,%eax              ret
     call Func_0x10e
     ret
Func_0x10e:
     fmul %st(1),%st
     je   BB_0x129
     ……
```

Figure 2.2: Rules application output



```
Func_0x100:                  Func_0x10d:
     fldl 0x8454310               push %ebp
     fldl 0x8454314               fmul %st(1),%st
     sub  $0x55,%eax              je   BB_0x129
     fmul %st(1),%st              ……
     je   BB_0x129                ret
     ……
     ret
```

Figure 2.3: Improvements output

Speculative disassembly [33] was proposed to tackle this problem. It assumes that anything in the code segment can potentially be code. In this case, every byte offset can be a potential entry point. This results in two main problems: 1) Code explosions due to many possible assembly listings. In the code example in Figure 2.1 we have 23 different entry points. Few of them will lead to illegal instructions, but we will still be left with a large number (up to 23) possible listings of which all but one are spurious. 2) Given variable-length x86 instructions, some of the possible disassembled listings may conflict with other listings because of unaligned and variable length instructions. As an example, The listing starting at 0x100 conflicts with the one starting at 0x10d since 0x10d starts in the middle of the instruction at 0x10b. Thus both sequences cannot be correct, but we cannot statically determine which one. Speculative techniques solve these two problems by removing conflicting code sequences to avoid code explosions. This sacrifices the complete code coverage as well as the functionality.

Recently, we proposed the binary characterization technique to greatly reduce the number of entry points (and thus spurious code) [63]. It aims to compute a super-set of all possible targets of indirect control transfers in the program. Its intuition is that under the conditions we will list later, all local control transfer instructions (CTIs) target addresses have to appear somewhere in the binary. The technique is summarized in section 2.3. In the example in Figure 2.1, this technique reduces the entry points to only the two addresses 0x100 and 0x10d.

Binary characterization partially solves the code explosion problem (around 40% of the code is still spurious, but that is better than the other speculative techniques). In this chapter, we complete the big picture for the whole system by proposing how we deal with the problem of having conflicting code sequences and obtaining accurate function boundaries out of the disassembled listings.

We have five main components to our techniques. They are presented next using an example.

Disassembly We disassemble the binary starting from each binary characterization entry point as well as from the binary entry point. During this, we keep conflicting code around. In the example in Figure 2.1 we will create two functions for both binary characterization addresses 0x100 and 0x10d as shown in Figure 2.2 and solve the conflict by creating a third function for the fall-through address from both functions (which is address 0x10e in this case). The fall through appears because both code sequences fall to the same instruction later. This step achieves complete code coverage while keeping conflicting code around. It does not solve the function boundaries and functionality problems.

Improving Function Boundaries The first step to solve the function boundaries problem is to use inlining. After our disassembly, functions exist but they are split into parts because of the spurious entry points and code conflicts. Inlining merges the splits back into one function. In this example, we inline the fall-through function into both parent functions. This guarantees that at least one of the functions will have the correct boundaries. The function at the fall-through address (0x10e) can be deleted in this case as shown in Figure 2.3. Section 2.4.2 includes guarantees on when it is safe to prune such spurious functions. This step achieves better function boundaries, but some spurious functions still exist (which are not safe to remove).

Marking Likely Spurious Functions We identify spurious functions based on their semantics. In this example, `Func_0x10d` uses the zero flag in a conditional jump without setting it. Such behavior is unlikely to happen in a valid code since such flags are most commonly used as local variables. We identify `Func_0x10d` as a potential spurious function and move it away into a separate file to enhance code readability. The function still exists since we cannot prove statically it cannot be called at runtime.

Pruning Spurious Functions Based on Dynamic Techniques We present novel techniques to safely prune spurious code based on dynamic information collected from the execution of the original binaries. For example, if we monitor that the instruction at 0x10b in figure 2.1 is executed at runtime, we can safely delete the function starting at 0x10d since its entry point instruction starts in the middle of the other executed instruction at 0x10b. This pruning is safe for all data sets, not just those used in the dynamic run(s). This is because if a portion of the code segment is found to be code for one input data set, it must be code for all input data sets. There are other ways of using dynamic information to have high quality code that we explore in this chapter.

High Level IR Functionality The last problem this chapter addresses is to guarantee correct behavior of detected functions if ever executed. In this particular example, we make sure that any caller to function 0x10d passes the value of the zero flag to the function. This is done by modifying the call translator function described in [63] to have such arguments. Detailed discussion about this can be found in section 2.7.

2.3 Binary Characterization and Code Coverage

Binary characterization's intuition was given above; here we formalize it. Binary characterization scans all the executable segments at every single byte and come up with a list of all constants that might be potential code addresses. A potential code address x satisfies the following condition: $L_i \leq x \leq U_i$, where L_i is the lower bound address of code segment i and U_i is its upper bound address.

Binary characterization is based on two assumptions which are usually valid in most application code: 1) Code addresses used in indirect control transfer instructions (CTIs) are not computed at run time. 2) No self-modifying code exists in the binary. The first assumption is valid in all compiled code which is neither position independent nor obfuscated. The second assumption is a limitation of all static executable analysis techniques. Section 2.10 discusses how these assumptions can be relaxed in our system.

The first assumption does not exclude binaries having indirect jumps with jump tables embedded in the code or the data segment. These indirect jumps will calculate the addresses of the jump table entries, not the target addresses themselves. The target code addresses are still non-computed constants present in the code or the data segment. They are loaded and then used in the jumps.

The first assumption implies that CTI local target addresses must appear somewhere in the executable segments. They can either appear as operands to instructions in the code segments, or initial values of global variables in the data segments.

When we talk about binary code coverage in this dissertation, we refer to covering instructions that can be executed in any dynamic run of the binary. We call such instructions the *live code*. There are parts of the binary that can never be executed under the assumptions above which we call *dead code*. Examples of dead code include functions that are inserted in the binary and are never reachable using any kind of control transfer instructions (either direct or indirect).

We formulate the fact that under the previous assumptions we achieve complete live code coverage in the next property. We write it as a property to easily refer to it throughout the dissertation.

Property 2.1 Let B be the binary characterization list of addresses and let x be a code address within the executable reachable indirectly. Address x must appear in B. Static recursive traversal disassembly starting from every such address $x \in B$ as well as from any externally visible function address in the binary guarantees the complete coverage of live binary code.

Proof The first part of the property falls directly from the binary characterization definition above. To prove the second part of the property, we use contradiction. Assuming that we start recursive disassembly from every address $x \in B$ as well as from the binary entry point and from externally visible functions (visible in export address tables for example) and we still do not achieve complete live code coverage. This implies that there is some part of the code that is reachable using some sort of control flow and yet is not covered by static disassembly. This code can be only reachable using direct control transfers, indirect control transfers from within the binary or from outside the binary. We prove each case individually. For code that is reachable from within the binary, if it is reachable indirectly, then it has to start with a binary characterization address which means it is covered by the static recursive traversal. If it is reachable directly, then recursive traversal will be able to follow it and hence it is also covered. For externally visible parts of the code within the binary, in order for them to be reachable by other external binaries, their entry addresses have to be externally visible to the outside code (for example in export tables, relocation tables, ... etc.). We already start static disassembly from such addresses and hence we cover such code as well.

The fact that there is some portion of the binary code that might be dead and hence not covered by our static disassembler means that we might have cases when our recovered IR after disassembly is smaller in size than the input binaries. We discuss this more in the results section and show this detailed effect on all benchmarks.

The two assumptions stated at the start of this section might seem to limit the applicability of this work especially in the security domain. This is not right because of two reasons:

1. These assumptions are only needed if we want a functional rewritten binary and 100% live code coverage. They can be easily relaxed if only binary analysis (but not rewriting) is to be performed like in many security applications, such as analyzing malware or discovering vulnerabilities in legitimate binary code. The price paid in this case is an IR which is not guaranteed to work correctly if recompiled.

2. The effect of code obfuscation does not limit the applicability of this work to the security domain. IDA Pro and Hex-Rays are well-known tools that do not present special handling to obfuscated binaries and they are still used by many security analysts to understand binaries. Our tool can be used the same way. As we show later, we have much higher precision than IDA Pro and other tools. In addition, existing de-obfuscation approaches such as [40], [42] can be used prior to our tool, to allow our tool to be used on obfuscated code.

We discuss the issue of how to relax the assumptions further in section 2.10.

2.4 Disassembly Methods

In this section, we describe our custom disassembler that overcomes the problems described in the previous section. We explain first the basic rules that create non-conflicting functions and attain complete code coverage possibly with less accurate function boundaries, and then we describe our techniques to improve the function boundaries.

2.4.1 Disassembly Rules

While disassembling the binary, a single code address `x` is reachable if it is one of the following: 1) A speculative entry point (like being a binary characterization address). 2) A target of a call. 3) A direct branch target. 4) An indirect branch target embedded in a jump table. and 5) A sequential address to a previously disassembled address. We start the disassembly process from the entry points of the binary as well as from every speculative entry point. We apply the rules in Figure 2.4 while disassembly.

For direct calls and jumps, we create a new function only in the case of call instructions, or in the case of jumps when they cross function boundaries. This latter case is needed because calls are sometimes implemented by a push of the return address followed by a jump. Indirect calls will be replaced by a call to the translator function explained in [63].

Indirect jumps are analyzed using a modified version of the well-known jump table recovery heuristics by Cifuentes and Emmerik [14]. We modified the heuristics to ensure that they can run across function boundaries. For simplicity, all recovered addresses from these heuristics will be initially considered as function entry points and annotated for later inlining. Even if the heuristics fail, all such addresses would have been recognized using binary characterization as function entry points. We use the heuristics as hints for better function boundaries accuracy because jump table targets are usually not function entry

Instruction	Rules
	If x is disassembled as part of function foo
call x //Direct call	Split foo at x
x is a constant address	Insert a call to foo_split (starting at x)
	Else Insert a call to a function starting at x
call *x //Indirect call	
x is a register or a	Replace with a call to the general call translator
memory location	
	If x is within the current function OR not disassembled yet
	Create a basic block at x inside current function
	Insert a branch to basic block x
jump x //Direct jump	Else Let foo = parent (x)
x is a constant address	Split foo at x
	Insert a call to the foo_split (starting at x)
	Annotate foo_split with JUMP_TARGET
	Run jump table identification heuristic
jump *x //Indirect	Annotate every target discovered with
jump	INDIRECT_JUMP_TARGET
x is a register or a	Create a new function for every discovered target address
memory location	Replace with a call to the general call translator
	Let foo be the current function, foo = parent (x)
	If y is not disassembled yet
	Disassemble y as part of foo
Other sequentials from	Else If y marks a start of a function bar
address x to address y	Insert a direct call to bar
where:	Annotate bar with SEQ_TARGET
x: instruction 1	Else if parent (y) = bar, bar != foo
y: instruction 2	Split bar at y
and y = x + sizeof	Insert a call to bar_split (starting at y)
(instruction 1)	Mark bar_split as a SEQ_TARGET
	Else //parent(y) = foo
	Create a basic block starting at y
	Insert a direct jump to basic block y

Figure 2.4: Disassembly Rules

points.

The rule for sequential addresses will create a new function if the sequential fall-through address is a function entry point, or if it crosses the boundaries of the current function. Such functions will be annotated as being sequential.

The disassembly rules in figure 2.4 do not describe when to end a function. We stop the disassembly process inside a function in three cases: 1) If another function begins. 2) If there is a one way control transfer instruction (CTI) redirecting control to a different function (like a jump instruction). 3) If a return instruction is detected. For call instructions, we keep disassembling the sequential address after the call. This is okay since even if the call instruction does not return back to the same call site, the disassembled extra code will never get executed which does not harm functionality.

After finishing this disassembly stage, a complete disassembly of the IR is obtained and according to property 2.1 it will cover 100 % of the input live binary code. There are two major problems in the IR after this stage: 1) There will be many function splits as shown in the example in figure 2.2. 2) Some functions might mistakenly appear as parts of other bigger functions if such functions are only reachable using direct jumps (which is the case for tail calls). The next section describes how to deal with these two problems.

2.4.2 Improving Function Boundaries

In order to do more improvements to the function boundaries, we perform three main tasks: 1) Use heuristics such as known function prologues and external stubs pattern matching. 2) Merge indirect branch targets into their parent functions. 3) Merge as many splits together as possible.

Even though this section will present some heuristics that give higher quality and more accurate IR from binaries, this does not sacrifice any correctness of the system. Whenever our heuristics fail, we still achieve our 100% live code coverage and IR correctness. We never delete IR code unless it is safe to do so. We never delete live IR code. Even if we have non-accurate function boundaries, our techniques presented in section 2.7 will make sure the flow of variables and arguments still happen to such false functions.

The first thing we do is to recognize known function prologue patterns. In our implementation, we rely on two main prologue patterns: 1) Initializing the frame pointer with the value of the stack pointer. 2) Indirect jumps to relocation entries which are used to form external function stubs in some binaries. Many more techniques and heuristics can be integrated in this step easily such as existing machine learning techniques for recovering function entry points [55].

The second improvement we do is related to merging as many splits as possible. For this, we define an *actual* function as any recovered function having at least one of the following four properties: 1) It is called directly. 2)It is non-speculative (reachable from the executable entry point). 3) It has a well-known function prologue. 4) Its entry is a binary characterization address not annotated as being sequential or a jump target. For every non *actual* function, we search all the callers up the call graph until we find a set of *actual* function parents. We then inline the non *actual* function into all its parents. To avoid code size explosions, we set a limit on the number of inlines per IR function; we call this limit the inlining threshold.

Algorithm 1 shows how an *actual* function can be identified as well as parents for

Algorithm 1 isActualFunction algorithm - detecting actual functions and parents of non-functions

```
    Input: $x$ : an address of a suspect function $func\_x$ in the IR
    Input: $analyzed$ : a set of already analyzed caller addresses (for handling recursion)
    Output: $isActualFunction$ : false if $x$ is not considered a function entry point
    Output: $parents$ : a set of all possible parent functions to the basic block starting at $x$
 1: if $x \in analyzed$ then
 2:     return with $isActualFunciton$ = false
 3: end if
 4: Set $isEntry$ IF ($x$ is the entry point function)
 5: Set $isDirectlyCalled$ IF ($x$ is reachable using at least one direct call)
 6: Set $NonSpeculative$ IF ($x$ exists in the function symbol table of the binary)
 7: Set $IsBinCharAddr$ IF ($x$ is a binary characterization address)
 8: Set $IsJumpAddr$ IF (function at $x$ is marked as JUMP_TARGET)
 9: Set $IsSeqAddr$ IF (function at $x$ is marked as SEQ_TARGET)
10: Set $IsKnownPrologue$ IF (function has a known prologue pattern)
11: if ($IsKnownPrologue$ OR $isEntry$ OR $isDirectlyCalled$ OR $NonSpeculative$
    OR ($IsBinCharAddr$ AND NOT($IsJumpTable$) AND NOT($IsSeqAddr$) AND
    NOT($IsJumpAddr$)) ) then
12:     return with $isActualFunciton$ = true
13: end if
```

```
14: insert $x$ into *analyzed* list
15: Let $C$ be the set of all functions calling *func_x* in the IR
16: for all $parent\_n \in C$ do
17:     (callerIsFunction, callerParents) = isActualFunction( AddressOf(parent_n), analyzed)
18:     insert callerParents into parents
19:     if callerIsFunction then
20:         insert AddressOf(parent_n) into parents
21:     end if
22: end for
23: return with isActualFunciton = false
```

non-actual functions. Lines 4-13 check for *actual* function attributes described above. If the function is non-actual, lines 16-22 traverse the call graph up (from callees to callers) and gets all *actual* parent functions.

There are many reasons why algorithm 1 can return more than one parent function for a certain input address. One reason is CFG conflicts. As we have already seen in the code example in figure 2.2, the function starting at 0x10e will not be considered an actual function and it will have two parents. Ideally, a single address should be part of only one function, but because of CFG conflicts and splits, we might end up having many parent functions.

It is important to notice that algorithm 1 is a heuristic which means it is not guaranteed to give accurate information. The results section quantifies how often this algorithm is accurate. The algorithm gives accurate information in most cases since suspect functions that are only reachable using direct jumps or as sequential addresses to other instructions are usually non-functions. The reasons false positives might happen where non-functions are considered functions by mistake are:

- If the jump table heuristic fails to identify some indirect branch addresses. These addresses will be considered function entry points.
- If a binary characterization address appears in the middle of a dead function which is not reachable in any run of the code. This kind of address will be considered a function.

In some other rare cases, algorithm 1 might give false negatives where it tells that an actual function in the binary is not a function and is part of another bigger function. This only happens because of tail calls which use direct jump instructions. If some function is only reachable using tail calls, and is not having any known function prologue, then algorithm 1 will mistakenly consider it as part of all parents (up to the constant threshold). The results show that this happens in less than 1% of the time.

After merging functions with their parents, some IR functions can be safely deleted. The following property states the conditions under which this can happen and proves that the IR will still be complete after such a code cleanup.

Property 2.2 Let B be the binary characterization address list. For every IR function `foo` starting at address x in the original binary where x is not externally visible in the original binary, if `foo` has been inlined with all parents, and $x \notin B$, then `foo` can be safely removed from the IR.

Proof Function `foo` is either reachable using a direct call/jump, or through an indirect call/jump. For direct calls and jumps, `foo` has already been inlined into all the parents, which means there is no more direct calls/jumps to `foo`. For indirect calls and jumps, suppose address x is reachable using indirect calls and jumps, then according to property 2.1, it has to appear in the binary characterization list B which is not the case. For calls from outside the binary, x has to be externally visible in the original binary which is not the case.

To illustrate the previous rule, figure 2.3 shows the code example in figure 2.1 after merging and deleting the function at 0x10e.

2.5 Marking Likely Spurious Code

In this section, we discuss semantic based techniques to identify likely spurious functions. The recovered functions from the previous techniques should do something meaningful. If they are doing something illogical, then probably they are not actual functions. We partition the recovered set of functions in the executable from the previous techniques into two sets according to their *execution probability*. 1) The set of functions that are unlikely to be executed at run time. We call them spurious functions. 2) All the other functions which are likely to be executed. Identified spurious functions are inserted in a separate file which the user can ignore reading. Our techniques will aim to minimize the percentage of spurious functions present in the main IR recovery file.

It is important to mention that the techniques described in this section are heuristics aiming to optimize the code for readability. We do not remove the identified spurious functions from the IR and they can still be executed (less likely). We discuss how to maintain correct execution of such functions in section 2.7. As we show in the results section, these heuristics prove to be very effective in reducing the spurious functions in the main IR recovery file down to 2% of all IR functions present in that file. None of the identified likely spurious functions using these heuristics executes in all of the tests we have done.

Even if our heuristics fail in identifying spurious code, or identifies valid code as being spurious, this does not impact the correctness of the recovered code by any means. We still keep this code around, and make sure the data flows correctly to this code if ever executed. The heuristics are used for better IR quality for manual analysis and code readability and do not impact the correctness and the code coverage by any means. The user may decide to delete the spurious file which automatically enhances the precision of any automated analyses running on the IR especially if the IR spurious functions have side effects (which is usually the case). In all our expirements, we never monitored any of the identified spurious functions got executed at run time which gives users of our tool more confidence to delete the spurious file.

2.5.1 Inlined Functions

In the last section, we mention that we inline all *non-actual* functions into their parents (if parents exist) up to a certain inlining threshold. Some of these *non-actual* functions get removed using property (2.2) above.

For the non-actual functions that did not get removed using property (2.2), if they got inlined to all the parents without hitting the inlining threshold, we mark such functions as being spurious. The reason behind this is that since they are already inlined to all parents, it is unlikely they will be executed as stand alone functions. In most cases, such inlined functions will get called (or branched to) from their parents. The only case they might get called as stand alone functions is when such inlined functions are actual functions in the input binary that are only called indirectly. This is very unlikely as we show later in our expirements.

Such non-removed non-actual inlined functions are usually split parts of bigger functions because of the existence of binary characterization address in the middle of their actual parent functions.

2.5.2 Identifying Actual Parent Function

In most binaries, a certain code region is usually part of only one function. We use this intuition to mark more spurious functions as follows.

Many of our inlined functions mentioned above have more than one parent function because of code conflicts. Algorithm 1 returns a set of parent functions for a specific inlined function. Only one of them is the correct parent and all the others are spurious.

We examine all parent functions and check how many properties for actual functions they have. Usually we find that only one function has higher properties than the others (for example, in most of our experiments, one parent only would have a known prologue pattern). If we have only one parent function having a higher number of function properties, we mark it as non-spurious and mark all the others as being spurious.

Properties for actual functions are checked in algorithm 1 (lines 4-13). We have different weights for each property. For example, the known prologue pattern is a stronger indicator than being a directly called function for example (since directly called functions might be called from spurious ones).

2.5.3 Memory Analysis

This readability optimization examines the operations done on the global memory as well as the memory stack. It moves a function to the spurious file if any of the following is true: 1) A memory access to a constant address not within the executable segments and not to a memory-mapped I/O location is detected in the function. 2) Very large number of memory arguments are passed to the function. This is tunable by the user. 3) A function accesses stack variables that are never detected to be allocated. 4) The function is accessing the memory stack using the frame pointer without initializing it first. (The frame pointer is usually a callee saved register). 5) The code is accessing the return address. Given that we currently do not rewrite position independent code (PIC), it is very unlikely that a function will access its return address. Handling PIC code is a future work as elaborated in section 2.10.

2.5.4 ISA Analysis

This readability optimization detects the binary instruction sequences that are less likely to be executed and move their parent functions into the spurious file. One optimization we do is that we move functions having instructions that access I/O ports –like x86 instructions `in` and `out` to the spurious files. We also move code that is doing software interrupts into the spurious file. These kinds of instructions and behaviors are not common in application code and more common in kernel code and drivers. User has a choice to turn this optimization off when rewriting kernel code and device drivers.

Another readability optimization moves functions that use synchronization primitives from the main file into the spurious file only if the binary is single threaded. To detect that, we assume that the binary has to do some library call(s) to create or manipulate threads. To determine that, we examine the dynamic relocation table of the binary looking for any known multithreaded library such that pthreads, OpenMP and MPI.

2.5.5 Empty Functions Detection

We let the LLVM optimizer run on the recovered functions and then detect if any of the recovered functions becomes empty without any code. This is usually an indication that the function is really not doing anything useful and have no side effects. Still as per property (2.2) such functions are not safe to remove, so we keep them but in the spurious file.

2.5.6 Conditional Handling

This readability optimization relies on the common practice in binaries where conditional flags are usually set and used in one single function. We are not aware of any calling convention that sets a conditional flag in a function, and then uses it in another function. In some rare cases, some compiler intrinsics emulating some floating point behaviors return conditional flags. If this ever happens, the techniques presented later in section 2.7 will guarantee the correctness of the rewritten binary.

Every function is analyzed with respect to all definitions and uses of conditional flags. A function can be moved to the spurious file if it has a use of a conditional flag without any definition. As an example, figure 2.3 shows the functions recovered from the code in figure 2.1. In function `Func_0x10d`, the zero flag is used without being defined. We can move this function to the set of likely spurious functions since it is highly probable it is not an actual function.

2.6 Static Function Identification Based on Dynamic Information

In this section, we discuss novel dynamically assisted static techniques that can be used to assist the previously discussed static techniques in pruning out more spurious code in a safe manner. During the dynamic run of the binary, we only measure the characteristics of the binary, but do not modify it. The collected dynamic information is then provided as feedback to a subsequent static analysis, thereby allowing the static analysis to improve. Even though the dynamic information is collected on particular data sets, the subsequent static methods we present are correct for all data sets, not just the ones seen. The techniques presented here are optional and are not necessary for the whole system to work. They prove to be effective in practice in reducing the amount of spurious code, reducing the total IR size for the spurious code, and making the static disassembly run faster.

This section is divided into the following parts: first we present the exact problem we are solving including a big picture of the dynamic component of our system with its inputs and outputs, then we proceed to discuss the technical details of how we implement this static analyses that use the collected dynamic information. Then we discuss the correctness of the dynamically assisted analyses in the sense that the IR is still complete regardless of the fact that dynamic information used might not be complete. Finally, we discuss how dynamic information from different runs can be unified in our framework.

2.6.1 Disassembler based on dynamic information - The big picture

In the previous sections, we show some techniques and heuristics that can remove some spurious functions altogether, or at least hide them from users if they are likely not to be executed. These techniques were based on static analyses only.

In this section, we extend the effectiveness of our previous techniques by letting them use some dynamic information collected from the input binary's execution. We let the input binary run for some input data set, then collect executed instruction traces and other information about locations and targets of control transfer instructions. We then use this information to safely remove spurious functions from the IR altogether.

The main challenge in the techniques presented in this section is that whatever information we collect from a certain binary's execution with a specific input data set must be valid for all possible input data sets to the binary. By other means, we have to prove that a function that we remove from the IR based on some dynamic information is safe to remove for any input data set to the binary. Otherwise, the techniques will not be safe and sound.

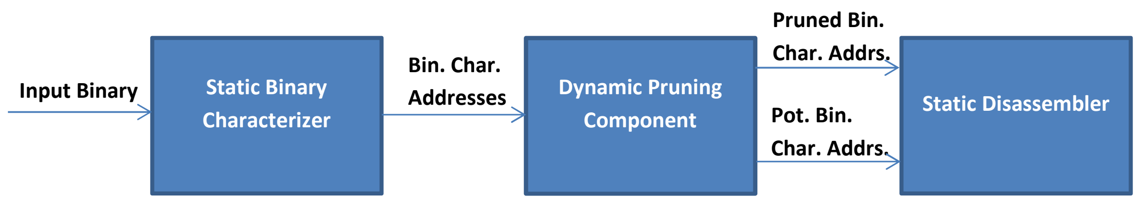


Figure 2.5: The disassembler with the dynamically assisted component inserted

As shown in figure 2.5, the dynamic component of the disassembler exists in the front end of the system and is used before the static disassembler component. First, the binary characterizer reads the binary and identifies a list of binary characterization addresses that are considered entry points to functions in our system. Instead of feeding this directly to the static disassembler (that is discussed in the previous sections), we feed that to our dynamic component which prunes out some of these addresses. The dynamic component produces a list of pruned binary characterization addresses as well as a list of potential binary characterization addresses. The static disassembler uses both to create the IR functions as we show in the next section.

The dynamic component of the system has indirect effects on the whole disassembly process. Since it reduces the amount of IR to be analyzed, the static disassembly time is reduced. The IR size is also reduced because of the same reason. We show detailed results related to these effects later in the results section.

2.6.2 Function Pruning using Dynamic Information

The dynamic information used in the techniques presented in this section are used to prune spurious functions by eliminating their starting addresses from being considered as valid entry points. We prune binary characterization addresses and hence achieve higher quality IR with less spurious code. We do not create spurious functions in the first place which is different from the previous static disassembly techniques which create functions and then prune them.

There are three main techniques we use to achieve this entry-points pruning. The first technique removes binary characterization addresses that conflict with other executed addresses. Recall that binary characterization described earlier in section 2.3 produces a superset of all possible function entry points. It does so by collecting all constant code addresses in all code and data segments inside the binary.

The second technique prunes out binary characterization addresses that are observed to be reachable only directly in the execution. The third technique relies on examining the stack memory at function entry points. All the techniques are presented next and their results are unified to produce the final binary characterization list of addresses that is supplied to the static disassembler.

2.6.2.1 Pruning conflicting addresses

In order to produce the list of pruned binary characterization addresses, we run the input binary once. We collect all instruction addresses that have been executed along with their length. We remove an address from the binary characterization list of addresses if it starts in the middle of an already executed instruction. The assumption here is that the binary does not contain overlapped instructions which are valid instructions starting in the middle of other valid instructions.

In a more formal way the following property clarifies when it is safe to remove an address x from the list of binary characterization addresses.

Property 2.3: let T be a list of pairs (i, l) where i represents the starting address of an executed instruction, l represents the corresponding instruction length. A binary characterization address x is an invalid address and can be pruned out if the following is true:

$$\exists(i, l) \in T, x = i + n, n \in \mathbb{N}, 0 < n < l \tag{2.1}$$

2.6.2.2 Pruning directly reachable addresses

The binary characterization list of addresses should contain the code addresses that are only reachable indirectly. If a certain code address is only reachable using direct control transfers, then it can be safely removed from the binary characterization list of addresses since recursive traversal can follow the direct control transfer to that function.

If a function is reachable using both direct and indirect control transfers, it might not always be safe to remove such function from the binary characterization list of addresses. To see why, consider the code example shown in figure 2.6. In this code example, function `foo` is reachable indirectly from `main`, and they reachable directly from `foo` itself (as a recursive call). If the address of `foo` is removed from the binary characterization list of addresses, the recursive traversal will not reach out to `foo` starting from `main` and hence it is not safe to remove such address.

For this pruning, we collect all indirectly reachable addresses from the execution

```
main () {
     …
     call *eax   //indirect call that targets foo at runtime
     …
}
foo () {
     …
     call  foo   //Direct call to foo
     …
}
```

Figure 2.6: Code example illustrating pruning directly reachable addresses

trace. These addresses include targets of indirect calls and indirect jumps. All such addresses are populated in a set of addresses called IT (indirect targets).

To prune a binary characterization address, we notice if it is only reachable using direct control transfers. If the address is contained in IT, then it cannot be pruned. Otherwise, it can be pruned if executed. We formalize this in the following property.

Property 2.4: Let IT be the set of addresses that are reachable indirectly during a certain binary execution. An address x can be removed from the binary characterization address list if the instruction at x was executed and $x \notin IT$.

To prove this property, we state the following property and then use it to prove property 2.4.

Property 2.5: Static recursive traversal starting from binary characterization entry points after removing all addresses that are only reachable directly will always cover all instructions that got executed while collecting the corresponding dynamic trace.

Proof: The execution trace is composed of some direct and indirect CTIs. For direct CTIs, static recursive traversal can follow them trivially. For indirect CTIs, their targets will never be pruned and hence they will still exist in the binary characterization addresses and will be followed because of that.

Proof of property 2.4: By contradiction: assume it is not safe to prune an address x that is executed during runtime and is not observed to be an indirect CTI target. This means that all code starting at instruction x will not be covered by recursive traversal in this case. This contradicts property 2.5 above since at least that execution trace will be covered using recursive traversal.

2.6.2.3 Pruning jump table target addresses

From the discussion in section 2.3, binary characterization addresses are either indirect function call targets, or indirect branch targets. Indirect call targets are always considered function entry points. Some indirect branch targets are also considered function entry points (like tail calls), but many of them are just case statement entry points represented as jump tables in the binaries.

We use the dynamic component to differentiate between indirect jump targets that are function entry points and the other non-function entry points. The intuition here is that for instruction set architectures where the return address is memory allocated (like the x86 architecture we currently support), the return address has to be stored in a specific memory location known to a function at its entry point such that the function can return back to some call site. This is only valid for non-obfuscated binaries which follow any compilation model. Obfuscated binaries in which return addresses are calculated and pushed on the stack at any arbitrary program point are not currently completely supported. We discuss later in section 2.10 how they can be supported using our framework.

We build a stack S of return addresses at the run time of the original binary by monitoring call instructions and what addresses they push on the memory stack. We also monitor return instructions and what addresses they pop from the memory stack. At every branch target, we monitor the stack pointer value before executing the first instruction at the branch target and check if it contains the last-inserted return address on the stack S (*i.e. TOP*)(S)). If the value at the stack pointer is not equal to *TOP*(S), then we insert this address into set P which represents addresses that are not function entry points according to our assumptions. This set of addresses P is the set of potential binary characterization addresses shown in figure 2.5. These are usually not function addresses but rather jump table target addresses that are reachable indirectly.

Given that the initial set of binary characterization addresses is C, we calculate the set $C' = C - P$ and start static disassembly from every address in C'. C' represents the set of binary characterization addresses that may be function entry points.

If we only disassemble from every address in C', we cannot have complete code coverage. To see why, consider that P contains all executed indirect jump table targets (representing case statements in the original source code). Such jump table targets will not have a valid return address on top of the stack since they are not function entry points. Yet they are removed from the binary characterization address list. If such addresses are only reachable indirectly, they will never get disassembled statically in this case. We usually identify such addresses using jump table heuristics when possible.

To solve the above problem, we finish all static disassembly starting from the addresses in C'. After all is done, we start the static disassembly again from any address x that is a binary characterization address and was monitored as being executed with no valid return address on top of the stack at its entry only if x has never been disassembled. This means that we start static disassembly again from any entry point address x satisfying the following three conditions:

$$x \in C \tag{2.2a}$$

$$x \in P \tag{2.2b}$$

$$x \quad \text{was never disassembled statically in the first round of static disassembly} \tag{2.2c}$$

For every such address x satisfying the previous conditions, it can be determined which function this address x belongs to by monitoring the function from which the jump originated. Address x can be then disassembled and inserted into that function in the IR.

We clarify here that the jump table targets pruning technique presented in this section only works for instruction set architectures having a memory allocated return address. The x86 architecture is one example of such an architecture where a function is called using the `call` instruction which pushes the return address on top of the stack. The return is done through the `ret` instruction which pops the return address from the memory stack. Other instruction set architectures like MIPS and ARM have a register-allocated return address that is accessible through regular instructions like branch and link and regular moves. The advanced pruning technique presented here is not currently supported for such architectures and should be turned off.

2.6.3 The dynamically assisted analyses code coverage guarantees

In this section, we discuss why the techniques presented earlier are sound and safe even though it might be coming from a limited input data set.

The first technique represented by property 2.3 excludes addresses that are in the middle of some actual valid instruction that got executed. It is evident that if the binary does not contain overlapping instructions, instructions starting in the middle of valid executed instructions cannot be valid.

The second technique represented by property 2.4 removes a binary characterization address if it is executed but is never monitored to be a target of an indirect control transfer instruction (CTI). This means that a normal recursive traversal is enough to reach out to that function and it is not necessary to include its address in the binary characterization list of addresses.

In the third technique when we remove jump table target addresses from the binary characterization list of addresses, we already add back any jump table target address that was never disassembled during the static run after pruning. Hence, we will never miss any parts of the code and still guarantee complete code coverage in this case.

2.6.4 Unification of dynamic information

In this section we discuss how we can unify information from multiple input binary runs with different input data sets. We present unification rules and discuss why they are correct.

The three techniques presented in the previous sections can be all applied one after the other to produce a final binary characterization list of addresses that is given as an input to the static disassembly. There are no restrictions on which order they are applied or which ones can be combined together. The final pruned list of binary characterization addresses is the one shown in figure 2.5. The potential binary characterization addresses shown in the same figure represent the jump table target addresses detected using the third technique above (represented by the set P).

For the first technique represented in property 2.3, the unification rule is simply the union of all x accross all dynamic runs satisfying equation 2.1. This is because such addresses are always false and this fact is not dependent on any input set.

For the second technique represented in property 2.4, the same property can be applied to dynamic information collected from different executions if all disassembled instructions are unioned and all IT sets are unioned as well. This can be verified by following the proof of the property in the previous section while applying the unification rules presented here.

For the third technique, if we have multiple runs with multiple P sets (which represent addresses of jump table targets that are non functions), we do a simple set union and apply the same technique. This is true under the assumption that in any execution of the binary, a function entry has to have a return address on top of its stack. As we mentioned before, this assumption is valid in all instruction set architectures (ISA) having memory allocated return addresses and in non-obfuscated binaries.

2.7 High Level IR Functionality

In this section, we go over some of the aspects of high level IR recovery and binary rewriting techniques used in SecondWrite and not discussed later in this dissertation and present the necessary modifications –if required– to support rewriting spurious functions correctly as well. The high level IR aspects we review in this section include call translation and conditionals. We present these below. Effects of having inaccurate function boundaries and spurious functions on other IR recovery aspects discussed later in this dissertation like identifying correct register arguments, memory arguments, variables and data types will be presented in the next few chapters.

2.7.1 Call Translation

In our previous work as well as this work, we assume there exists a translator function that is inserted at every indirect call and branch to redirect the execution at run time to the correct IR function. The translator function is a statically inserted function with a gigantic if statement that checks for the value of the target address and calls the corresponding function. The details of the translation mechanisms is discussed in our previous WCRE work [63].

Our previously published call translation mechanism will still work correctly for the IR with spurious code only if the spurious functions are added as extra entries in the translator table. This ensures correct control flow redirection in the rewritten binaries from the indirect call and branch sites to IR functions.

2.7.2 Conditional Handling

The conditionals are presented in the IR as variables that get assigned at conditional generation instructions (like arithmetic operations for example), and then get used at conditional use instructions (like conditional branches).

Splitting functions will cause some conditional flags to be defined in one function and used in another one. To guarantee the correctness of conditional flag uses, we iterate over all the functions in the IR and check if there is a use of a flag without a dominating definition. If there exists a function `foo` that satisfies this condition, we do the following three steps: 1) At every direct call/jump to `foo` from within the IR, we pass all the used flag values as extra arguments to `foo`. 2) If `foo` can be called indirectly (its address is a binary characterization address as stated in property (2.1)), then at every indirect call and indirect branch site in the IR, we pass all the used flag values as extra arguments to the translator function and hence to `foo`. 3) `foo` as well as the translator function have to return the latest version of the modified flags back to all the call sites of `foo` and the call translator.

The above three steps will guarantee the data flow of conditional flags between definition points and use points across function boundaries in all cases.

2.8 Results

This section discusses all the experiments done to test the methods described in this chapter and confirm their validity and effectiveness compared to the related work in the field.

Work	100% Coverage	Function Boundaries Accuracy	Identify Likely Spurious Funcs.	High Level Functionality
Rosenblum et. al.	✗ Known Prologue Patterns Only	$F_{0.5}$=98.9%	No	✗
IDA Pro	✗ Known Prologue Patterns Only	$F_{0.5}$=87.6%	No	✗
DynInst	✗ Known Prologue Patterns Only	$F_{0.5}$=97.1%	No	✗
Our Work	✓	$F_{0.5}$=99.5%	Reduction in spurious functions from 34.6% to 2%	✓

Figure 2.7: Recent related work results summary

Although some existing tools aim to detect function boundaries, unlike our method, (i) they do not maintain correctness in all cases (e.g., by discarding conflicting sequences); (ii) most do not guarantee complete code coverage; and (iii) they do not mark likely spurious functions, hurting readability.

The table in Figure 2.7 summarizes all our proposed features against the other tools. The F measure is an accuracy measure. We show the exact definition of that metric later in this section. The tables shows that our method to detect function boundaries is the first in the literature to ensure that the output is functional, while maintaining an accuracy that is comparable or better than existing techniques.

Speculative techniques like the one by Harris et al. [33] remove code conflicts altogether which is not suitable for our target of obtaining a complete set of functions in the IR. This is demonstrated by the fact that 12.5% of the recovered disassembled binaries using their techniques cannot run correctly. In our case, 100% of the binaries run correctly if recompiled from the recovered IR.

Despite the fact that current tools are incomparable to us as they are solving a partial problem of what we are solving, we compare our accuracy to them to show that we did not sacrifice the function boundaries accuracy while maintaining functionality and complete coverage. We also show more results on the amount of spurious functions we are able to eliminate which is a readability metric that none of the related work on speculative disassembly shows.

We show more detailed results on the SPEC2006 benchmarks compiled using two compilers. Table 2.8 shows the complete list of binaries compiled using the GCC 4.3 compiler along with their size (in assembly instructions and lines of code (SLOC)) and the number of functions each benchmark contains. Table 2.9 shows our visual studio binaries. Some GCC binaries in the first table are not shown in the VS table because visual studio does not compile Fortran code and some C and C++ SPEC benchmarks.

The charts used here in this section are all based on optimized binaries because these are the challenging ones where function boundaries are harder to recover. Non optimized binaries usually have standard prologue patterns and almost no tail calls which makes identifying the boundaries much easier. Optimized binaries are also more common among deployed binaries.

To collect dynamic information traces to apply the techniques in section 2.6, we use the PIN tool by Intel to run the input binaries and collect all the required instruction traces and other information as per the discussion in section 2.6. We noticed from our experiments that the sensitivity of the results obtained to the input data set is very low. The results almost did not change by having different input data sets with different sizes.

Application	Lang	Inst	Funcs	SLOC	Type	Version
specrand	C	290	5	49	SPEC2006	2006
mcf	C	3,357	26	2,685	SPEC2006	2006
lbm	C	7,740	22	1,155	SPEC2006	2006
astar	C++	12,677	155	5,842	SPEC2006	2006
libquantum	C	13,800	121	4,357	SPEC2006	2006
bwaves	F	19,002	9	918	SPEC2006	2006
bzip2	C	21,408	105	8,293	SPEC2006	2006
sjeng	C	32,238	146	13,847	SPEC2006	2006
milc	C	34,183	237	9,784	SPEC2006	2006
sphinx	C	41,669	373	13,683	SPEC2006	2006
leslie3d	F	43,432	23	3,807	SPEC2006	2006
hmmer	C	85,981	541	35,992	SPEC2006	2006
namd	C++	103,365	154	3,188	SPEC2006	2006
soplex	C++	116,743	1,593	28,592	SPEC2006	2006
zeusmp	F	118,429	79	19,068	SPEC2006	2006
omnetpp	C++	148,453	2,770	20,393	SPEC2006	2006
h264ref	C	170,684	593	51,578	SPEC2006	2006
gobmk	C	196,230	2,683	157,883	SPEC2006	2006
cactusADM	C	218,896	1,395	60,452	SPEC2006	2006
povray	C++	288,957	2,098	108,339	SPEC2006	2006
perlbench	C	313,036	1,872	126,367	SPEC2006	2006
gromacs	C/F	396,450	3,872	65,182	SPEC2006	2006
calculix	C/F	506,725	1,386	105,683	SPEC2006	2006
dealII	C++	766,555	18,779	96,382	SPEC2006	2006
gcc	C	934,292	5,627	236,269	SPEC2006	2006
xalancbmk	C++	965,001	30,062	267,318	SPEC2006	2006
tonto	F	1,303,359	4,086	108,330	SPEC2006	2006

Figure 2.8: Application Table (GCC-compiled binaries)

Application	Lang	Inst	Funcs	SLOC	Type	Version
specrand	C	302	5	49	SPEC2006	2006
mcf	C	2,149	26	2,685	SPEC2006	2006
lbm	C	2,174	22	1,155	SPEC2006	2006
astar	C++	6,681	155	5,842	SPEC2006	2006
bzip2	C	10,785	105	8,293	SPEC2006	2006
sjeng	C	20,838	146	13,847	SPEC2006	2006
milc	C	26,987	237	9,784	SPEC2006	2006
sphinx	C	37,901	373	13,683	SPEC2006	2006
hmmer	C	60,737	541	35,992	SPEC2006	2006
namd	C++	72,517	154	3,188	SPEC2006	2006
omnetpp	C++	101,480	2,770	20,393	SPEC2006	2006
h264ref	C	113,550	593	51,578	SPEC2006	2006
gobmk	C	179,612	2,683	157,883	SPEC2006	2006
perlbench	C	222,994	1,872	126,367	SPEC2006	2006
gcc	C	702,755	5,627	236,269	SPEC2006	2006

Figure 2.9: Application Table (VS-compiled binaries)

Because of that, we only present dynamic information based results in this section for the combined traces from the test and the ref data sets of the SPEC 2006 benchmarks suite.

All benchmarks are rewritten successfully and the recovered high level IR (with functions, arguments and variables) is recompiled using LLVM's backend. The rewritten binaries produce the correct answers which shows the output functionality and the complete coverage we achieve. To the best of our knowledge, no static rewriter can produce a correct rewritten binary with accurate function boundaries from binaries exceeding a million instructions. We do not show numbers on the run time of the rewritten binaries since this is mostly affected by the SecondWrite framework itself (with its internal passes) and not by our techniques. The effects of SecondWrite on the rewritten binaries runtime can be found in [3]. Next, we show that we do not sacrifice the accuracy of the recovered function boundaries. We also show the amount of spurious code we were able to identify as well as the time spent during the disassembly process. We also show the effect of the individual heuristics in identifying spurious functions. We show how these results change with the dynamic information present and used.

2.8.1 Comparison with best-effort techniques

We compare against the machine learning technique by Rosenblum et. al. [55] which aims to solve a much simpler problem than what we are trying to solve. They only recover function entry points. Unlike our method, they do not recover complete boundaries with guaranteed functionality. If Rosenblum's technique is used for disassembly to identify code, it would lead to incomplete coverage.

Rosenblum et. al. [55] calculate the F-measure $F_{0.5}$ which is a well-known accuracy metric in the machine learning field. The F-measure is usually used for binary classification problems where a test is being done on a certain data set and the test has only two possible outcomes. The F-measure is the harmonic mean of the precision and the recall of the test. In general, the precision (PR) and the recall (RC) are calculated according to the following formula:

$$PR = \frac{TP}{TP + FP}$$

$$RC = \frac{TP}{TP + FN}$$

Where TP is the true positive results, FP is the false positives, and FN are the false negative results. The $F_{0.5}$ gives more relevance to the precision than to the recall. It can be defined as follows:

$$F_{0.5} = \frac{1.25 * PR * RC}{0.25 * PR + RC} \tag{2.3}$$

In our test (as well as Rosenbulum et al. test), the true positives are the functions with correct entry points. The false positives are the entry points we identified as being function entry points but they are not which represents spurious functions in the context of this dissertation. The false negatives are the entry points of functions that we missed during our analysis. We never miss any function as per our code coverage guarantees, but we might inline a function into some parent and hence miss that entry point. We calculate the inlined functions (which are real functions in the debugging information of the binary) as the false negative ones.

Rosenblum et al. [55] report an $F_{0.5}$ measure of 98.8% among all recovered entry points for stripped binaries compiled from gcc and 92.3% for visual studio. We calculated the same measure $F_{0.5}$ for our techniques for both compilers and it is 99.4% for gcc and 95.3% for visual studio. Visual Studio binaries usually have more functions with no default prologue patterns and hence their numbers are usually less.

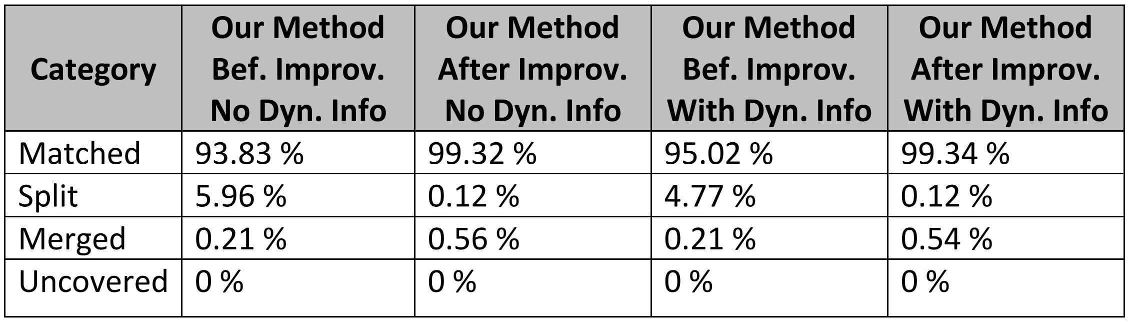
It is worth mentioning that we already perform much better than the well-known disassemblers IDA Pro [31] and DynInst. The reported $F_{0.5}$ for IDA Pro for GCC is 87.6% and for Visual Studio is 78.9%. For DynInst, the reported $F_{0.5}$ is 97.1% for GCC and 6.7% for Visual Studio. This is as reported by Rosenblum et al. [55].

If we incorporate the dynamic information while identifying functions, the numbers become slightly better. The $F_{0.5}$ for GCC becomes around 99.5% while for Visual Studio it becomes 95.5%. Dynamic information is more beneficial in reducing the size of the IR and reducing the disassembly time by eliminating many spurious entry points.

These results show that despite solving a more difficult problem, we are able to achieve higher quality of function entry points. This shows that our techniques can replace even the best machine learning techniques and get better IR with all our added features of functionality and complete code coverage.

2.8.2 Function Boundaries Accuracy

In this section, we describe the quality of the recovered function boundaries.



Category	Our Method Bef. Improv. No Dyn. Info	Our Method After Improv. No Dyn. Info	Our Method Bef. Improv. With Dyn. Info	Our Method After Improv. With Dyn. Info
Matched	93.83 %	99.32 %	95.02 %	99.34 %
Split	5.96 %	0.12 %	4.77 %	0.12 %
Merged	0.21 %	0.56 %	0.21 %	0.54 %
Uncovered	0 %	0 %	0 %	0 %

Figure 2.10: Function Boundaries Accuracy

We define three metrics for every function in the original binary (read from the debugging information) indicating its quality in the recovered code. A *matched* function is when the exact function boundaries are discovered. A *split* function is when a single function from the input binary is divided into many different recovered IR functions. A *merged* function is when the input binary function is recovered as being part of another bigger function in the IR. Theoretically, an original function has to be one of these three categories.

Figure 2.10 shows the average matched, split and merged functions on all our binaries. We initially detect 93.83% of the functions with exact boundaries and this improves to 99.32% after doing our proposed improvements presented in section 2.4.2.

The dynamic techniques presented in section 2.6 have a very small effect on the results presented in this section. There are two reasons of this. First, the results are already very good from our static methods (more than 99% of the functions are already matched). Second, the effect of the dynamic information is usually a reduction in the number of the entry points (binary characterization addresses). All of such addresses that are pruned are spurious and represent no functions in the input binary. This has no effect on real functions in the input binary. The effect of dynamic information is presented more on the IR size and the amount of time spent in disassembly as we show next.

We do not show detailed per benchmark result for matched, merged and split functions since in most of the binaries we get 100% matched functions comparing to the functions in the symbol table of the original binary. Some larger binaries will have a tiny percent of merged functions (usually less than 1%). Examples of such binaries are gcc, xalancbmk and gromacs. The common trend in such binaries is that they have larger functions in the input binary, so the binary characterization detects more spurious addresses in the same function which requires more splits to happen in the first disassembly stage, and more merges to happen in later stages.

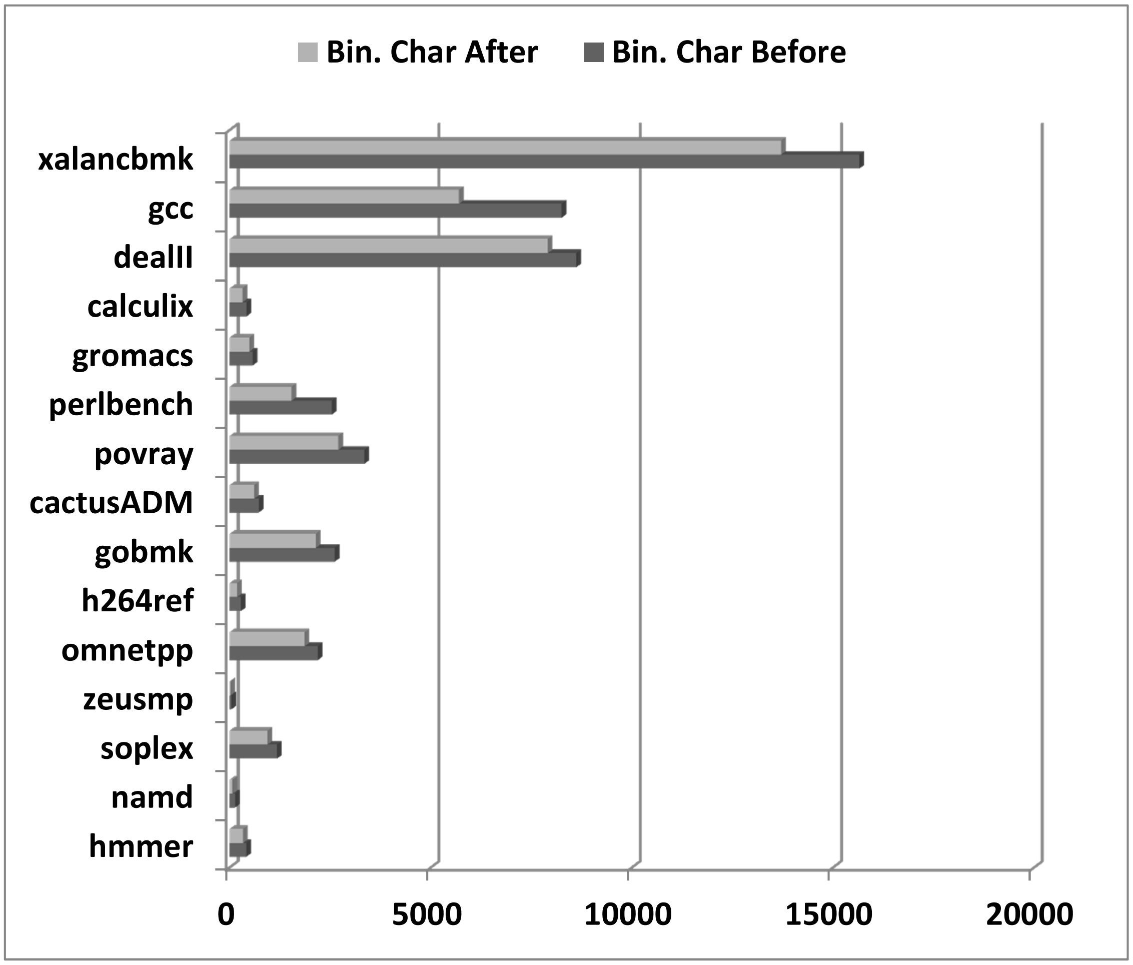


Figure 2.11: Reduction in number of binary characterization addresses

2.8.3 Dynamic Based Reduction in Binary Characterization Addresses

In this section, we show the quantified effect of having dynamic information present while doing the static disassembly on the number of entry points for static disassembly. We show how the binary characterization list of addresses (which constitute the above entry points) change by having the dynamic information.

The average reduction in the number of binary characterization addresses is 31.4% in GCC binaries and 22% on Visual Studio binaries. This reduction is calculated after using all the discussed dynamic techniques in section 2.6.

Figure 2.11 shows the details of the number of binary characterization addresses before and after the pruning for some large GCC benchmarks. We do not show results for smaller benchmarks because they will not appear on the graph as their number of binary characterization addresses is negligible compared to the larger benchmarks.

The percentage of reduction does not change much by changing the input data set for the SPEC benchmarks from the test input to the ref input data set. The results are shown for the combined data sets.

Some benchmarks like dealII have a small reduction percentage (8.2%) compared to other larger reduction percentages (gcc has 30.1% reduction). This is usually because the dynamic runs used to calculate the dynamic traces needed for the experiment did not cover large parts of the binary characterization entry points in dealII and similar binaries. This is a feature of the input data set used to conduct this experiment.

2.8.4 Spurious Functions

Binary characterization as described in section 2.3 can lead to redundant function entry points. This is the price paid to guarantee complete code coverage and functionality. Here we present detailed statistics regarding spurious functions.

Figure 2.12 shows the percentage of the spurious functions present in the main IR file after every stage of our techniques for GCC binaries. We show this for both cases when we use the dynamic information and when we do not use it. Figure 2.13 shows the same percentages for Visual Studio binaries.

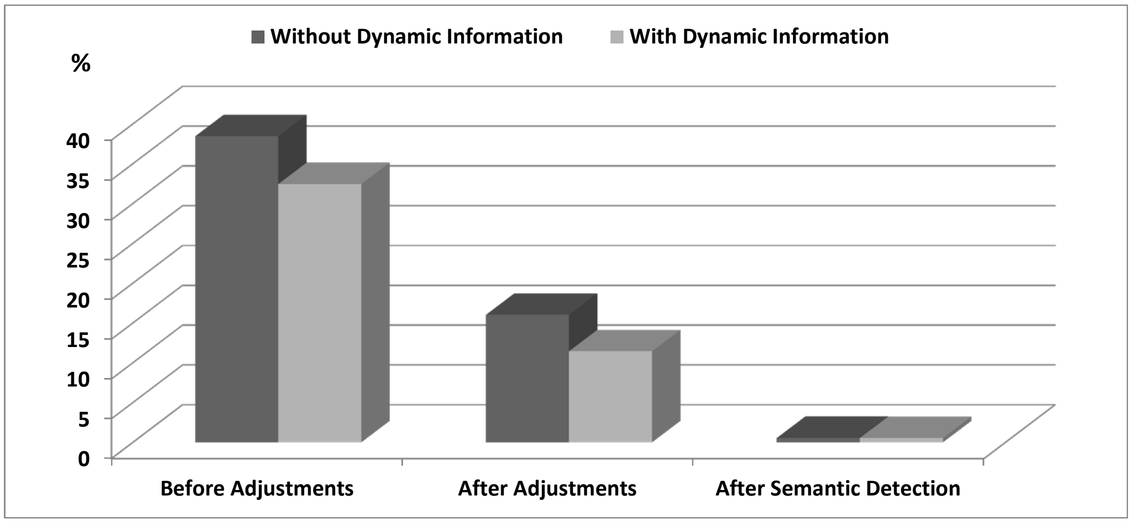


Figure 2.12: Percent of spurious functions - GCC binaries

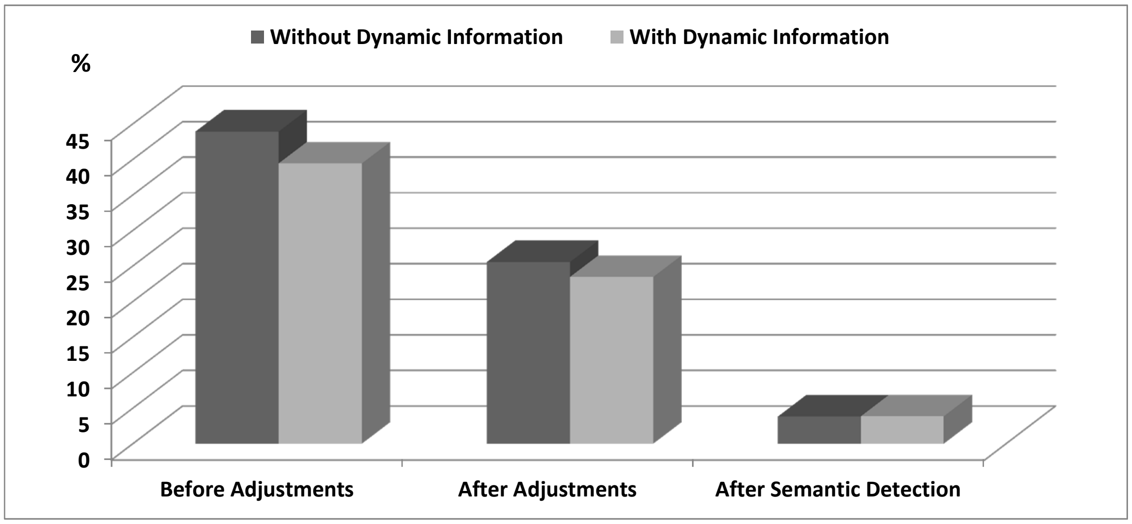


Figure 2.13: Percent of spurious functions - VS binaries

For GCC binaries, after the basic disassembly algorithm described in section 2.4.1, an average of 38.5% of all IR functions are spurious. This percentage is reduced to 32.4% if we use dynamic information. During the function boundaries improvements phase, some of these spurious functions can be safely removed using our property 2.2. This brings down the spurious IR functions to an average of 16% of all IR functions (11.4% if we use the dynamic information). Finally, applying the heuristics described in section 2.5 prunes away most of these remaining spurious functions from the main file resulting in around 0.55% spurious functions (also the same 0.55% if we use dynamic information). The final spurious functions percentage after applying the heuristics did not change when we use the dynamic information since the total number of functions is smaller and the remaining spurious functions in the IR at this stage is negligible, so the overall percentage does not come down that much compared to the percentage without using the dynamic information.

The same trend happens in Visual Studio binaries as shown in figure 2.13. The only difference is that we have higher percentages of spurious code at every stage. The reason is that we noticed that Visual Studio binaries have much larger set of binary characterization addresses than GCC binaries. As an example, the gcc compiler binary compiled using GCC has 8,249 binary characterization addresses while the corresponding Visual Studio binary has 11,155 addresses.

Figure 2.14 shows the detailed per benchmark results for spurious functions detection in GCC binaries. Figure 2.15 shows the same result for Visual Studio binaries. The x-axis in these figures represent the percentage of the spurious functions that remain not detected in the main IR file. Zero percent in these graphs means no spurious functions

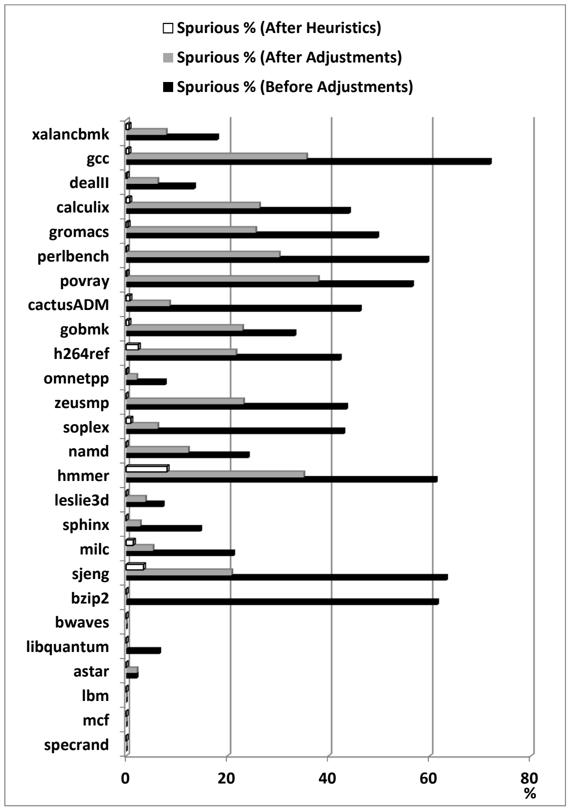


Figure 2.14: Spurious Functions - GCC

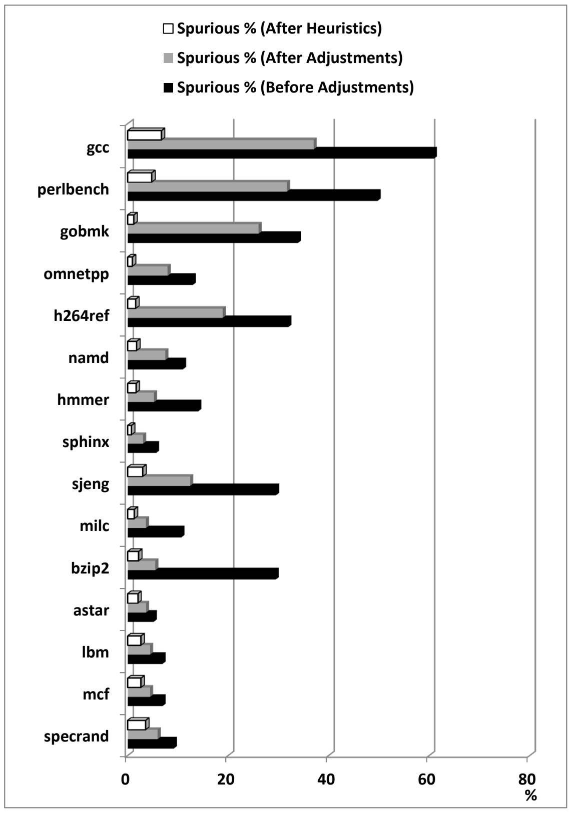


Figure 2.15: Spurious Functions - VS

remain undiscovered in the main IR file. We show only the results without using the dynamic information. Using the dynamic information gives the same trend. As expected, small binaries usually have smaller spurious code that remains after all our adjustments and heuristics. This is because the number of binary characterization addresses is usually small in such functions resulting in a lower number of spurious functions.

As we discussed in the section 2.5, we do not remove the spurious functions from the binary for safety reasons such that if one of our heuristics fail we still have functional rewritten binaries. We monitored the execution of the rewritten binaries and none of the spurious functions detected by our heuristics gets executed. This shows that the spurious functions detection does not have any false positives. False negatives do happen; these are when we leave a function as a non-spurious function but it is actually spurious. False negatives happen with a rate of 0.55% as we discuss earlier.

There are other methods that can prune out spurious code from obfuscated binaries. They are trying to solve a different problem which is de-obfuscation. If such methods are applied to our code with our assumptions, they would delete valid code which is not acceptable. As an example, one of the heuristics used in [40] to delete spurious code from conflicting CFGs is to delete one random function such that the conflict in CFGs is resolved. That is an unsafe approach since the IR is incomplete in this case, resulting in a non-functional recovered IR.

2.8.5 IR Size Changes due to Adjustments

Our function boundaries adjustment techniques may result in a change of the IR code size. The main factors that affect the IR code size are: 1) Inlining. 2) Spurious code removal (property 2.2).

We show the effect of our adjustments to the IR compared to the original IR obtained using the basis disassembly techniques presented in section 2.4.1. The original IR obtained using the basis disassembly techniques contains many function splits as we showed before. We also show how the IR code size changes after applying our dynamic techniques presented in section 2.6.

Figure 2.16 shows the detailed results of the increase of IR code size while doing the adjustments for GCC binaries. Figure 2.17 shows the same results on Visual Studio binaries. On average, the IR code size increases by 5.5% for GCC binaries due to having spurious functions as well as due to the function boundaries improvements we do (like inlining). The increase is 7.2% for Visual Studio binaries. This shows that the growth in code size from our methods is modest and manageable. This increase requires more memory, but on the other hand, our detection of spurious functions makes the amount of code a human reverse engineer has to look at significantly smaller. This is a significant benefit in reverse engineering.

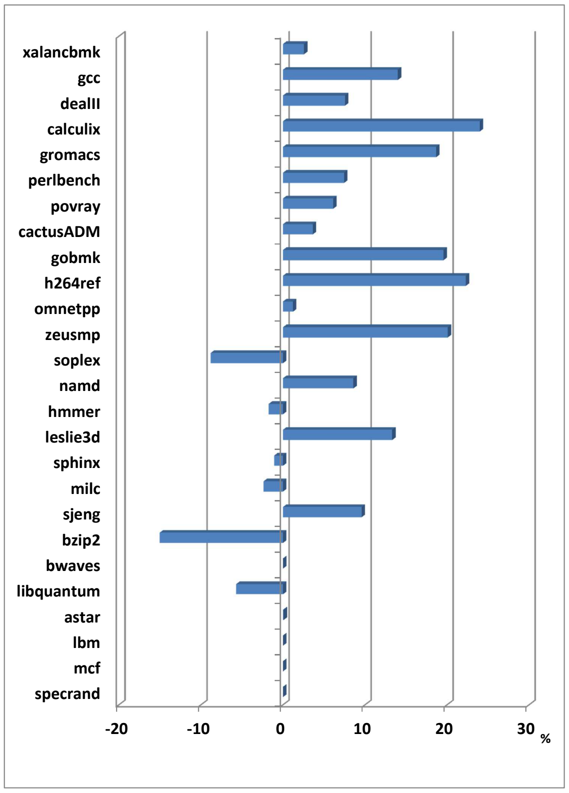


Figure 2.16: Code Size Effect (GCC)

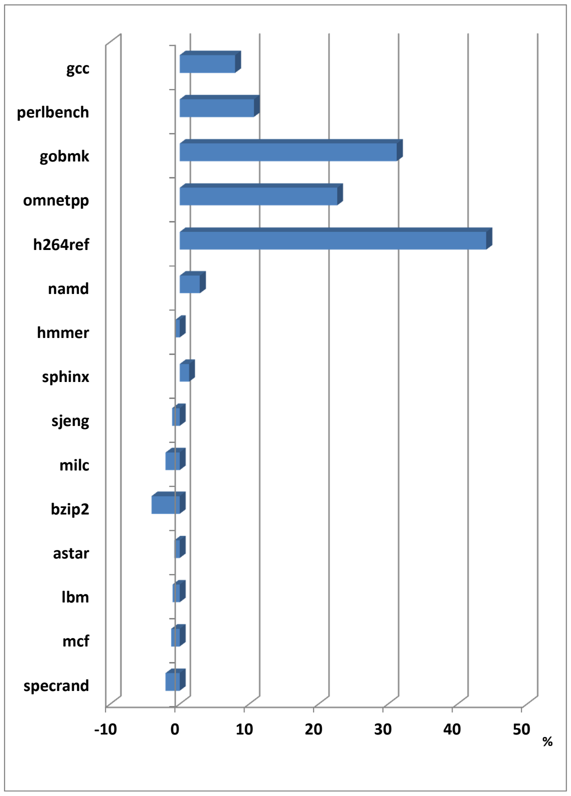


Figure 2.17: Code Size Effect (VS)

The figure shows that Fortran and some larger C binaries usually have some code increase. This happened because many of these binaries contain bigger functions which usually causes more splits to happen. Since there will also be many binary characterization spurious addresses in the same function, we will not be able to remove many of these splits as the starting address of removed functions cannot to be in the binary characterization list. This overall behavior increases the code size.

If we apply our dynamic techniques, many binary characterization addresses get removed as we showed before. This results in a decrease of the IR size. The average code size reduction is about 7.4% in GCC binaries and around 3.3% for Visual Studio binaries. Visual Studio binaries have much more binary characterization addresses than GCC binaries and hence not many addresses get pruned with the same dynamic information. This results in larger Visual Studio binary sizes.

2.8.6 Disassembly Time

Figure 2.18 shows a scatter plot between the time spent in SecondWrite for our techniques (in seconds) versus the binary size for all the binaries we have in our tests. We show two sets of points – one with dynamic information being used and the other one without dynamic information being used.

The average runtime of our disassembly techniques was 3.1 minutes with a maximum of 55 minutes on the gcc binary compiled using GCC (which is 934,292 instructions).

If we add the dynamic information to aid the static disassembly techniques, the average disassembly time reduces by 32%. The average disassembly time becomes 1.7 minutes and gcc takes around 21.7 minutes during disassembly. This is expected since the entry points needs to be disassembled is reduced when having the dynamic information.

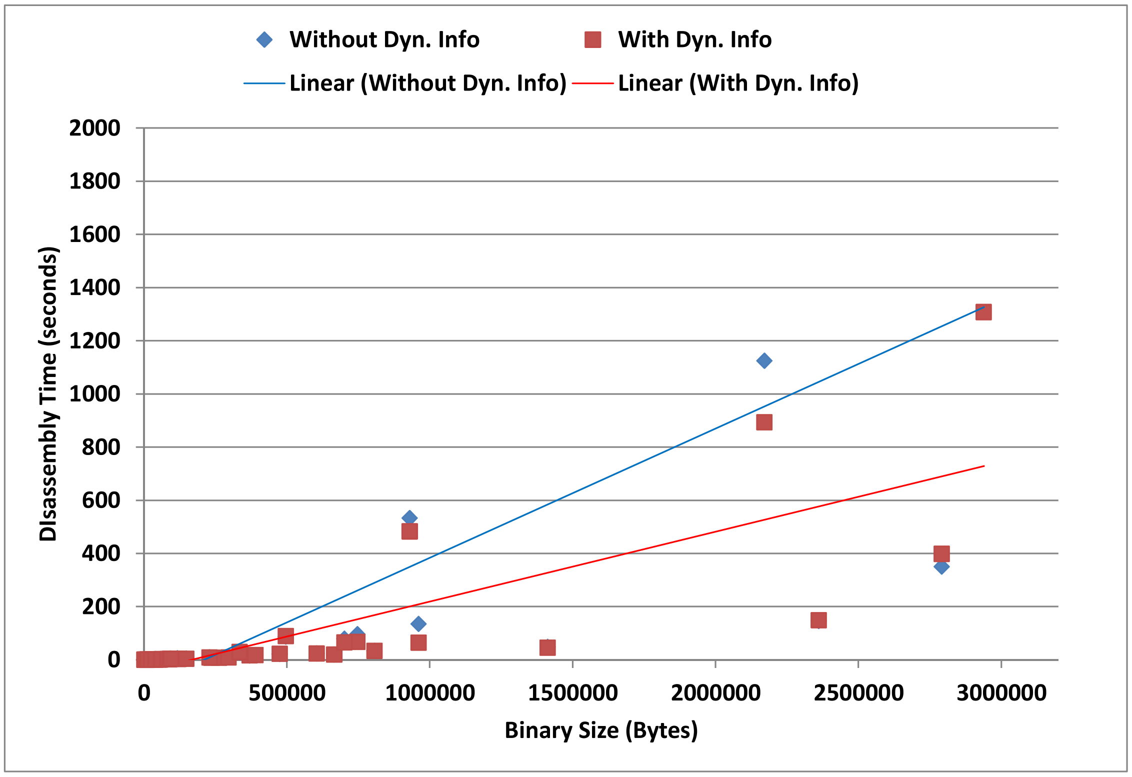


Figure 2.18: Time spent during disassembly

2.8.7 Heuristics Effect

Figure 2.19 shows the share of every heuristic among the ones discussed in section 2.5 while identifying spurious functions in GCC binaries. Figure 2.20 shows the same results for Visual Studio binaries.

The best heuristic is the one based on inlining functions to all parents. Many spurious functions have direct control transfers from other functions and they are successfully inlined to all of them and hence are less probable to execute at run-time and are considered spurious. Such spurious functions represent 74% of all spurious functions on average in GCC binaries and 47% of all spurious functions in VS binaries. The Visual Studio binaries have many more binary characterization addresses which makes this heuristic less effective.

The next effective heuristics in GCC binaries is the one based on detecting the actual parent of inlined functions. Only one parent is usually an actual parent and all other parents are marked as being spurious. 16% of the spurious functions on average are detected based on this heuristic. For Visual Studio binaries, the same heuristic is not as effective and ISA based heuristic performs better with 24% reduction in spurious functions. For VS binaries, the detection of the actual parent heuristic has an 18% share in the spurious functions reduction.

The remaining semantic based techniques contribute to the remaining 10% of the spurious functions. The most effective heuristic out of all semantic based ones is the memory analysis based heuristic. This is true in both GCC and VS binaries. Many of the detected spurious functions have unusual memory accesses and can be identified based on that.

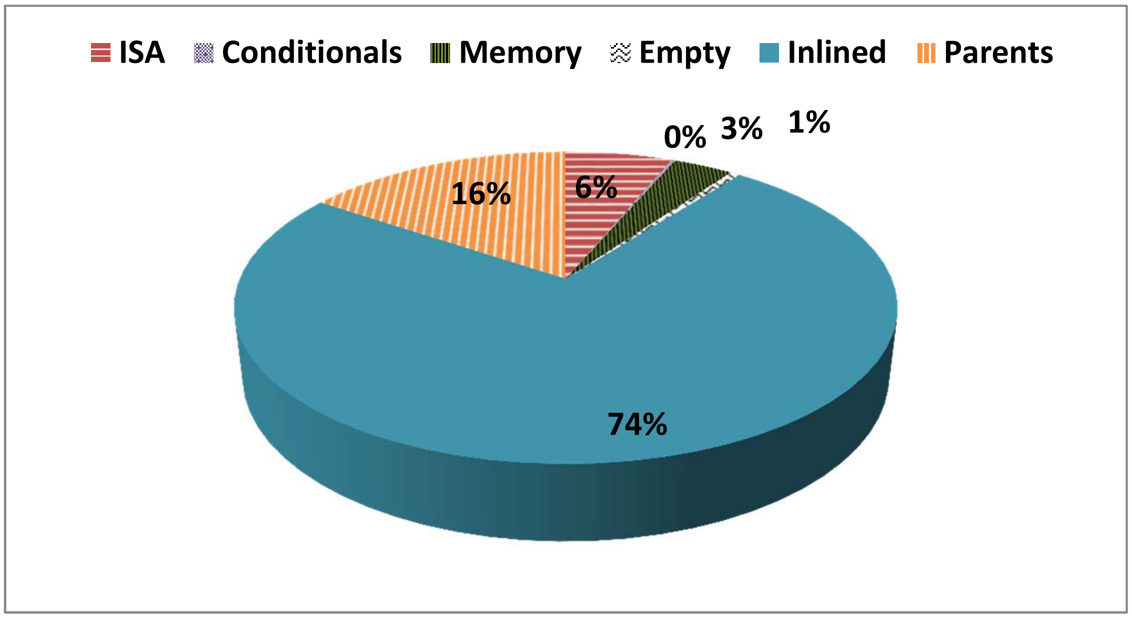


Figure 2.19: Heuristics Effect - GCC binaries

Some of the spurious functions have more than one property that qualifies them to be spurious. For example, most inlined spurious functions can be caught using the memory analysis techniques as well. In the results shown in figures 2.19 and 2.20, we do not show this effect. We run the heuristics in order and stop once one heuristic identifies a function as being spurious. We start first with the inlining heuristics and then use the semantic based ones. We could have chosen any other order. We noticed that inlining is the most effective heuristic so we run it first.

2.9 Related Work

Section 2.8 has already compared with some of the related work. In this section, we cover other related work to the disassembly process and function boundaries recovery.

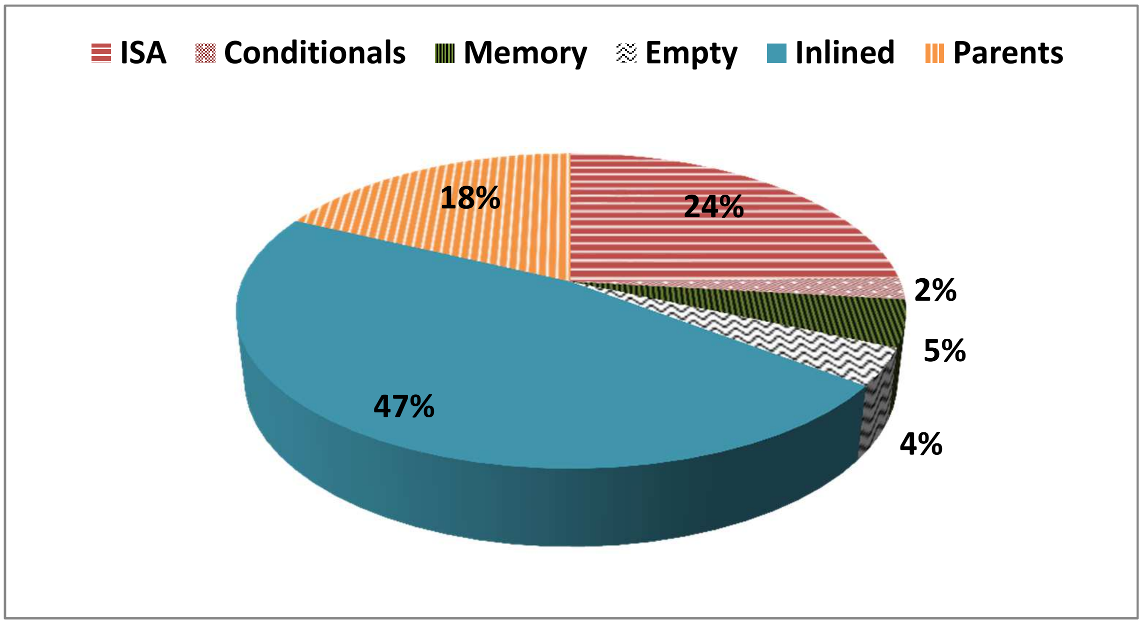


Figure 2.20: Heuristics Effect - VS binaries

De-obfuscation techniques are orthogonal to the techniques discussed in this chapter, so we will not discuss them here. We discuss some of these techniques in section 2.10.

Cifuentes et al. [14] propose some methods which are used by the UQBT tool [16] to recover indirect control transfer targets from binary code based on program slicing and pattern matching with some well-known function prologues. They do not recover any function boundaries. Their technique does not guarantee 100% code coverage (in fact they report an average of 74% code coverage). Their methods are not robust since prologue patterns depend on the particular compiler, its version number, and flags used. Other work by Theiling [67] has the same problem.

Other control flow graph (CFG) construction techniques proposed in [38] use data flow analysis to reason about indirectly reachable targets. It is used in the Jakstab [36] tool. It does not guarantee full code coverage and do not try to recover function boundaries. A later hybrid disassembly approach was developed [35] to improve the Jakstab tool code coverage. The technique is based on a formal description of a technique similar to the one used in the BIRD dynamic binary rewriter [52].

Sutter et. al. [20] and Schwarz et. al. [59] also look at resolving CFGs from binaries but they are not practical since they require relocation information.

Tallent et. al. [66] develop binary analysis techniques to aid the attribution of dynamic runtime costs to dynamic calling contexts. For that they have techniques to recover function entry and exit points in binaries, as well as recovering complete stack unwinding information. They assume that some part of the binary has to be non-stripped and debugging information exists in the binary which is not suitable for our problem. Their work is used in the HPCToolkit suite of performance monitoring of applications [2].

Shen B. et. al. implement a binary translation system called LLBT [61] which is ARM to LLVM based. One of their code discovery techniques is similar to the binary characterization technique [63]. Their disassembler is suitable for aligned instructions and cannot be used for variable length instructions like in x86 where code conflicts problem can arise.

Recently, machine learning techniques were introduced [56] to detect which compiler was used to produce a certain binary or to differentiate code from data in x86 binaries [76]. Such techniques are best-effort and do not guarantee complete code coverage.

Another machine learning technique presented by Wartell et al. [76] provides methods to differentiate code from data in x86 executables. It does not recover any function boundaries though. Their work identified the problems with IDA Pro regarding differentiating code from data and build a classifier to overcome this based on training the classifier on 10 binaries and then testing it on one binary.

Marco et. al. introduced a static system [19] built on top of Schwarz et. al. disassembler [59] to detect vulnerabilities in x86 binaries based on symbolic execution techniques. Their work assumes binaries have relocation information which is not true in stripped binaries. Another tool by Wartell et al. [74, 75] enforces security by doing binary rewriting. The rewriter relies on the IDA Pro disassembler [31] which is a best-effort disassembler that cannot guarantee complete code coverage.

Binary rewriting has been considered by a number of researchers. There are two main categories when talking about binary rewriters – dynamic binary rewriters and static binary rewriters. Dynamic binary rewriters rewrite the binary during its execution. Examples are Pin [48], BIRD [52] and DynInst [34]. None of the dynamic binary rewriters can guarantee complete code coverage. They can only cover the portion of the code that is being executed. Examples of existing static binary rewriters include ATOM [28], PLTO [60] Spike [18] and Diablo [70] none of which support stripped binaries as they require relocation information.

Some binary analysis platforms like BAP [9], [64] and CodeSurfer [30] rely on the IDA Pro [31] disassembler which cannot guarantee complete code coverage. Some other tools like Boomerang [26] rely on specifying where the entry point of the program is which makes it of a very limited capability. All such tools can benefit greatly from the techniques described in this chapter.

2.10 Limitations and Future Work

In this section, we describe some of the limitations to the work presented in this chapter and possible directions on how to tackle them. The three main limitations to this work are position independent code (PIC), obfuscated code and self-modifying code. Below we describe each one of them.

2.10.1 Position Independent Code

One main assumption this work relies on is that the binary does not have any calculated addresses. This is valid for application code we dealt with till now, but it is not valid in some shared library code.

Shared libraries are usually loaded at run time. There are two main techniques to load shared libraries: 1) Load-time relocation. 2) Position Independent Code.

Load-time relocation simply uses a relocation table in the library code. Every entry in the table is updated at load time with the correct address. The good thing about these libraries is that the relocation table cannot be removed even if the library is stripped. This allows accurate function boundaries to be identified without any issues. Usually such libraries are handled nicely in our framework. Fortunately, all Windows DLLs and many Unix ones fall in this category.

Position independent code is the other technique some Linux shared libraries use to avoid the overhead of the load-time relocation. Such code computes the addresses of functions and variables at run-time. Whenever the compiler decides to calculate a function address at a certain location in the binary, the compiler will first load the current program counter, and then adds the offset to that function from the current location. This invalidates our assumptions that addresses are not calculated at run-time.

Many instruction set architectures (ISA) will have different techniques to implement position independent code. There are two main techniques to load the program counter at run time. The first one is to use a dedicated program counter register that is accessible in regular move instructions in the ISA. This technique is usually found in RISC architectures like ARM and MIPS. In x64, there is an explicit addressing mode called the RIP-relative addressing mode which makes the program counter register visible to some instructions. The second way of loading the program counter is used in ISAs where no dedicated program counter is available to instructions. In these cases, there is usually an instructions that pushes the return address into memory for function calls. A simple technique to load the program counter is to call some address which then pops whatever on the stack and jumps back to the original call site.

The idea to support position independent code is to detect where the code is trying to access the program counter. For ISAs where the program counter is visible, this is very easy and obviuos. For other ISAs (like x86) where the program counter is not visible to instructions, instruction sequences that simulate the loading of the program counter value can be detected using pattern matching techniques. Once we detect the binary locations that are loading the program counter, constant propagation and memory analysis will lead to actual address calculations. We are currently looking at some techniques to recover this information efficiently. Fortunately, the sequences of instructions that can read the program counter are not many, and simple pattern matching techniques can be effective.

One important note is that the reader might think that since we do not support position independent code (PIC), then we do not support the binaries compiled for address space layout randomization (ASLR). This is not exactly right since ASLR will not result in a PIC binary code. For example, in Windows OS, if ASLR is turned ON for a specific binary (achieved by turning on the (`/dynamicbase` option in the linker in Visual Studio), the binary will still have a preferred load address that is hard coded in the binary image (the static binary entry point). The operating system relocates the binary image at load time to a different pseudo random base address (rebasing). No position independent code is necessary for rebasing as we discussed earlier. Windows does not use PIC code to achieve rebasing.

The only technique that might be affected by ASLR is the dynamic based techniques discussed in section 2.6. Such dynamic techniques needs to collect the executed addresses during the original binary run-time. Such addresses are then used during the static disassembly process. Since the static disassembly is based on the preferred static base address of the binary, the collected instruction addresses at run-time might be totally different from the static image addresses and hence become useless.

It turns out that ASLR is not a real problem for our dynamic techniques. Instead of collecting the executed virtual addresses at run-time, we can collect the binary file offsets. The conversion between the executed virtual addresses and the binary file offsets is trivial if we know what is the actual dynamic base address of the running binary. During run-time, knowing the actual binary base address is trivial. So, as a conclusion, ASLR is supported in our framework only if it does not result in PIC binary code which is satisfied in most of the cases.

2.10.2 Obfuscated Code

Obfuscated code is one challenge that we currently cannot completely handle. There are two main techniques to obfuscations: source code obfuscations and binary obfuscations.

Source code obfuscations are easier to implement and more wide spread. They rely on making the source code very hard to read by complicating simple operations and adding more redundancies. As long as these techniques do not introduce calculated addresses, we can handle them nicely in our framework since they will be compiled using a compiler and the executable will follow a certain compiler model.

The binary obfuscations are implemented on the low executable level, where obfuscation is inserted on the assembly level. The survey by Roundy et al. [57] summarizes all such techniques and current work in handling them. The most famous research on binary obfuscation techniques is by Linn and Debray [45]. In general, the most used obfuscation techniques are: 1) Inserting junk code in unreachable code places to trick the linear sweep disassemblers, 2) Using a return instruction to simulate a direct call. 3) Altering the return address of call instructions and inserting junk code after calls, 4) Using interrupt handling to make function calls [53].

There are many static de-obfuscation techniques that were developed recently trying to handle such problems. Most of them disassemble the binary starting from every single byte offset. Static techniques presented in [40], [71] try to resolve conflicts in the disassembled code by removing any conflicting CFG. This cannot be safe if this technique is applied to non-obfuscated binaries as well as obfuscated ones. Since they do not try to solve the problem of having complete code coverage, this is acceptable in their case.

Some other work by Lakhotia et al. [42] try to reason about function boundaries in the presence of obfuscation techniques. They rely on IDA Pro [31] for the disassembly process. IDA Pro is not accurate enough even for non-obfuscated binaries [55]. Their technique is to build an abstract stack from the physical stack and monitor the behavior of return instructions and calls.

Another work presented by Ma et al. [49] tries to extract control flow of binary code with calls simulated by returns. They use prologue epilogue patterns as well as tracking the stack pointer manipulation to detect the returns acting as function calls. They present their technique only on one binary program. Other work by Boccrado et al. [8] and Lakhotia et al. [41] achieve the same goal by precisely tracking the stack memory.

To handle the obfuscation problem in our framework, we are looking at static techniques that can insert a translator function similar to the one we described in [63] at every return instruction. If no calculated addresses exist in the binary, the translator will be able to redirect execution to the correct code. The translator structure has to be changed to accommodate the normal return instruction use as well as the obfuscated use.

The interrupt handling mechanisms can be handled by developing techniques to recover the exception tables from the binary code. We are working on techniques to recover such information.

One promising technique to be able to use our techniques as is without change is to record one execution of the obfuscated binary and build a control flow graph and a call graph at run-time and produce a de-obfuscated binary that can be analyzed by SecondWrite. During run-time, we can collect targets of branches, calls and returns and can build a deobfuscated binary this way. This is an ongoing work in our group that is under test. The only disadvantage of such techniques is that the resulting recovered IR will be only valid for this particular input data set. It will not be generalized to any other input data set. We are looking at some ways to overcome this problem. Some existing dynamic techniques of malware code extraction and reuse like the Inspector Gadget [39] and Trace Oriented Programming (TOP) [77].

2.10.3 Self Modifying Code

Like most static binary tools, we do not handle self-modifying code. Various tools [73] statically detect the presence of self-modifying code in a program. Such a tool can be integrated in our front-end to warn the user and to discontinue further operation.

The most common scenario where self-modifying code exists is for malware binaries that are packed and unpack themselves at run-time. The good thing is that unpacker has to emit the complete code that can be executed by the malware. To guarantee code coverage, the dynamic technique described at the end of the previous section can be used. It can monitor the unpacking process and tracks what instructions are emitted at run time and emits an unpacked binary. After that, our current static techniques can disassemble this image and proceed.

Chapter 3: Recovering Function APIs

3.1 Introduction

In this chapter, we present our techniques to recover function prototypes for the recovered functions from the previous chapter. The recovery process is presented such that the IR is still functional, accurate, of a high quality and the recovery process is scalable.

This chapter is composed of five parts. Section 3.2 addresses the problem of recovering a complete and precise set of register arguments and returns to internal functions whose body is inside the binary. Section 3.3 extends the discussion to include external functions which only have calls from within the binary. We show how to pass the correct arguments to such functions even if their prototypes are not known and show under what assumptions this is guaranteed to work. Section 3.4 shows the effect of having inaccurate function boundaries from the techniques discussed in chapter 2 on our previously published memory arguments recovery techniques [3] as well as to the register arguments recovery techniques. Section 3.5 shows the results of our proposed techniques. Finally, section 3.6 shows the related work in the literature.

3.2 Function Prototypes Recovery

Detecting the complete and accurate set of function arguments and returns is essential in producing a high quality code that can run correctly if recompiled. If some arguments are missing, the code will not work correctly in all cases. If more unnecessary arguments are identified, the code will run correctly, but will be less understandable by users.

We show how to accurately identify the register arguments and returns. Existing techniques show how to identify the exact set of memory arguments. SecondWrite already uses a variant of the algorithm used by Balakrishnan et al. [5] to identify memory arguments [3]. Surprisingly, we did not find any related work that correctly and accurately recognizes register arguments and returns. Not recognizing register arguments and returns is acceptable if the goal is to help human understanding of binaries (as for existing methods), but unacceptable if the goal is to generate correct rewritten code (as for our method.) Typical x86 codes have less register arguments than memory arguments, but they still have large numbers of register arguments especially for optimized executables.

A brute force algorithm for identifying register arguments and returns is to define the set of registers read without being initialized inside a procedure as arguments, and the registers modified inside a procedure and then later used at some of the call sites as returns. This technique will result in many spurious arguments since all registers which are saved and then restored back in a function (such as callee saves) will be declared as arguments and returns for this function, which is not true. Further, this algorithm might miss some arguments if not carefully implemented. For example, a procedure not accessing any register at all might be declared as taking no register arguments, which may not be true since it might be calling a function which is taking a register argument.

We propose below an algorithm which identifies accurately all register arguments and returns. Our algorithm is conservative since it will not miss any arguments. It is also accurate since it prunes out unnecessary extra arguments in many cases.

The main challenge in being accurate and yet conservative is to accurately track all registers that are saved and restored (callee-saves). Such registers are usually re-used inside functions for their local variables and temporaries. They are saved at the beginning of a function and then restored back at the end of the function. This allows the function to write and read from them without corrupting their original values. Such registers will be considered as arguments and returns to functions by mistake using the brute force technique described earlier if they are not identified.

The key challenge in tracking callee saved registers is that the stack locations used to save such registers need to be tracked to make sure they are only used for this purpose, thus allowing those registers to be pruned from the arguments or returns. The stores of the register values at the beginning of the function should dominate the loads used to restore them back. There should not be any write to those stack locations in between. If those stack locations are read in the middle of a function, the corresponding registers must be declared as arguments.

Our register arguments and returns detection technique is shown in algorithm 2. It is composed of five steps. 1) We assume all registers are arguments to every function and there are no register returns. 2) We declare all registers written to inside a function or any of its callees as potential return registers. 3) We run our algorithm for detecting saved

Algorithm 2 The algorithm to detect register arguments and returns

```
    Input: LLVM IR for a binary
    Input: $AllRegs$ : set of all available physical registers
    Output: *RegArgs* : map between functions and their register arguments set
    Output: $RegRets$ : map between functions and their return registers set
1:  for all Function F do
2:      *PotArgs*(F) = $AllRegs$
3:  end for
4:  *PotRets* = FindPotentialReturns ()
5:  ($DeadStores$,*PotRets*) = FindDeadStores (*PotArgs*, *PotRets*)
6:  *RegArgs* = PropagateArguments ($DeadStores$, *PotArgs*)
7:  $RetArgs$ = PruneReturns (*PotRets*)
```

locations by detecting the set of stores to the memory stack which are never loaded back except before the return from the function. We call those store instructions *DeadStores* since they will be eventually removed from the code. For each of the detected dead stores, we determine the corresponding saved register and remove it from the potential returns set. 4) We run our algorithm to propagate the register arguments correctly and prune unused ones. 5) We prune the unused return registers out. Next, we describe each of those steps in details. Step 1 is trivial. We proceed from step two.

The second step in our algorithm is to detect the initial set of potential return registers. Algorithm 3 shows the details of the detection method. The simple idea is that any register which is being written to inside a function is a potential return register from this function. For example, if a function `foo` is calling function `bar`, and `bar` is modifying

Algorithm 3 The algorithm implementing: FindPotentialReturns ()

```
    Input: LLVM IR for a binary
    Output: PotRets : map between functions and their potential return registers
 1: WorkList = Functions sorted in reverse call graph order
 2: while WorkList is not empty do
 3:     remove a function F from WorkList
 4:     mark function F as started
 5:     for all Instruction I in F do
 6:         if I writes to a register $r$ then
 7:             PotRets(F) = PotRets(F) $\cup \{r\}$
 8:         else
 9:             if I is a call instruction to function X then
10:                 let $callee$ = called function
11:             else if $callee$ is started then
12:                 PotRets(F) = PotRets(F) $\cup$ PotRets(X)
13:             else
14:                 add F to the end of the WorkList
15:             end if
16:         end if
17:     end for
18: end while
```

`eax`, then `foo` and `bar` will be declared as potentially returning `eax` despite the fact that there is no write to `eax` inside of `foo`. We do a post-order depth-first search traversal of the call graph (which visits child nodes before their parents) and propagate the set of potential return registers upwards in the call graph by looking for the written-to registers. Whenever we find a call to a function, we add its potential returns to the caller function potential returns. We handle recursion using a work list mechanism such that whenever we detect a call to a function which has not been analyzed yet, we add the caller function back to the work list.

After detecting the potential returns, we add them to the IR in every return statement inside every function. If more than one register is returned, we return a structure containing all combined potential return registers.

The third step in our algorithm is to detect the callee saves registers and exclude them from the list of potential returns. Since callee-saves values are saved to the memory stack, we need a memory analysis technique to track the memory stack locations where they are saved. Tracking memory in executables is not a trivial task. Our saved registers detection does not need a sophisticated memory tracking algorithm because it only needs to track stack memory. Neither heap nor global memory need to be tracked.

We modify the Value Set Analysis (VSA) algorithm proposed by Balakrishnan et al. [5] by removing global and heap memory tracking, keeping only stack memory tracking. We also remove the context sensitivity from the algorithm since it is not needed in this application. The resulting algorithm is less powerful for general memory tracking but is sufficient for this purpose.

As a quick summary of the VSA algorithm, it derives a conservative estimate of the set of addresses and integer values every memory location and register can contain at any program point. Every set of values is represented as a strided interval with a lower and upper bounds; and a stride. In our modified implementation of VSA, we only keep track of the lower and upper bounds.

In our modified version of the VSA, we assume that indirect calls will only access stack locations up to a certain offset determined by the maximum number of memory arguments to all functions in the binary. We also use the know external function prototypes to determine that maximum offset for external functions. Finally, we assume that TOP VSA values do not alias with the stack offsets used to save registers. These TOP values are usually input dependent values, global pointers or heap pointers that usually do not alias with the memory stack. The only exception is for arrays allocated on the stack with statically unbounded indexing. Those are usually not common since such arrays are usually allocated on the heap (sometimes they are allocated on the global memory).

Before we run the saved registers detection algorithm, we convert the registers inside of each function into the SSA form. This is straight forward; indeed in our implementation LLVM already does that. Our algorithm works on a temporary copy of the IR.

Algorithm 4 detects the dead stores used to save registers and prunes those saved registers from the potential return register set. Lines 6 through 12 in the algorithm collect the addresses on the stack that are used to store register values. For each of those addresses, a simple memory liveness analysis is being conducted using standard memory-to-register promotion and dead code elimination compiler passes (both these passes are already available in LLVM). Lines 13 through 16 create a dummy memory location in

Algorithm 4 The callee-saves detection algorithm (FindDeadStores)

```
    Input: A copy of the LLVM IR for a binary
    Input: $PotArgs$ : maps functions to their potential register arguments
    Input: $PotRets$ : maps functions to their potential return registers
    Output: $DeadStores$ : maps functions to the dead register stores
    Output: $PotRets$ : The input map after pruning saved registers
 1: for all $reg \in PotArgs$ do
 2:     Create a dummy register $dummy$ ; $DummyRegs(reg) = dummy$
 3: end for
 4: $ADDRS = \phi$
 5: for all Function F do
 6:     for all Instruction I in F do
 7:         if I = store $reg$, $Ptr$ AND $reg \in PotArgs$ then
 8:             if ValueSet($Ptr$) = $\{address\}$ (Singleton) then
 9:                 $ADDRS = ADDRS \cup \{(reg, address, \mathrm{I})$
10:             end if
11:         end if
12:     end for
13:     for all $(reg, address, \mathrm{I}) \in ADDRS$ do
14:         allocate a dummy pointer $DummyPtr((reg, address))$ at the beginning of F
15:         store $DummyRegs(reg)$ to $DummyPtr((reg, address))$
16:     end for
```

```
17:   for all Instruction I in F do
18:     if I is UnsafeInstruction(address) where (reg,address,X) $\in$ ADDRS then
19:       insert a volatile load from DummyPtr((reg, address))
20:     end if
21:     if I = store value, Ptr AND ValueSet(Ptr) $\supseteq$ {address} AND (reg,address,X)
    $\in$ ADDRS then
22:       insert a store value to DummyPtr((reg, address))
23:     end if
24:     if I = load Ptr AND ValueSet(Ptr) $\supseteq$ {address} AND (reg,address,X) $\in$ AD-
    DRS then
25:       insert I' = load DummyPtr((reg, address))
26:       for every use of I insert a cloned use of I'
27:     end if
28:   end for
29:   Run LLVM Memory to Register Promotion on All DummyPtr
30:   Run LLVM Dead Code Elimination on F
31:   for all (reg,address, I) $\in$ ADDRS do
32:     if DummyPtr(reg, address) is deleted AND DummyRegs(reg) has no uses OR
    only used in return instructions then
33:       DeadStores(F) = DeadStores(F) $\cup$ {I}
34:     end if
```

```
35:         if DummyRegs(reg) has no uses OR DummyRegs(reg) is used in all return
    instructions of F then
36:             PotRets(F) = PotRets(F) - {reg}
37:         end if
38:     end for
39: end for
```

the IR for each pair of address and register identified. We initially store a dummy value we create to each one of those memory locations. Lines 17 through 28 examine the uses of every address using VSA. At every possible read of an address, we insert a load from the dummy memory location we create. At every possible write to that address, we insert a store to that dummy memory location of the stored value. After that, we run the memory-to-register promotion compiler pass again on those memory locations. Finally, lines 31 through 38 determine the final set of dead stores. If the dummy memory location is promoted successfully to registers, and the only use of the dummy value is at the return then it is saved and can safely be removed from the potential return. The corresponding initial register stores are declared to be dead in this case. If the same previous conditions occur and also there are other uses of the dummy value, then the register is removed from the potential returns, but the initial store is not dead and is considered a real use of the register; i.e. the register becomes an argument.

The *UnsafeInstruction*(*address*) functions appearing in line 18 in the algorithm is responsible of deciding whether the instruction may have side effects which can potentially access that *address*. External calls without a known prototype where any stack address appears in the value sets of one of the arguments are considered unsafe as they may do arithmetic on those addresses and potentially read from or write to our *address*. Some external functions are pre-identified safe and known not to do arithmetic on pointer arguments, or do it with a bounded identified offset. For example, we parse format strings of `printf`, `scanf` and similar functions and in some cases we can prove those functions are safe.

After detecting the dead stores used to save registers and pruning the callee-saves from the potential returns, we proceed to step four which identifies the actual register arguments. Algorithm 5 shows the method to do so. We traverse the call graph of the executable in post-order depth-first search traversal, which ensures child nodes are visited before their parents. For each potential register argument inside a function, we declare it as an argument if and only if we see a "real" use of this register in the function. If a register is used in a store instruction among the dead stores identified by algorithm 4, the store is not considered a real use. Uses in calls are only considered "real" if the callee takes the register as an actual identified argument. A work list mechanism is maintained to handle the dependencies between functions. PHI nodes that link multiple SSA versions of the same register are not considered uses and are tracked. Returns are not considered real uses because if the return is the only use of a register, there is no need to pass it as an argument.

Propagating the actual return registers (step 5 in our algorithm) is done in a similar way to the one above except that it works on functions in the forward call graph order and looks for uses of return values at call sites.

The correctness of our register arguments and returns algorithm is guaranteed for

Algorithm 5 The algorithm to propagate register arguments

```
    Output: $PotRets$ : map between functions and their potential return registers
 1: $WorkList$ = Functions sorted in reverse call graph order
 2: while $WorkList$ is not empty do
 3:     remove a function F from $WorkList$
 4:     mark function as $started$
 5:     for all $reg \in PotArgs(F)$ do
 6:         for all Instruction I that uses $reg$ do
 7:             if I is a RealUse($reg$) then
 8:                 $RegArgs(F) = RegArgs(F) \cup \{reg\}$
 9:             end if
10:             if (I = call X) AND (X is not $started$) then
11:                 Add F to $WorkList$
12:             end if
13:             if I = call X AND X is $started$ AND $reg \in RegArgs(X)$ then
14:                 $RegArgs(F) = RegArgs(F) \cup \{reg\}$
15:             end if
16:         end for
17:     end for
18: end while
```

internal functions. The reason is that we start our algorithm initially by having all registers as arguments, and then remove those which are not really used. For returns, we start the algorithm by adding all registers that are written to inside of a function or one of its callees, we then remove the ones which are unused at call sites. The correctness in the presence of indirect calls, external calls and call backs is described below.

Our algorithm runs the same way on indirect calls and is correct. At every indirect call, SecondWrite inserts a call translator function that checks the value of the function pointer and calls the corresponding IR function accordingly. In this case, this call translator is treated the same way as any normal function in this algorithm under the assumption that the call translator will call all possible target functions. External calls are discussed separately in the following section.

3.3 External Calls Prototypes

In the previous part of this chapter, we proposed sound techniques to detect register arguments and returns. We showed the correctness of our techniques when all calls in the binary are to internal functions.

In this section, we extend our methods to support rewriting external calls correctly and making sure all required arguments are passed correctly under certain assumptions. We start first by describing why it is important to handle such calls. We then state our assumptions. After that, we move forward to describe how we can detect external calls in the original binary. We then show how we represent external functions in the IR. We show the details of our rewriting techniques of external calls. We finally prove that our techniques are sound and the external functions will receive the same arguments as in the original binary and will therefore return the same values.

3.3.1 Overview and Problem Statement

Resolving external calls while recovering IR from executables is very important. Almost no real-world binary is free from external calls. External calls are usually done to libraries to perform certain tasks that are not part of the main application stream.

Functions that are external to the application are usually found in libraries. There are two main ways to link libraries to application code: 1) Static linking. 2) Dynamic linking.

Static linking is done when the linker decides to insert the function body inside the application code itself. The advantage of that is fast execution time of the calls to such functions and no overhead in loading the application, but the price paid is the increase in the binary code size.

On the other hand, dynamic linking is when the linker decides to keep the external functions outside the binary application and refer to them by using their names (or locations) in the dynamic link library that contains them. During the application load time, the libraries are loaded with the application such that when the application calls one of these external functions, they can execute correctly. The advantage of dynamic linking is the smaller binary code size and application modularity (by keeping the binary code that is not related to the application main stream external). Another advantage is that dynamic libraries are almost always *shared* on the system. The disadvantage is usually slower binary load time.

In this section we only target recovering functional IR with external function calls that are dynamically linked to the binary. We do not target statically linked external functions in this section since they are already handled correctly by our earlier techniques since their code exists inside the binary itself.

For every external call site in the original binary, the problem we are solving here is to recover some code that replaces the original call in the IR such that all of the following is true when the IR is compiled to a rewritten binary:

1. The rewritten binary redirects control to the same external function when executed.
2. The external function takes the same memory and register arguments that were passed at the original binary call site..
3. The return value(s) if any from the external function are passed back to the rewritten binary correctly.

This problem is challenging in a static binary analyzer because of two main reasons. 1) Statically detecting an external call site in a binary is not always trivial. 2) External function prototypes and calling conventions are usually not known to a static binary analyzer. This section gives an overview of the first problem and discusses in details how the second problem can be solved.

3.3.2 Assumptions

As per any static binary analysis system, it is impossible to handle all scenarios. In this section, we discuss our main assumptions while recovering external calls.

In this work, we assume that any external function has to adhere to some known application binary interface (ABI). This means that any call site to an external function in the binary has to have a known calling convention from a finite set of supported calling conventions. More specifically, we assume all of the following is true:

1. All external functions can only take arguments either in registers or using the memory stack.
2. All external functions expect memory arguments at specific stack offsets.
3. All external functions expect register arguments only in specific registers. These registers are found in a set we call *RegArgs*
4. All external functions can only return values in specific set of registers we call *CallerSaves*.
5. Any register other than the *CallerSaves* must be saved and restored back if used by an external function.

The previous assumptions are valid in almost all compiled code. The reason is that external functions are usually shared between multiple applications and sometimes between different systems. For them to be portable, they have to adhere to a certain ABI with some specific calling convention such that it is more convenient for compilers to interface with them. In practice, we found that almost all libraries adhere to the above assumptions in all our tests.

In theory, external functions are not required to adhere to some certain ABI. Developers writing source code usually specify a prototype for every external function they use, which includes the complete calling convention of the function. The compiler then reads this calling convention and adheres to it while emitting low level binary code. One challenge in dealing with binaries as we mentioned above is the lack of this prototype.

One example we found that does not adhere to these assumptions is very few internal compiler intrinsics. These are compiler specific functions inserted in the binary to speed up specific tasks (or for other reasons). Very few of these intrinsics do not adhere to the above assumptions since they are already known to the compiler. We support this by maintaining a list of known compiler intrinsics with their custom calling conventions. We do not support other compiler intrinsics not found in this list and not adhering to the above assumptions.

3.3.3 Detecting External Function Calls

At compile time, the compiler does not know the external function addresses at their call sites. These addresses are only known when the operating system loads the binary. Because of this, the compiler has to call such functions indirectly through some memory location that the loader updates with the actual address of the called function.

There has to be a common language between the compiled binary and the operating system. The operating system has to know which memory locations to update when loading the external libraries. Different systems implement different mechanisms of handling this issue. For example, some Linux binaries use a dynamic relocation table, Windows binaries use an import address table and so on.

In theory, since the loader is able to update these memory locations, these memory locations have to be visible to any static binary analysis system. All external libraries and functions used in a binary can be statically known from the binary image.

The challenge exactly is how to determine which indirect call leads to which external function. Static binary analysis and rewriting systems usually rely on pattern matching techniques that are mostly accurate and depend on certain compiler behaviors. Some of these techniques include:

1. If the indirect call site uses the memory location that is visible to the loader (specified in the dynamic relocation entry or the import address table), then it is simple to determine which function it is calling by looking up in that table. IDA Pro [31] uses this method.

2. Some compilers implement procedure linkage tables which are composed of external function stubs. Each stub has an indirect jump to the external function. Such stubs are usually standard and can be detected accurately. The well-known Linux disassembler tool `objdump` uses this technique.

There are some other techniques compilers use to make external calls. Such techniques are outside the scope of this dissertation. Static binary analyzers can fail in detecting if an indirect call is calling an external function or not. In all our tests this happened in very rare cases, but it needs to be handled for correctness. In such cases, our IR representation of indirect calls guarantees correct rewritten binary execution as we discuss in the following section.

3.3.4 External Calls IR Representation

We divide indirect call sites in a binary into two categories: 1) The call sites that are detected to call external functions using a static binary analyzer. 2) All other indirect call sites that may or may not call external functions.

We maintain a list of the prototypes of as many external functions as possible (like the standard libraries functions). This list may not be exhaustive since for custom DLLs it is hard to ensure they are all considered.

For detected external calls, if their prototypes are known (by searching the known prototypes list), we present them as direct calls to these external functions in the IR and pass all required arguments with their correct data types. For unknown prototypes, we represent them by a call to a special function we create in the IR called the *trampoline* function. We discuss the trampoline function in the next section.

For all other indirect call sites that are not guaranteed to call external functions, we use the call translation mechanism described in our WCRE paper [63]. The call translator function is a static function inserted in the IR that has a large switch statement that redirects control from constant original function addresses to their corresponding IR functions. The number of cases in the switch statement is the number of all IR functions that can be possibly reached indirectly in the original binary. We replace the assertion in the default case of the translator function by a call to the *trampoline* function described in the next section. The translator function looks like the function in figure 3.1 in this case.

In order for the rewritten binary to work correctly, we maintain the same dynamic relocation entries (or the import address table entries) at their original locations in the

```
switch (input_address) {
case 0x400:
     call rewritten_0x400;
case 0x500:
     call rewritten_0x500;
……
default:
     call *input_address;
}
```

Figure 3.1: The call translator function

rewritten binary. We discuss this more after we describe the trampoline function.

External calls get executed in the rewritten binary the following way: first, a call to the call translator function is executed and the call translator function case statement comes to the default case (since the external call address is not any one of the original function addresses). The trampoline function gets executed and redirects the control to the external function as we discuss in the next section.

3.3.5 Trampoline Function

The *trampoline* function is a custom function only existing in the recovered IR without a corresponding function in the input binary. It is used to redirect control to external functions at external call sites in one of three cases: 1) In case the call site could not be proven to call a specific external function. 2) In case the external function has an unknown prototype. 3) Or, in case the external function has a known prototype but with a variable number of arguments.

The *trampoline* function has to do all of the following tasks:

1. Redirect control to the correct external function.
2. Pass memory and register arguments correctly from the IR call site to the external callee.
3. Pass the return value(s) correctly from the external callee to the call site.

In order for the trampoline function to work correctly, it takes the following arguments:

1. The function address being called (usually a register or a load from some memory location).
2. The *abstract stack* pointer value in the IR right before the call site.
3. The values of all registers in the set *RegArgs* before the call.
4. Pointers to place holders of the return values, one for each register in the *Caller-Saves* set.
5. Pointer to a variable holding the stack *balance number* of the external call.

The first argument above is the function address. This is usually a result of some register read or some memory load. It cannot be a direct address that is known since the call site is for an external function. This address is used inside the trampoline function to redirect control correctly to the destination external function.

The second argument is needed to adjust the memory arguments and put them into their correct offsets on the memory stack of the rewritten binary. The *abstract stack* is the stack array in the IR resembling the original stack in the input binary. More information about how the physical stack in the input binary is converted to abstract stack arrays in the recovered IR can be found in our EuroSys paper [3].

The third set of arguments are required to pass the correct register argument values to the external function. We pass all of the possible register arguments conservatively.

The fourth set of arguments are pointers to variables declared at the call sites of the trampoline function to pass the return values back to the caller. The trampoline function updates each of them.

The last argument to the trampoline function is a pointer to a variable declared at the call sites of the trampoline function to hold the stack *balance number* of the external call. This balance number is the difference between the stack pointer before and after executing the external function. The abstract stack pointer value in the caller has to be adjusted to this value after returning from the trampoline function.

```
void trampoline (fn_address, $SP_{ORIG}$, RegArgs, CallerSaves, BalNum)
{
     /* Assume the physical stack pointer register is: ESP */

     (1)  Let $SP_{CURR}$ = ESP, SIZE = $SP_{ORIG}$ - $SP_{CURR}$
     (2)  Allocate a temporary memory at $SP_{TEMP}$
     (3)  Save the contents between $SP_{CURR}$ and $SP_{ORIG}$ to the temp
          memory between: $SP_{TEMP}$ and $SP_{TEMP}$ + SIZE
     (4)  Set ESP = $SP_{ORIG}$
     (5)  Copy all RegArgs values to the physical registers
     (6)  Call the function at fn_address
     (7)  BalNum = $SP_{ORIG}$ - ESP
     (8)  Copy return register(s) to CallerSaves
     (9)  Restore back the contents between $SP_{TEMP}$ $\rightarrow$ $SP_{TEMP}$ + SIZE
          to between $SP_{CURR}$ $\rightarrow$ $SP_{ORIG}$
     (10) Set ESP = $SP_{CURR}$
     (11) Return
}
```

Figure 3.2: Pseudo code of the trampoline function

Figure 3.2 shows pseudo code of the trampoline function with the previously discussed arguments. The memory stack layout of the rewritten binary immediately after executing a call to the trampoline is shown in figure 3.3. Line (1) in the code saves the current stack pointer of the rewritten binary in a local variable. Line (2) allocates a temporary storage that is needed for saving some values as we show next.

The rewritten stack layout shown in figure 3.3 contains the abstract stack array where its TOP value is represented by the SP_{ORIG} variable. The abstract stack is the IR array that represents the input binary's physical stack of the caller. For the external call to be executed correctly, it needs to have the same stack view as what was there in the input binary, which means the stack pointer must point to the top of the abstract stack frame. The problem that happens in this case is that functions usually assume that any address that is lower than the current stack pointer at the function's entry point is free space that can be used by local variables of the function. In this particular case, the stack addresses between the top of the abstract stack pointer (SP_{ORIG}) and the top of the rewritten stack frame (SP_{CURR}) are used by the caller function in the IR as shown in figure 3.3. This means that we have to save this stack region such that if the external function allocates some stack space and corrupts this region, we can restore it back.

This is exactly what line (3) in figure 3.2 does. It saves the region on top of the abstract stack frame in the rewritten stack frame to the temporary storage.

Line (4) in figure 3.2 sets the current physical rewritten stack pointer to point to the top of the abstract stack frame of the original caller function. Line (5) copies the register arguments to the actual physical registers. Line (6) calls the external function.

At the point of the call at line (6), the physical registers have all the arguments. The memory stack view is the same as what was there in the input binary. Hence, the external function will produce the same result as the input binary.

After returning from the external call, line (7) stores the balance number by subtracting the stack pointer before and after the call. Line (8) copies the return physical registers into the IR return variables. Line (9) restores back the stack memory region that was saved in line (3). Line (10) restores the physical stack pointer to its value at the entry to the trampoline such that line (11) can return back to the correct call site.

Because the trampoline code writes to physical registers, and reads from them, it is usually written as a separate function in low level assembly, and then linked with the rewritten binary when re-compiled. This way we maintain the readability of the recovered IR by hiding all these low level details. The user will only see a call to the trampoline function with the first argument being the function that will be called (or some pointer to it).

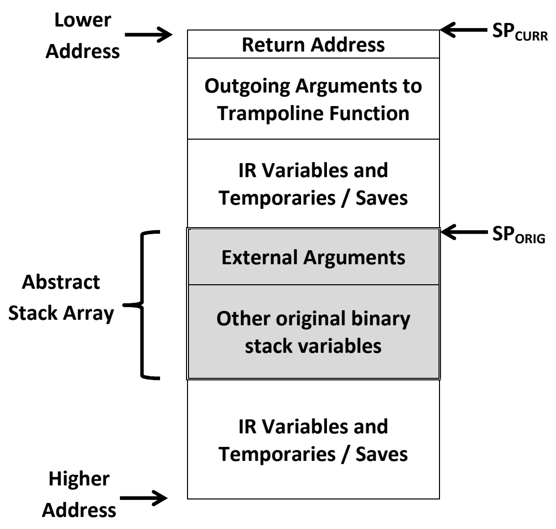

Lower Address
Return Address
SP_CURR
Outgoing Arguments to Trampoline Function
IR Variables and Temporaries / Saves
SP_ORIG
External Arguments
Abstract Stack Array
Other original binary stack variables
IR Variables and Temporaries / Saves
Higher Address

Figure 3.3: The memory stack layout after executing a trampoline call

3.3.6 Correctness of the Trampoline Function

The trampoline function in figure 3.2 achieves all the three goals stated at the beginning of the previous section under the assumptions in section 3.3.2. We clarify this here.

The first goal of the trampoline function is to redirect the control to the correct external function. This is guaranteed as long as the address of the function passed to the trampoline is correct.

We notice that the address passed to the trampoline is the address of the external function in the input binary which might be different than the address of the same function in the rewritten binary (because of different loader behavior). The key point here is that the place holders of these addresses are exactly the same in the input and the rewritten binary as we state at the end of section 3.3.4. These place holders are known to any static analyzer as they have to be visible to the loader before executing the binary. They depend on the binary format (dynamic relocation tables in ELF or import address tables in PE).

Under the assumption that any external call has to load the function address from these place holders, then the external function address passed to the trampoline function is correct and the execution will be redirected to the correct external function.

The second goal of the trampoline function is that any register or memory arguments have to be passed correctly to the external function. Register arguments are passed correctly in line (5) of the function in figure 3.2. Memory arguments are passed correctly since the external function has the same stack view as the original binary's stack view which is guaranteed by changing the stack pointer value in line (4) of the function to point to the top of the abstract stack frame (which contains all arguments).

The third goal of passing the returns correctly is achieved by executing line (8) of the trampoline function.

3.4 Effect of inaccurate function boundaries

In this section, we discuss the effect of having inaccurate function boundaries on the function API recovery process described in this chapter. As per chapter 2, the IR might have inaccurate function boundaries as well as spurious functions that have to work correctly in all cases for guaranteed functionality of the IR as we discussed in chapter 2.

This section is divided into two parts, the first part addresses the modifications required for the previous stack memory arguments identification techniques for SecondWrite presented in [3]. The second part talks about the register arguments identification techniques presented in this chapter.

3.4.1 Memory Stack

When the function boundaries are not accurate, the assumption that every function has a return address on top of its physical stack (which is the stack memory in the original binary) is no longer valid. This assumption is used in the EuroSys work recovering abstract stack and arguments from physical stack accesses [3]. This assumption is not usually valid in spurious code.

In this section, we show that modifications are needed to the previous high level symbol promotion work [3] in case procedure boundaries are not accurate. We first introduce a summary of our previous technique and why it cannot work in case of inaccurate procedure boundaries (spurious code), then we proceed with the modified technique to overcome that and prove that it will work.

Our previous symbol promotion work aims at converting the physical stack frame in the original binary to a set of abstract stack frames (which are IR arrays representing the physical stack) for every recovered procedure in the IR. This is an important step of the high level IR recovery. The way we do that is that we allocate a local array in every IR procedure with a size that is equal to the maximum allocated stack size in this procedure. This maximum size can be a fixed constant or a non-constant expression. In case we cannot come up with an expression, we do not convert the physical stack into abstract stacks.

Instead of describing the details of the previous techniques, we give an example of why they will not work in case of inaccurate procedure boundaries. Consider the code example shown in figure 3.4-a. In this example, function *foo* is split into two parts in the IR *foo* and *foo_split*. Assume the indirect jump in *foo* was compiled from some case statement whose targets were not known in the IR and one of its targets is: *foo_split*. Assume the indirect call is calling the actual procedure *bar*. *foo_split* is not an actual procedure in the input binary and will not have any return address allocated at the top of its stack in the IR as it is originally a part of *foo*. *bar* will have a return address allocated on top of its stack by the call instruction. If we use the rules described in our previous paper [3], and assuming the return address is four bytes long, the local variable access at offset 5 in *foo_split* will be translated as index: (5-4) = 1 in *foo*'s IR abstract stack array to account for the return address (which does not exist in this case). This will result in a wrong access since the right one is actually 5 (without subtracting 4 bytes for the return address). On the other hand, the access to offset 9 in *bar* will be translated correctly to offset (9-5-4) = zero in the recovered stack array of *foo* because *bar* has a return address stored on top of its stack (the call instruction pushes that return address on the stack). The recovered code which will not work correctly is shown in figure 3.4-b.

```
foo:
sub $10, %esp
…
movb $4, 5(%esp)
jmp *eax
…
//Arg moved to TOP
movb %dl, (%esp)
call *ebx
…
…
//not a function
foo_split:
movb 5(%esp), %ebx
…


bar:
sub $5, %esp
movb 9(%esp), %edx
… //No calls
```

(a)

```
foo () {
     char MStack[10];
     …
     MStack[5] = 4;
     call_translator (eax, &MStack[0]);
     …
     MStack[0] = dl;

     call_translator (ebx, &MStack[0]);
     …
}

foo_split (char* Parent_Stack) {
     char ebx = Parent_Stack[1];
     …
}

bar (void* Parent_Stack) {
     char MStack[5];
     char edx = Parent_Stack[0];//Arg1
     …
}
```

(b)

```
foo () {
     char MStack[14];
     …
     MStack[9] = 4;
     call_translator (eax, &MStack[4]);
     …
     MStack[4] = dl;
     MStack[0] = ret_address;
     call_translator (ebx, &MStack[0]);
     …
}

foo_split (char* Parent_Stack) {
     char ebx = Parent_Stack[5];
     …
}

bar (char* Parent_Stack) {
     char MStack[5];
     char edx = Parent_Stack[4];//Arg1
     …
}
```

(c)

Figure 3.4: Stack Functionality. a) The input binary. b) The broken recovered IR using previous techniques. c) The correct recovered IR.

To avoid this problem, we introduce our modified physical to abstract stack translation rules and prove they will work in all cases even for split functions. Before we begin, we introduce some notation. We assume a general case in the recovered IR of a call stack (chain of recovered procedures reachable using calls/jumps in the original binary) with length n where the entry point procedure is referred to as index zero in this chain. The IR procedures in this chain may or may not be actual procedures in the original binary. The following notation will be used:

- *SP*$_x$ refers to the physical stack pointer value in the input binary at the entry point of procedure x in the chain before executing its first instruction.

- *ST*$_{(i,x)}$ refers to the physical stack pointer value in the input binary after executing instruction i inside procedure x in the chain. In case i is a call instruction, *ST*$_{(i,x)}$ represents the stack pointer value after executing the call and pushing the return address, but before jumping to the callee.

- *MStack*$_{x_1}[x_2]$ is the recovered abstract stack array in the IR indexed at x_2 for procedure x_1 in the chain.

- *SZ*$_x$ is the recovered size of *MStack*$_x$ array in bytes.

First, the recovered size of *MStack*$_x$ is calculated as follows: *SZ*$_x$ = max(*SP*$_x$ - *ST*$_{(i,x)}$) $\forall i \in$ procedure x in the chain. This means the maximum growth in the stack in this particular procedure. The growth of the stack pointer value includes all growth due to any kind of stack pushes including the return address push that happens with a call instruction. This is a major fix to the previous work [3] that does not consider the return address push during a call as a stack allocation.

Next, we assume the following relation holds for any recovered procedure n in the chain: *SP*$_n$ =*SP*$_{n-1} - Y_{n-1}$ where $Y_n \in Z, Y_n \geq 0$. Y_n represents the stack offset immediately before procedure $n-1$ jumps to procedure n. This relation means that the physical stack grows in the negative direction when procedure $n-1$ calls procedure n. This is valid in most compiled code.

For any physical stack pointer value that can be represented as: *ST*$_{(i,n)}$ = *SP*$_n$ -

X_n in the input binary where $X_n \in Z$, it will be translated to IR access *MStack*$_{n'(X_n,n)}$ [*SZ*$_{n'(X_n,n)} - T_n(X_n)$] where:

$$n'(X_n, n) = \begin{cases} n, & X_n > 0 \\ n'(X_n + Y_{n-1}, n-1), & X_n \leq 0 \\ Where \qquad \textit{SP}_n = \textit{SP}_{n-1} - Y_{n-1} \end{cases}$$

$$T_n(X_n) = \begin{cases} X_n, & X_n > 0 \\ T_{n-1}(X_n + Y_{n-1}), & X_n \leq 0 \\ Where \qquad \textit{SP}_n = \textit{SP}_{n-1} - Y_{n-1} \end{cases}$$

In the above recursive equations, $T_n(X_n)$ represents the offset from the end of the recovered abstract stack of procedure n for a particular physical offset X_n in the original binary. n' is some ancestor procedure of n in the call stack. $T_n(X_n)$ is measured from the end of the abstract stack array because the physical stack grows backwards as per our assumption above. For every physical stack pointer value *ST*$_{(i,n)}$ there are two important definitions: 1) The originating procedure is the IR procedure from which this access originates in the chain which is procedure n. 2) The landing procedure which is the IR procedure whose abstract stack is the one accessed by translating this stack pointer value. We refer to that procedure as n' in our notation. If $n' = n$ then the originating procedure and the landing procedures are the same which means the stack access refers to a local variable inside procedure n. On the other hand, if $n' \neq n$ this means that the stack pointer value refers to an argument obtained from the parent procedure n'.

The intuition behind the above equations is: in cases where $X_n > 0$ the access is inside the local abstract stack frame because the stack grows in the negative direction, any positive value subtracted from the stack pointer value at the entry point of the procedure means a local allocation. It has to be translated to the same offset. On the other hand, $X_n \leq 0$ represent positive offsets relative to the stack pointer at the beginning of the procedure which means they are previously allocated in a parent abstract frame.

To implement this correctly in the recovered IR, a call instruction in the original binary will be translated into a store of the return address into the location in the recovered abstract stack array representing the stack pointer value immediately after the call according to the above translation rules. This stack array location is passed as a pointer to the caller. For jump instructions, the stack array location will be passed without storing any return address. This can be shown in the code example in figure 3.4-c.

In cases when the stack size is not constant, or constant stack offsets cannot be inferred from the binary, the translation rules above are implemented as runtime checks in the same way described in [3] but with the new translation rules described above.

To prove this will always be correct, we prove that the recovered abstract stack exactly resembles the original physical stack. Instead of comparing absolute values of the physical and abstract stacks (which are runtime values), we compare relative values on the physical stack (stack differences) and prove they are exactly equal to the offset on the abstract stack. The next lemma proves this.

Definition 3.1 A functional binary is a binary where there does not exist a memory access accessing stack locations not allocated inside the binary. *i.e.* for any stack access represented as: $ST_{(i,n)} = SP_n - X_n$, $n'(X_n, n) \geq 0$ and $X_n \leq SZ_n$

Preposition 3.1 Any stack pointer value has to access either the same stack frame or an ancestor stack frame in the IR. ***i.e.*** $n'(X_n, n) \leq n$

Proof Follows directly from the mathematical definition of n'.

Lemma 3.1 For any n chain of IR procedures reachable using calls/jumps and forming a call stack in the recovered IR of a functional binary, and for any instruction i in the original binary, the physical stack pointer value at i relative to the physical stack pointer value at the entry point of the landing procedure is exactly equal to the translated abstract stack offset of that physical stack pointer value if the translation rules above are used.

This Lemma can be formulated as follows: for a particular physical stack pointer value at instruction i represented as $\mathit{ST}_{(i,n)} = \mathit{SP}_n - X_n$, $\mathit{SP}_{n'(X_n,n)} - \mathit{ST}_{(i,n)}$ is exactly equal to $T_n(X_n)$.

Proof For cases where $n'(X_n, n) = n$ the proof is trivial by doing simple substitution in the formulas and the definitions above.

For cases where $n'(X_n, n) \neq n$, we use mathematical induction. We prove the relation at $n = 1$, assume it is valid at n and prove it at $n + 1$. In the proof, our left hand side (LHS) is $\mathit{SP}_{n'(X_n,n)} - \mathit{ST}_{(i,n)}$ and our right hand side (RHS) is: $T_n(X_n)$.

Base Case: $n = 1$ In this case, $n' = 0$ according to preposition (1). LHS = $\mathit{SP}_0 - \mathit{SP}_1 + X_1 = \mathit{SP}_0 - (\mathit{SP}_0 - Y_0) + X_1 = Y_0 + X_1$ RHS = $T_1(X_1) = T_0(X_1 + Y_0) = X_1 + Y_0$ which is the same as the LHS.

Inductive Case: Assuming: $\mathit{SP}_{n'(X_n,n)} - \mathit{ST}_{(i,n)} = T_n(X_n)$ we want to prove that: $\mathit{SP}_{n'(X_{n+1},n+1)} - \mathit{ST}_{(i',n+1)} = T_{n+1}(X_{n+1})$

LHS = $\mathit{SP}_{n'(X_{n+1},n+1)} - \mathit{SP}_{n+1} + X_{n+1} = \mathit{SP}_{n'(X_{n+1}+Y_n,n)} - (\mathit{SP}_n - Yn) + X_{n+1}$

Assume $X_{n+1} + Y_n = P_n$, LHS = $\mathit{SP}_{n'(P_n,n)} - (\mathit{SP}_n - P_n) = T_n(P_n) = T_n(X_{n+1} +$

$Y_n) = T_{n+1}(X_{n+1})$ = RHS

Lemma (3.1) shows that the relative offsets between the physical stack pointer values in the original binary are exactly the same as the relative offsets on the recovered abstract stack arrays. This implies correct stack memory behavior in all cases under the assumptions stated earlier in this section.

3.4.2 Register Arguments

Earlier in this chapter, we presented a sound technique that can be used to recover register arguments for any function in the binary. The technique is based on tracking memory locations on the stack used to save and restore registers. The memory tracking is done using a simplified version of the Value Set Analysis (VSA) technique [5] that runs on the IR before the identification process takes place.

Our previous techniques for detecting register arguments will still be correct and sound for spurious code provided that the Value Set Analysis is run on the IR after converting the physical stack into an abstract stack using the translation rules discussed in the previous section. This is necessary to ensure that the stack memory values flow correctly to spurious functions (as well as other functions). If the older stack translation presented in our previous EuroSys work [3] is used, it might lead to inaccurate flow of values to spurious code as discussed in the previous section.

3.5 Results

3.5.1 Register Arguments and Returns

In this section we show the accuracy of the detected register arguments and returns. We get our results only for the C and C++ benchmarks shown in figure 2.8 and present the average number of added register arguments and returns (false positives). We never had any false negatives in any of the binaries we tested. We could not compare Fortran binaries since currently, we do not support reading Fortran prototypes from debugging information.

As shown from the figure 3.5, the average number of false positive arguments is 0.2 per function. The average number of false positive returns is 0.44 registers per function.

While collecting these results, we assume that return registers are either `eax` or `edx` or both in x86. This is valid in all the code we know of that runs on x86 systems since this is a standard ABI feature for x86. Our main algorithm does not require this feature to be functional.

The number of false positive return registers is higher because the return registers identification process is usually less accurate than register arguments identification process. To see why this is true, consider a function `foo` that is called 10 times in the whole binary in different call sites. To identify register arguments to `foo`, only the entry point of `foo` has to be analyzed for uses of arguments. To analyze for `foo` returns, all 10 callsites have to be analyzed for real uses of the return registers. Since the number of callsites of functions in the binary is usually larger than one, the return registers identification process is likely to introduce more false positive returns.

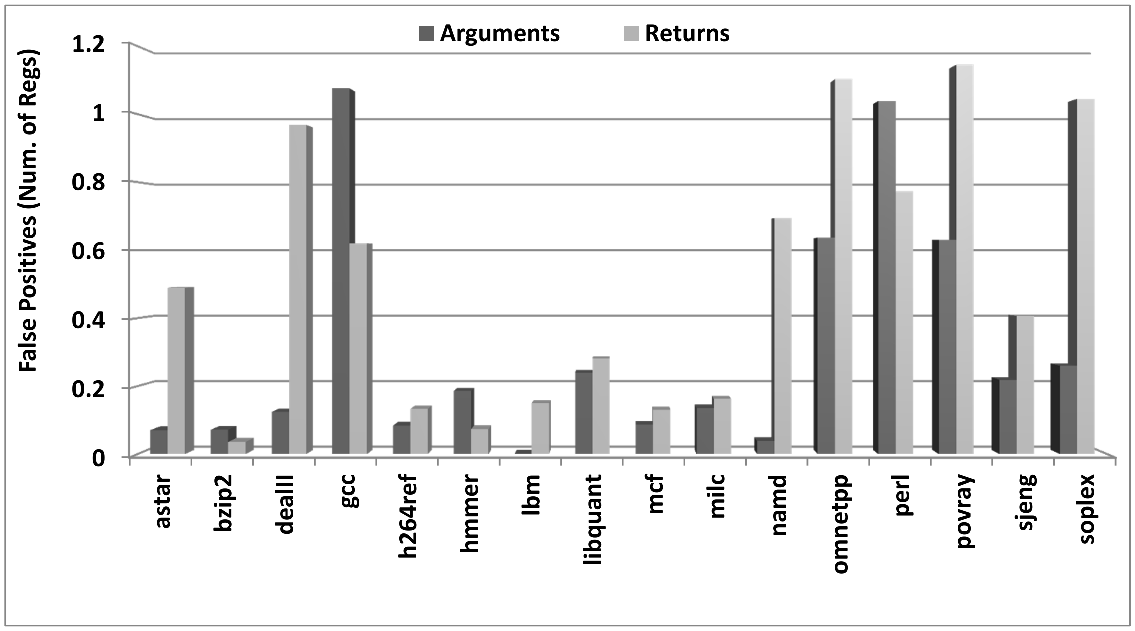


Figure 3.5: Accuracy of register arguments and returns

In contrast to the work in [10], our method has three advantages: (i) it is guaranteed to discover all arguments; (ii) it has been demonstrated on a much larger programs; and (iii) it is orders of magnitude faster. First, their method cannot guarantee full coverage of arguments and returns because of being a dynamic analysis. Any unused argument or return during an execution trace can be missed. Missing arguments or returns is acceptable for human understanding of binaries, but unacceptable for rewriting binaries. Second, our method has been evaluated on far more functions (48,854 functions for our method, vs. just 13 functions for theirs.) Third, our analysis is much faster: for example, it takes only 30 seconds to analyze a program like *soplex* which has 116,743 instructions containing 1,523 procedures and produces prototypes for all of them. In their case, they need the same 30 seconds to only extract `MD5_Final` which is a single function of 67 instructions. This shows that our analysis is two to three orders of magnitude faster than their method, at the expense of a small loss in precision.

3.5.2 Trampoline Function Overhead

In this section we show the effect of inserting the trampoline function discussed in section 3.3 on the overall performance of the rewritten binary.

As we discussed in section 3.3, the trampoline function is inserted to guarantee correct execution of external functions that do not have a known prototype, or for functions with variable number of arguments. The trampoline function adds some overhead of restoring the state of the rewritten binary as it was in the original binary before running the external call.

We measure the time spent in the trampoline function relative to the whole run time of the rewritten binary. We do not consider the time spent in the external function itself since this time is not considered an overhead. We use the Perf Linux performance monitoring tool to measure the overhead. All original binaries were compiled using the maximum optimization level when performing this experiment.

As shown from the figure 3.6, the overhead of the trampoline function is negligible in all cases. The average overhead is 0.18% of the runtime of the rewritten binary. In many cases, there is no overhead at all when the binary does not have any external call with unknown prototype or a variable number of arguments. This shows that the cost of achieving correctness in case of rewriting external functions that do not have a known prototype is very small.

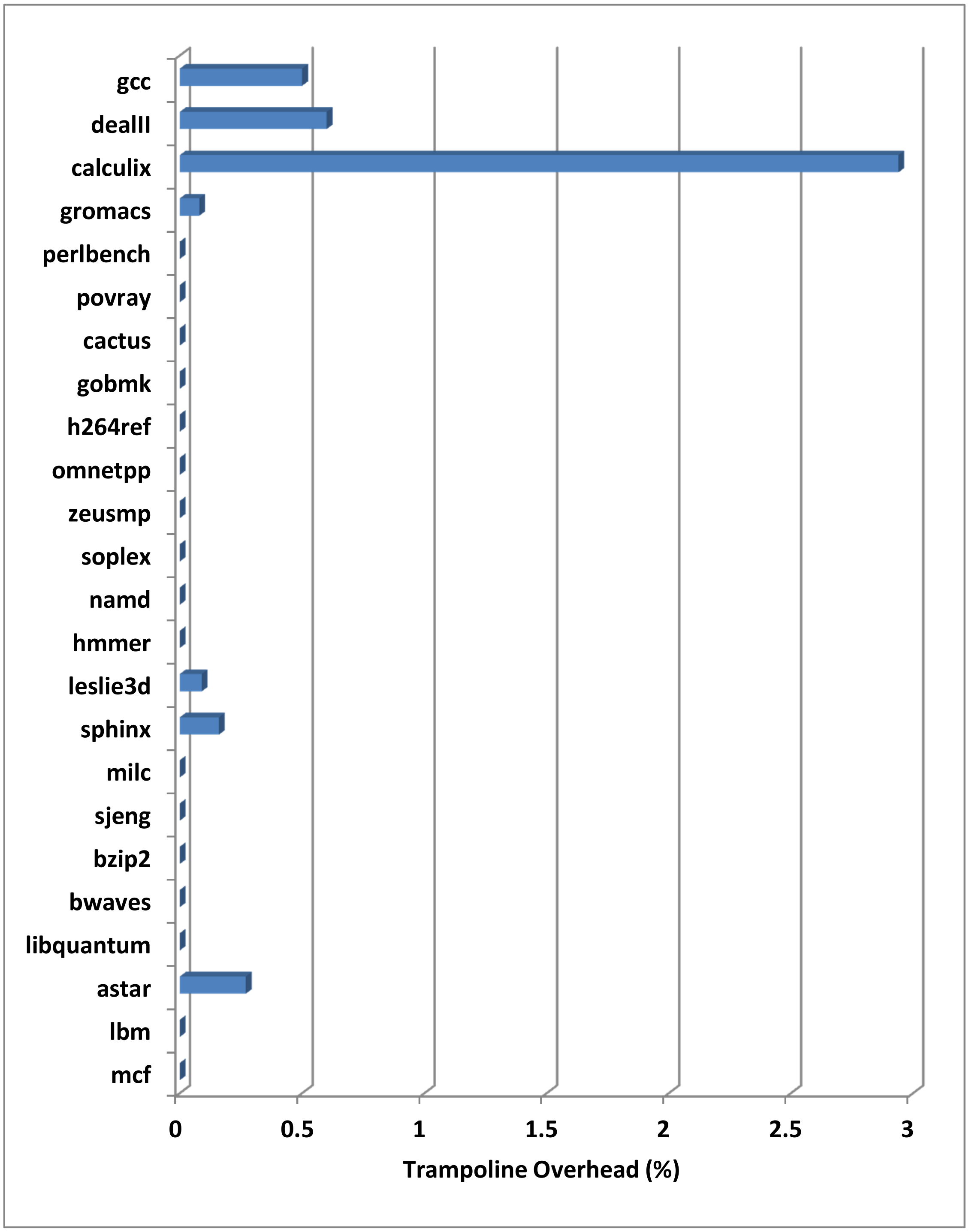


Figure 3.6: Overhead of the trampoline function in the rewritten binary

We do not show the runtime of the rewritten binaries compared to the runtime of the original binaries here since it involves many other factors in the SecondWrite system that are outside the scope of this dissertation. This is illustrated in our previous work [3].

Some binaries use external compiler intrinsics to achieve some tasks. Those intrinsics we do not have prototypes for. This behavior is shown in calculix binary which has 3% trampoline overhead. In calculix, lots of low level Fortran intrinsics are used and SecondWrite does not have their prototypes and hence it used the trampoline function. They get called with high frequency.

The other main reason some binaries have larger trampoline function overhead is calling the printf family functions (like printf, scanf, fprintf, ...). Some binaries like gcc, dealII, astar and sphinx use them more often than other binaries. Those functions have a variable number of arguments and hence SecondWrite uses the trampoline function while rewriting them. For such printf family functions, we have implemented some static techniques that are now under test which can detect the format string and extract the exact number and data types of arguments; this removes the need to insert the trampoline function for these cases. The results above do not reflect this implementation.

3.6 Related Work

Cifuentes and Simon present techniques to recover procedure abstraction from binaries [17]. They present an abstraction language that can specify machine independent prologue patterns, epilogue patterns, stack frames, argument locations, return value registers, and other issues related to procedure calls. Their abstraction depends on specifying a certain ABI that the binary has to follow. This abstraction language is used in the UQBT [16].

In her PhD dissertation, Cifuentes [15] presents a simple technique based on liveness analysis to detect register arguments and returns. This technique is implemented in the dcc decompiler.

Zhang et al. present a technique to recover function arguments and returns from executable [78]. Their technique is similar the brute force technique described in section 3.2 which leads to imprecise results.

Chapter 4: Recovering Floating Point Stack Allocated Variables

4.1 Introduction

In this chapter, we describe our techniques to convert all the x86 floating point stack operations into higher level code that uses floating point variables, function arguments and function returns, instead of the low level stack layout used in the assembly. We present sound techniques for this process and prove that they work in all scenarios under certain assumptions that are stated clearly in this chapter.

This section has five main parts. Section 4.2 discusses the x86 floating point stack and how it is maintained in the x86 executables. Section 4.3 discusses the assumptions based on which we develop our techniques. Section 4.4 describes a basic recovery technique that can work in all cases given that all indirect branches are resolved correctly from the executable. Section 4.5 discusses essential techniques that are necessary in case some indirect branches are unresolved in the binary. Finally in section 4.6, we prove that the stated techniques can work under the stated assumptions. Results as well as related work are presented at the end of the chapter.

4.2 x86 Floating Point Stack Layout and Problem Overview

We begin by introducing the x86 floating point stack. The floating point hardware stack has a maximum height of 8 which means there are only 8 physical floating point registers that can be used at any time. The names of those registers, as used by the hardware instructions, are dynamic and are relative to the current top of the floating point stack.

If we assume the fixed physical register names are: PST_0 - PST_7, then the x86 assembly instructions will refer to another set of names ST_0 - ST_7, where ST_0 always refers to the register at the top of the stack. For example, if the height of the stack is one, then ST_0 refers to PST_0. If the stack is full (with stack height of eight), then ST_0 refers to PST_7. In general, ST_x is mapped to PST_y where $y = \textbf{\textit{TOP}}(\mathrm{I}) - 1 - \mathrm{x}$ where $\textbf{\textit{TOP}}(\mathrm{I})$ is the stack height at instruction I and $0 \leq y < \textbf{\textit{TOP}}(\mathrm{I})$.

Whenever a function returns a floating point value in a register, it pushes the value on the floating point stack. Whenever a function takes floating point values as arguments in registers, the caller pushes the values on the floating point stack. It is assumed that $\textbf{\textit{TOP}}(\mathrm{I})$ cannot be negative at any instruction I.

In the recovered intermediate representation (IR), we create floating point variables corresponding to every physical floating point register in the hardware. For simplicity, we use the same physical stack register names PST_0 - PST_7 to refer to the IR floating point variables as well. Such variables are declared as local variables for every recovered function in the IR.

Decoding the floating point stack operations means mapping every assembly operand among ST_0 - ST_7 into a corresponding IR register among PST_0 - PST_7. It turns out from the previous equations that we only need to identify for every instruction I, what is the corresponding *TOP*(I) in order to decode the floating point operands successfully. Figure 4.1-c shows an example of the recovered output of the decoding process of the assembly instructions in figure 4.1-b. The *TOP*(I) values are shown in figure 4.1-a.

`TOP Value:`	`Foo:`	`double foo (`
		`double arg1, double arg2)`

`0`		`{`
		`     double PST0,PST1,temp;`

`1`	`fld 0x08(%ebp) //push arg1 → st(0)`	`PST0 = arg1;`
`2`	`fld 0x10(%ebp) //push arg2 → st(0)`	`PST1 = arg2;`
`2`	`fadd %st(1)    //st(0)=st(0)+st(1)`	`PST1+=PST0;`

`2`	`fxchg %st(1)   //st(1) ⇔ st(0)`	`temp=PST1; PST1=PST0;`
		`PST0=temp;`

`1`	`fstp 0x8(%esp) //pop st(0)→ Memory`	

`1`	`ret`	`return PST0;`
		`}`

(a)	(b)	(c)

Figure 4.1: Floating Point Stack Illustration: a) TOP values b) Original assembly code c) Recovered Code

If there is no indirect or unknown control transfer instructions in the program, the floating point stack decoding problem is trivial because we can traverse the control flow of the program, tracking the floating point stack height at every point, and set the value of *TOP*(I) at every instruction I depending on the floating point operations observed. This analysis will not work in the presence of indirect and external control transfers because when we hit such transfers, we will not know what code will be executed next and how the height of the stack will be affected by this control transfer. The following sections describe how to overcome this problem.

4.3 Floating Point Stack Assumptions

It is statically indeterminable to be able to decode the floating point operations correctly in all cases in the presence of unknown indirect control transfer instructions. In this work, we show that if we make some assumptions, we can actually guarantee a correct and functional representation of the floating point stack operations in all cases that adhere to those assumptions. Our assumptions are:

1. At control-flow join points, the floating point stack height must be the same for every predecessor basic block.

2. At indirect and external calls, the floating point stack height must be zero before the call.

3. Every indirect or external call can return at most a single floating point value on the floating point stack.

The above assumptions are correct in compiled code in every case in every compiler we are aware of. They are also true in most hand written assembly code, but may not be always true in theory. The justifications for the assumptions for compiled code are as follows:

1. If the stack height is not balanced at join points, any subsequent floating point stack access will be indeterminable as it might access different values depending on the path taken at run time.

2. For indirect and external calls, the behavior of their targets is usually unknown to the compiler, and hence the compiler must assume they might use all the floating point stack registers. As a result it has to clean the stack before such calls. We can state this assumption by saying we assume floating point registers are scratch registers. Theoretically, a compiler might know in some cases the behavior of the functions being called and may not clean the floating point stack, but practically we are not aware of such a compiler.

3. The assumption that the maximum number of floating point register returns equals one comes from the fact that we are not aware of any calling convention that allows the return of more than one floating point stack register from indirect calls and externals.

4.4 Basic Approach for Decoding the Floating Point Stack

In this section, we describe our basic approach to decode the floating point stack. In this basic approach, we assume that all targets of indirect branches in the executable are known. This assumption is relaxed in the next section. Resolving the targets of indirect branches in compiled executables can be done using efficient heuristics like the ones described in [14].

To solve the floating point stack decoding problem, we use a symbolic analysis scheme by maintaining a symbolic value X_i for every indirect and external call i representing the difference of the floating point stack height before and after the call. Sometimes we refer to that difference as *StackDiff* in this chapter. After doing the symbolic analysis, each *TOP*(I) will become a symbolic expression in terms of the X_is. We build symbolic linear equations to solve for X_is. Once the X_is are calculated, *TOP*(I) will be known for every instruction.

We translate the above assumptions into the symbolic analysis propagation rules present in figure 4.2, explained as follows. For internal function calls, we use helper variables $Y(F)$ to represent the symbolic expression representing *StackDiff* of every function F. The executable is traversed in a depth first search manner starting from the entry point function for the binary, and from functions that are never called directly in the code. The assumptions (1) through (3) in section 4.3 above represent the symbolic equations in lines (1) through (3) in figure 4.2. The actual values of X_is can only be zero or one because before the indirect and external calls, the stack height is zero according to assumption (2), and the call can return at most one value according to assumption (3). The height of the stack cannot go negative and hence the actual value of the X_is cannot be negative.

The symbolic equations represented by equations (1) through (3) in figure 4.2 along with the symbolic unknowns X_is are transformed into a linear system of equations. To solve these equations, we employ our custom linear solver that categorizes the equations into disjoint groups based on the variables used in every equation, and then solves every group only if the number of equations is equal to the number of unknowns. We keep propagating calculated values to other groups until no more calculated values are present. Most of the X_is are usually solved using equation (3) in figure 4.2.

The remaining unknowns are assumed to take a value of $X_i = 1$ conservatively. This will be always correct because from our third assumption in section 4.3 above, the stack height is either zero or one after every indirect and external call. In this case, if we

Unknown Symbolic Values :

X_i, where X_i = *StackDiff* of indirect/external callsite i

Helper Variables :

$Y(F)$ = *StackDiff* of function F, where F is an internal function

TOP(I) = top of the stack after executing instruction I

TOP$_{\mathrm{b}}(\mathrm{I})$ = top of the stack before executing instruction I

Initial Conditions :

Root functions “not called directly anywhere” as well as the entry point function have entry *TOP*$_{\mathrm{b}}(\mathrm{I}) = 0$ where I is the first instruction of those functions.

Data flow rules :

For every instruction I:

```
if $I$ = push ... ⇒ $TOP(\mathrm{I}) = TOP_{\mathrm{b}}(\mathrm{I}) + 1$
if $I$ = pop ... ⇒
    if ($TOP_{\mathrm{b}}(\mathrm{I}) = \mathrm{X_i}$) $X_i = 1$ ——————(3)
    $TOP(\mathrm{I}) = TOP_{\mathrm{b}}(\mathrm{I}) - 1$
```

```
if $I$ = call $F \Rightarrow$
    if ($F$ is an external or indirect)
        $\textit{TOP}_{\mathrm{b}}(\mathrm{I}) = \mathrm{zero}$ ———————— (2)
        $\textit{TOP}(\mathrm{I}) = \mathrm{X_i}$
    else
        $\textit{TOP}_{\mathrm{b}}(\mathrm{A}) = \textit{TOP}_{\mathrm{b}}(\mathrm{I})$ where $A$ is the first instruction in $F$
        Analyze $F$ to get $Y(F) = func(X_1, ..., X_n)$
        $\textit{TOP}(\mathrm{I}) = \textit{TOP}_{\mathrm{b}}(\mathrm{I}) + \mathrm{Y(F)}$
if $I$ = jmp $L \Rightarrow$
    Assume $L$ points to a set of known targets $S$
    $\forall i \in S$ let instruction $I_i$ be the instruction at $i$
        $\Rightarrow \textit{TOP}_{\mathrm{b}}(\mathrm{I}) = \textit{TOP}_{\mathrm{b}}(\mathrm{I_i})$ ———————— (1)
if $I_R$ = return from $F \Rightarrow$
    $Y(F) = \textit{TOP}_{\mathrm{b}}(\mathrm{I_R}) - \textit{TOP}_{\mathrm{b}}(\mathrm{A})$, $A$ is the first instruction in $F$
    $\forall Z =$ return from $F \Rightarrow \textit{TOP}_{\mathrm{b}}(\mathrm{Z}) = \textit{TOP}_{\mathrm{b}}(\mathrm{I_R})$
    $\forall I \in C$ where $C$ is the set of call sites of $F \Rightarrow \textit{TOP}(\mathrm{I}) = \textit{TOP}_{\mathrm{b}}(\mathrm{I_R})$
if $I$ = any other instruction
    $\textit{TOP}(\mathrm{I}) = \textit{TOP}_{\mathrm{b}}(\mathrm{I})$
```

Figure 4.2: Data flow rules used to decode the floating point stack

declare by mistake that a particular indirect call has one element on the stack top after its return; this element will never be accessed. In this case, even if there are subsequent floating point stack operations, they have to push values on the stack before reading them.

The floating point register arguments and returns are declared in the IR as follows: a) Whenever a function has *TOP*$_\mathrm{b}(\mathrm{I}) > 0$ at its entry point instruction I, the function is declared in the IR to take as many floating point values as the value of *TOP*$_\mathrm{b}(\mathrm{I})$. They will be passed as arguments and copied to the correct local variables according to the mapping we described earlier. b) Whenever X_i or $Y(F)$ are greater than zero at a call site, this call site will be returning as many float returns as the $Y(F)$ or the X_i values in the IR and they will be copied to the corresponding local variables in the callers.

4.5 Decoding the Floating Point Stack in the Case of Unresolved Indirect Jumps

We say that an indirect jump in the binary is not resolved when the jump table identification heuristics used during disassembly (such as in [14]) fails and thus no target addresses are statically identified for the indirect branch. When this happens, portions of the input binary can be disassembled as separate functions in the IR, but in fact they are just some targets of unresolved branches in the original binary. This causes a problem to our floating point analysis technique described above since the floating point height after indirect jumps will not be known. We refer here to both indirect unconditional jumps as well as indirect conditional branches but not indirect calls. Indirect calls are handled correctly in the previous section by assuming the TOP value after them is either zero or one. *In the rest of the section, we use the term 'branch' to refer to both conditional branches and unconditional jumps*.

Applying the same indirect and external calls technique described in the previous section will not work for unresolved indirect branches. To see why, consider equation (2) in figure 4.2 which sets the TOP variable to zero before indirect calls. If we set TOP to be zero before unresolved indirect branches, we are implicitly assuming that the corresponding case statements in the original source code cannot have register-allocated floating point variables defined before the case statement and later used inside the case statement. Any optimizing compiler can invalidate this assumption which leads to a problem. Since the jumps are one way transfers, putting any kind of constraint on the TOP value after the jump instruction is not feasible.

To clarify the issues with indirect jumps that are unresolved, consider the code example shown in figure 4.3-b. In this code, function *foo* has an indirect jump, and let us assume that one of its targets (located at label *A* in the figure) is unresolved. For a binary analysis tool with a complete code coverage (like SecondWrite [63]), the code at *A* will be recovered as part of a new function in the IR since no direct or indirect control transfer instruction was detected to reach *A*. By looking into what the code does, the part before the indirect jump pushes two elements on the floating point stack, and then later after the jump these two elements are being added and the added value is being returned from the function. If we assume a value of zero for TOP before the indirect jump, it will be not true since there are two elements present on the stack at this point.

Before we describe our method to handle such a case, we mention here that current known heuristics to resolve jump tables from binaries are very accurate. For example,

```
TOP Value:   foo:                                       double ST[8], TOP;
                                                        void foo (double arg1,
0                                                                   double arg2) {
                                                              double PST0,PST1,temp;
1            fld 0x08(%ebp) //push arg1 → st(0)              PST0 = arg1;

2            fld 0x10(%ebp) //push arg2 → st(0)              PST1 = arg2;

2            jmp *%ecx      /* jumping to A:                 TOP=2; ST[0]=PST1;
                                 (part of Foo) */             ST[1]=PST0;
                                                              call_translator (ECX);
                                                              return;
                                                        }


X            A:                                         void A() {

X            fadd %st(1)    //st(0)=st(0)+st(1)               PST1+=PST0;

X            fxchg %st(1)   //st(1) ⇔ st(0)                   temp=PST1; PST1=PST0;
                                                              PST0=temp;
X-1          fstp 0x8(%esp) //pop st(0)→ Memory

X-1          ret                                              return PST0;
                                                        }
    (a)                      (b)                                   (c)
```

Figure 4.3: Floating Point Stack Problem with Indirect Jumps: a) TOP values b) Original assembly code c) Recovered Code

SecondWrite uses a modified version of the heuristics described by Cifuentes and Emmerik [14] which leads to almost 100% accurate recovery in binaries compiled from two compilers (GCC and Visual Studio). This shows that the technique described in the previous section handles most of the cases. The techniques described here are needed in case such heuristics fail (may be for a compiler not known to the community, or for hand coded assembly). Our method is important since without it, even a single unresolved branch may result in non-functional recovered IR from the binary.

In order to solve this problem, we perform a check first to see if we need to implement this technique and not the other one described in the previous section. The check algorithm is shown in algorithm 6. The algorithm returns false meaning that no adjustments are needed if all indirect jumps are resolved. In this case, the technique in the previous section is enough. If at least one indirect jump is not resolved, then for every recovered function in the IR having no direct call or direct jump to it, we check to see if there is any instruction accessing the floating point stack (by running the function *AccessesFPStack*. If so, then this IR function is identified in the *Adjusts* set for later processing.

The function *AccessesFPStack*(*F*) conservatively returns true for any function *F* which cannot be analyzed statically to determine if it uses the floating point stack. Examples of such functions include IR functions which have unresolved indirect jumps dominated by the function entry. One target of the indirect branch might use the floating point stack and we should assume that for correctness.

Once the check in algorithm 6 returns true, we proceed to apply the same data flow rules as in figure 4.2 but with the modified unknown variables and initial conditions as shown in figure 4.4. We only set the executable's entry point TOP value to be zero. We do

Algorithm 6 Algorithm to check if adjustments are needed to the floating point stack recovery

```
1: Input: $Funcs$ : a set of recovered IR functions with their complete bodies reachable
   only indirectly
2: Input: $Jumps$ : a set of unresolved indirect jumps in the IR
3: Output: $adjustmentNeeded$ : a boolean representing the need to adjust the floating
   point stack for unresolved indirect jumps
4: Output: $Adjusts$ : A set of functions detected to have floating point accesses
5: adjustmentNeeded=false
6: if $Jumps \neq \phi$ then
7:     for all $F \in Funcs$ do
8:         if $AccesseFPStack(F)$ then
9:             $adjustmentNeeded$=true
10:            $Adjusts = Adjusts \cup F$
11:        end if
12:    end for
13: end if
```

Unknown Symbolic Values :

X_i, where X_i = *StackDiff* of indirect/external callsite i

y_i, where y_i = *TOP* of the floating point stack at the entry point of some IR function i

Helper Symbolic Variables :

$Y(F)$ = *StackDiff* of function F, where F is an internal function

TOP(I) = top of the stack after executing instruction I

TOP$_{\mathrm{b}}(\mathrm{I})$ = top of the stack before executing instruction I

Initial Conditions :

TOP$_{\mathrm{b}}(\mathrm{I}) = 0$ where I is the first instruction of the entry point function.

Data flow rules :

Same as before in figure 4.2.

Figure 4.4: Modified initial conditions for the data flow rules decoding the floating point stack

not do that for other functions that are not reachable directly (we call them Root functions in the original data flow rules in figure 4.2). We set the TOP value for other functions reachable indirectly to be new unknowns y_i's. The reason these y_i's are not set to zeros is that such functions reachable indirectly can be some unresolved branch targets and not actual functions. For branch targets, it is acceptable to have floating point variables pushed on the stack at their entry points.

We solve the linear system of equations as before getting values for all unknown X_is and y_is. For all unknown X_i's, we set their values to ones conservatively as before.

After solving all equations and setting all unknown X_is to ones, some unknowns (y_is) might remain. This is unlike when using the techniques presented in the previous section where at every point in the binary the top of the stack becomes known after solving all equations and setting all unknowns to ones. If stack height is not known, we use a run time global variable to represent the current stack height and use it to index a global array that simulates the physical floating point stack. The challenge here is to make correct transfers between using local variables when the stack height is known to using the global runtime variables when the stack height is not known.

To convert floating point stack accesses into variable accesses in the IR, we follow the rules stated in figure 4.5. The left hand side shows the original binary instruction, and the right hand side shows the instruction that has to be emitted into the IR. Below we discuss these rules.

For any instruction accessing a floating point register where the TOP value is known, we do a direct translation to IR local floating point variables since we know exactly which variable the instruction is accessing. For instructions accessing floating point registers

#	Binary instruction	IR code inserted
1	I = op (st(i)), TOP_b(I) = constant	I = op ($PST_{TOPb\ (I)-1-i}$)

2	I = op (st(i)), TOP_b(I) ≠ constant
	Assume C(I) = StackDiff of instruction I
		I = op (GST[TOP-1-i])
		TOP = TOP + C(I)

3	I = call F, F is known statically (direct call)
	U(F) = true, TOP(I) is known
	Note: If arguments are passed, they are copied to the local variables of the callees at their entries
		//Pass arguments only if TOP_b(I) is known
		Call F ($PST_{TOPb(I)-1-i}$, $PST_{TOPb(I)-i}$, …, PST_0)
		//Move from globals to locals
		for (i=0: TOP(I))
		    $PST_{TOPb(I)-I-1}$ = GST[TOP_b(I) – I – 1]

4	I = call F, F is known statically (direct call)
	U(F) = false, TOP(I) is known
	Note: If arguments are passed, they are copied to the local variables of the callees at their entries
		//Pass arguments only if TOP_b(I) is known
		Call F ($PST_{TOPb(I)-1-i}$, $PST_{TOPb(I)-i}$, …, PST_0,
		&$retArg_0$, &$retArg_1$, …, &$retArg_{TOP(I)-1}$)
		//Move from return pointers to locals
		for (i=0: TOP(I)-1)
		    $PST_{TOPb(I)-I-1}$ = $retArg_i$

5	I = call F, F is unknown statically (indirect call)	TOP = zero Call call_translator //Read the return from global PST_0 = GST[0] //Remove if X_i is zero here

6	I = jmp L, L is not known and unresolved
	TOP_b(I) = constant
		for (i=0: TOP_b(I)-1)
		    GST[TOP_b(I) – I – 1] = $PST_{TOPb(I)-I-1}$
		TOP = TOP_b(I)
		Call call_translator
		Return

7	Before entry point instruction I of a function F only reachable indirectly, y_I is known	for (i=0: y_I-1) $PST_{TOPb(I)-I-1}$ = GST[TOP_b(I) – I – 1]

8	Before entry point instruction I of a function F reachable directly, y_I is known	F (arg_0, arg_1, …, arg_{yI-1}) {
		for (i=0: y_I-1)
		    $PST_{TOPb(I)-I-1}$ = arg_i
		…
		}

9	I = return, TOP_b (I) is known
	returned to function is unknown OR
	there exists F where F is a returned to function, U(F) = true
		for (i=0: TOP_b(I)-1)
		    GST[TOP_b(I) – I – 1] = $PST_{TOPb(I)-I-1}$
		    $retArg_i$ = $PST_{TOPb(I)-I-1}$ //when applicable
		Return

10	I = return, TOP_b (I) is known
	Any other case than above
		for (i=0: TOP(I)-1)
		    $retArg_i$ = $PST_{TOPb(I)-I-1}$
		return

U(F) = true if F or one if its direct callees has an indirect branch that is unresolved.

Figure 4.5: Translation to IR rules when some indirect jumps are unresolved

where the TOP value is not known, we use the global floating point array as well as the global variable representing the TOP value.

For direct calls, if the callee has unresolved indirect jumps then it will be using the global variables to represent the floating point stack. Upon return from the callee, we make sure that the global variables are updated correctly. After the return, if the TOP value is known, we copy the contents of the global variables into the local variables such that instructions start using the local variables correctly. If the TOP value is known before the call, floating point local variables are passed as arguments to the callee.

For indirect calls, we have an assumption that before such calls the stack top value should be zero. We copy this to the global TOP value in case the callee is using the globals to represent the floating point stack operations. Before returning from all functions that can be called indirectly, we make sure to update the global variables contents (if locals were used).

Before indirect jumps that are unresolved, we copy the stack values from local variables to the global variables (in case the local variables were used before the jump). The general translator function is used in the indirect branch to redirect control to the correct IR function. The global TOP variable is updated also.

For all of this to work, if some functions that can be called indirectly have a known TOP value on their entry point, the floating point variables are copied from the global variables to the local ones. For functions reachable directly, if they have floating point arguments, they are copied to local floating point variables in the IR at the function entry point.

4.6 Correctness Proofs

In this section, we prove that all the techniques presented in this section will produce a correct IR with respect to floating point stack operations. We show first a proof of the simple technique when all indirect jumps are resolved, then we proceed to proving the general technique when some indirect jumps are not resolved correctly.

Definition 4.1 an *fp-functional* rewritten binary is a rewritten binary that executes correctly with respect to floating point accesses which means that any floating point value that is being read/written into a floating point register in the original binary results in reading/writing the same value to some floating point variable in the rewritten binary when executed.

When we prove the correctness of our floating point stack recovery techniques presented before, we prove that our rewritten binaries compiled from the recovered IR are *fp-functional*. To prove this, we usually prove that we track the original binary's TOP of the floating point stack value correctly.

Lemma 4.1 Under the assumptions stated in section 4.3, if algorithm 6 returns false, then solving the equations resulting from the propagation rules in figure 4.2 and setting the remaining unknowns to ones will always ensure that the rewritten binary is *fp-functional*.

Proof Suppose the input binary has no direct or indirect calls. The TOP of the stack is tracked correctly for every program point since instructions are always known to the static analyzer and no uncertainty happens. Since the recovered TOP is exactly equal to the original TOP value in the input binary, the lemma holds in this case.

Now, if the binary have direct calls, the callee functions will always be known to our static techniques and hence no uncertainty occurs in recovering $TOP(I)$ for any instruction I. Hence the lemma also holds in this case.

The only uncertainty occurs when some indirect control transfers happen. Indirect control transfers are either indirect calls or indirect branches.

For indirect calls, uncertainty happens when some X_is are set to one conservatively. An X_i represents the top of the stack after the return from indirect call site i. Per our second and third assumptions in section 4.3, $max(X_i) = 1$. If the actual call site in the input binary does not return any floating point value and we set X_i to one conservatively, the lemma still holds since the input binary will never have any access to the additional non-existing return value that we created in the IR. The binary has to push some element to the stack before reading it. In this case, the recovered $TOP(I)$ for any instruction I will always be one plus the original value during both stack writes and reads which guarantees correct behavior.

Regarding indirect branches, Since the check in algorithm 6 fails, this means either all indirect branches are resolved, or no indirect branch target can access the floating point stack. In case all indirect branches are resolved, their targets are known to the static analyzer and hence the recovered $TOP(I)$ for any instruction I will be tracked correctly and hence the lemma holds. If some indirect branches are unresolved, and it is known that none of their targets can access the floating point stack, then no problem occurs in this case since no floating point register is accessed.

Lemma 4.2 For a direct call instruction I_F in the IR calling function F. Let the first IR instruction in F be I. If ***TOP***$_b(I)$ is known, then ***TOP***$_b(I_F)$ cannot be unknown and vice-versa.

Proof Since the propagation rules in figure 4.4 assign the same symbolic expression to both *TOP*$_b(I_F)$ and *TOP*$_b(I)$, then if one of them is known, the other is automatically known and vice versa.

Lemma 4.3 For a return instruction I_R in the IR returning to function F that can be determined statically. Let I be any call site to F. If *TOP*(I) is known, then *TOP*$_b(I_R)$ cannot be unknown and vice versa.

Proof Can be proved in a similar way to lemma (4.2).

Lemma 4.4 Under the assumptions stated in section 4.3, if algorithm 6 returns true, then the IR recovery rules stated in figure 4.5 will always ensure that any floating point value that is being read/written into a floating point register in the original binary results in reading/writing the same value to some floating point variable in the rewritten binary when executed.

Proof From the first two translation rules in figure 4.5, at any instruction I that can access the floating point stack, the IR uses either the local variables *PST* when the *TOP*$_b(I)$ value is known, or the global array *GST*[] if the *TOP*$_b(I)$ value is unknown. If we guarantee correct flow of floating point values between local variables (*PSTs*) and global variables (*GSTs*) in the IR, we can prove this lemma.

To make it easier to understand the proof, we introduce the following claims. If all these claims are true, the proof can be constructed in a simple way. We assume they are true, prove the lemma, then state the proof of each claim individually.

Claim A: For a trace of IR instructions containing no calls or returns, let I be the first instruction in the trace, if *TOP*$_b(I)$ is statically known and local variables *PST*$_x$ contain all live floating point values at I that are stored in order starting from $x = 0$ until $x = TOP_b(I) - 1$, then this trace is *fp-functional*.

Claim B: For a trace of IR instructions containing no calls or returns, let I be the first instruction in the trace, if *TOP*$_b(I)$ is statically unknown, then this trace is *fp-functional* given the following two conditions: 1) The global variable *TOP* contains the value *TOP*$_b(I)$ before executing I. 2) The global array *GST[x]* contains all live floating point values at I that are stored in order starting from $x = 0$ until $x = TOP_b(I) - 1$.

Claim C: $\forall F \in FuncsK$ where *FuncsK* represents the set of IR functions with a statically known entry point *TOP*$_b(I)$ where I is the first instruction in F, let I_F be any call site of F, the local variable *PST*$_x$ at F entry point contains one of the following values for all $0 \leq x \leq TOP_b(I) - 1$:

$$PST_x(\text{before } I) = \begin{cases} PST_x(\text{before } I_F), & TOP_b(I_F) \text{ is known} \\ GST[x](\text{before } I_F), & TOP_b(I_F) \text{ is unknown} \end{cases} \tag{4.1}$$

Claim D: $\forall F \in \text{FuncsU}$ where *FuncsU* represents the set of IR functions with a statically unknown entry point *TOP*$_b(I)$ where I is the first instruction in F, let I_F be any call site of F, all the following is true before executing I: 1) Global variable *TOP* contains the value *TOP*$_b(I_F)$. 2) Global variable *GST*$[x]$ contains one of the following values for all $0 \leq x \leq TOP_b(I) - 1$:

$$GST_x(\text{before } I) = \begin{cases} PST_x(\text{before } I_F), & TOP_b(I_F) \text{ is known} \\ GST[x](\text{before } I_F), & TOP_b(I_F) \text{ is unknown} \end{cases} \tag{4.2}$$

Claim E: $\forall I \in \mathrm{CallSitesK}$ where *CallSitesK* represents the set of all IR call sites with a statically known *TOP*(I), let F be any actual function that can be called from I (possibly through a call translator) and let I_R represent any return instruction inside F, the local variable *PST*$_x$ after I has one of the following two values:

$$PST_x(\text{after } I) = \begin{cases} PST_x(\text{before } I_R), & TOP_b(I_R) \text{ is known} \\ GST[x](\text{before } I_R), & TOP_b(I_R) \text{ is unknown} \end{cases} \tag{4.3}$$

Claim F: $\forall I \in \mathrm{CallSitesU}$ where *CallSitesU* represents the set of all IR call sites with a statically unknown *TOP*(I), let F be any actual function that can be called from I (possibly through a call translator) and let I_R represent any return instruction inside F, all the following is true after executing I_R and returning to F: 1) Global variable *TOP* contains the value *TOP*$_b(I_R)$. 2) Global variable *GST*$[x]$ contains one of the following values for all $0 \leq x \leq TOP_b(I_R) - 1$:

$$GST_x(\text{after } I) = \begin{cases} PST_x(\text{before } I_R), & TOP_b(I_R) \text{ is known} \\ GST[x](\text{before } I_R), & TOP_b(I_R) \text{ is unknown} \end{cases} \tag{4.4}$$

Claims A and B state correct and functional execution traces with no calls or returns. Claims C and D ensure correct floating point values flow from call instructions to the functions being called. Claims E and F ensure correct floating point values flow from return instructions to the call sites.

Assuming claims A through F are correct. We can use mathematical induction to prove lemma (4.4) as discussed below.

Assuming the dynamic execution of the rewritten binary is divided into n instruction traces separated by calls/returns and ending with the program termination. n can be arbitrary large and cannot be computed statically, but we do not seek calculating n, rather we will use mathematical induction on n to prove the lemma.

Base case: $n = 1$ Since the entry point has a zero top of the stack (according to our assumptions), there is no live floating point values at the trace entry and hence lemma (4.4) is true as a direct result of applying claim (A).

Inductive case: Assuming trace n is *fp-functional*, we want to prove that trace $n+1$ is also *fp-functional*.

There are two cases: either a call instruction separates the two traces, or a return instruction separates the two traces. We discuss every case individually. For both cases, we assume that the live floating point values are stored correctly in trace n since it is *fp-functional*.

1) If a call instruction I_F separates the two traces, let the callee be F with instruction I the first instruction in function F (the first instruction in trace $n+1$). We have four cases depending on if we statically know the values $TOP_b(I_F)$ and $TOP_b(I)$. Each of the cases is proved in table 4.1.

2) If a return instruction I_R separates the two traces, let the returned to function be F with instruction I being the call instruction that was used to reach F. We have four cases depending on if we statically know the values $TOP_b(I_R)$ and $TOP(I)$. Each of the cases can be proved in a very similar way to the previous case in table 4.1 but using Claims (E) and (F) instead of claims (C) and (D).

Below we discuss the proofs of every claim from the above.

$TOP_b(I_F)$	$TOP_b(I)$	Proof
X	X	Both traces n and $n+1$ are using the global array *GST[]* as per rule (2) in figure 4.5. Claim (D) guarantees that the global array *GST[]* will not change in this case across the call. Applying Claim (B) to trace $n+1$ proves lemma (4.4) in this case.
X	$\surd$	Trace n uses the global array *GST*$[x]$ but trace $n+1$ will be using local variables *PST*$[x]$ as per rules (1) and (2) in figure 4.5. Claim (C) guarantees that the global array *GST*$[x]$ will be copied over to local variables *PST*$[x]$ in F. Applying Claim (A) to trace $n+1$ proves lemma (4.4) in this case.

$\surd$	X	Trace n uses local variables *PST*$[x]$ but trace $n+1$ is using the global array *GST*$[x]$ as per rules (1) and (2) in figure 4.5. Claim (D) guarantees that local variables *PST*$[x]$ will be copied over to the global array *GST*$[x]$ before calling F. Applying Claim (B) to trace $n+1$ proves lemma (4.4) in this case.
$\surd$	$\surd$	Both traces n and $n+1$ are using the local variables *PST*$_x$ as per rule (1) in figure 4.5. The local variables in this case are in different functions. Claim (C) guarantees that the local variables *PST*$_x$ in the caller are copied to the local variables *PST*$_x$ in the callee (F). Applying Claim (A) to trace $n+1$ proves lemma (4.4) in this case.

Table 4.1: Proofs of Lemma (4.4) for trace $n+1$ if reachable using a call instruction

Claim A Proof Since the trace of instruction with no call/returns represents a binary trace that is completely known to the static analyzer, the recovered *TOP*(I') value at any instruction I' in this trace is statically determinable with respect to the trace entry point *TOP*$_b(I)$ where I is the trace entry instruction. Given that all floating point variables are stored correctly to local variables *PST*$_x$ before the trace entry, any instruction that reads these values will get them from the same *PST*$_x$ variables as per rule (1) in figure 4.5.

Claim B Proof Can be constructed using the same argument in the proof of claim (A) but referring to the global variables *GST*$[x]$ instead of local variables and rule (2) instead of rule (1) in figure 4.5.

Claim C Proof Instruction I in F is either reachable using a direct call, indirect call, or an unresolved indirect jump. We discuss every case below: 1) Direct calls: according to lemma (4.2), *TOP*$_b(I_F)$ is known in this case and hence rules (3) and (4) in figure 4.5 govern the direct calls in this case. In both rules, the local variables are passed directly as arguments to F and inside F they are then copied to local variables as per rule (8) in figure 4.5. The stack height at the call site cannot be known in this case according to lemma (4.2).

2) Indirect calls will always have zero TOP of the stack before the call site as per our second assumption in section 4.3 and hence no transfer of variables is required in this case. Rule (5) in figure 4.5 sets the global variable *TOP* to zero at the call site.

3) At unresolved indirect jumps, if *TOP*$_b(I_F)$ is known, local variables are copied to the corresponding globals in rule (6). If *TOP*$_b(I_F)$ is not known, globals are already up to date according to rule (2). Inside F, globals are copied back to locals inside F as per rule (7).

Claim D Proof Can be proved using similar argument to claim (C) proof above. Will only write the relevant rules for each case below: 1) Direct calls: they have to have an unknown stack height as per lemma (4.2) and hence global variables remain to be used.

2) Indirect calls: have to have a zero stack height on their front and hence no transfer is required.

3) Unresolved indirect jumps: globals are updated according to rule (6). The same argument in the claim C proof applies here.

Claim E Proof Here the stack height after call site I is known, hence local variables have to be updated after I. I can only be reachable through a return instruction. We have two cases:

1) If *TOP*$_b(I_R)$ is known: either rule (9) or rule (10) applies in this case as follows:

- In case the return cannot be statically proven to return to I, or I's parent function has one or more unresolved jumps, then the globals are updated in rule (9). The return will come back to a call translator function in the IR which returns back to either an indirect call site or to an unresolved indirect jump site. For indirect calls, rule (5) will copy *PST*$_0$ from the global array. For unresolved indirect jumps, they return back to some call site as per rule (6). This call site will have $U(F) = true$ and rule (3) will copy variables from the globals back to locals.

- If the return can be proven to return to some IR function whose indirect jumps are all resolved, then rule (10) applies and return arguments are propagated. Rule (4) propagates the return arguments back to local variables after I.

2) If *TOP*$_b(I_R)$ is unknown, no transfers are required. The globals will be ready at

I_R as per rule (2). According to lemma (4.3), I_R cannot be statically proven to return to I since in this case I would have an unknown *TOP*(I) which is not the case. For non-statically resolvable returns, they are handled the same way as in the previous case (1) above in this same proof.

Claim F Proof Can be proved using similar arguments to claim (E) above. We show a summary of which rules apply here.

1) If *TOP*$_b(I_R)$ is known: As per lemma (4.3), I_R will not be statically proven to return to I since in this case *TOP*(I) would have been known which is not the case. Return site updates globals in rule (9). Rule (2) will use these globals at the call sites.

2) If *TOP*$_b(I_R)$ is not known: globals are already used both at the return sites and also at the call sites per rule (2). No transfer is needed in this case.

4.7 Results

In this section, we show the effectiveness of our techniques in identifying floating point stack variables.

In all of our experiments, the check described in algorithm 6 returns false which means that we did not need to do any global adjustments to the floating point stack variables in any of our tests. The reason behind this is that our jump table heuristics are very accurate and resolve most of the indirect jumps. SecondWrite implements a modified version of the heuristics described in [14]. Almost 100% of the indirect branches are resolved (by knowing all their targets) statically. For the unresolved ones, the indirectly called functions in these cases were never detected to access the floating point stack.

We show the percentage of the symbolic values that were not solved using our linear solver and required the conservative assumption of $X_i = 1$. As mentioned in chapter 4, the main challenge while decoding the floating point stack is to identify whether an indirect or an external call is modifying the floating point stack height. According to our assumptions, whenever we are not sure about an indirect or an external call site, we decide conservatively that it is modifying the floating point stack by pushing a single value. We show how often we took that conservative decision in different binaries.

All register allocated floating point stack variables were recovered correctly and all the rewritten benchmarks ran correctly and produced correct answers. The conservative decision taken does not affect correctness as we explained in chapter 4. It only adds extra return values to some indirect and external calls and this might reflect adding more return values to internal functions as well.

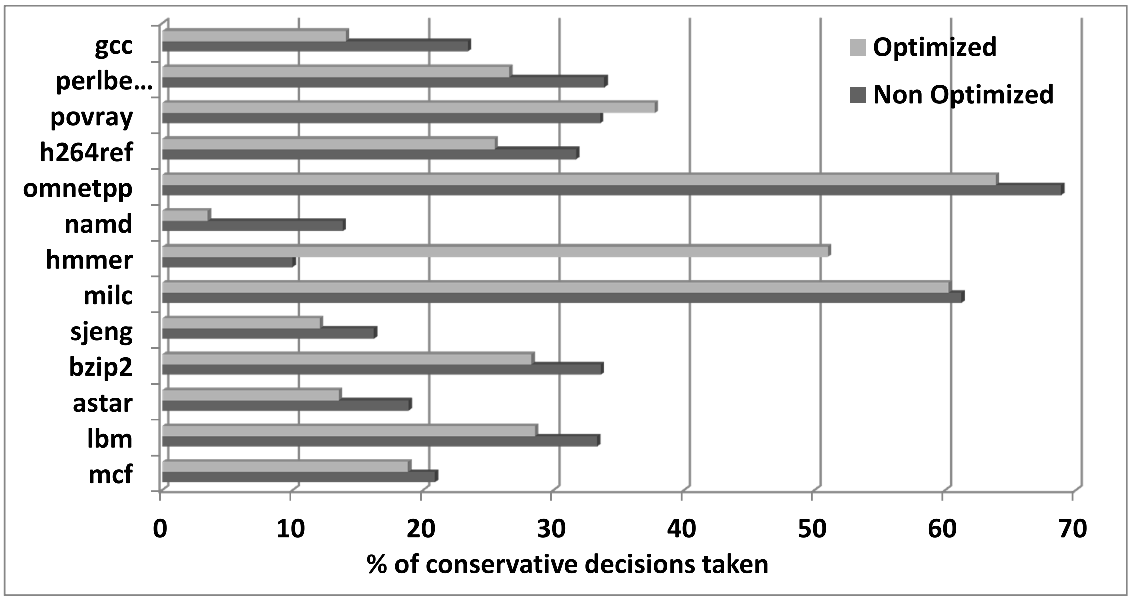


Figure 4.6: Conservative floating point decisions for Windows

Figure 4.6 shows the percentage of the unknown calls for which we took the conser-

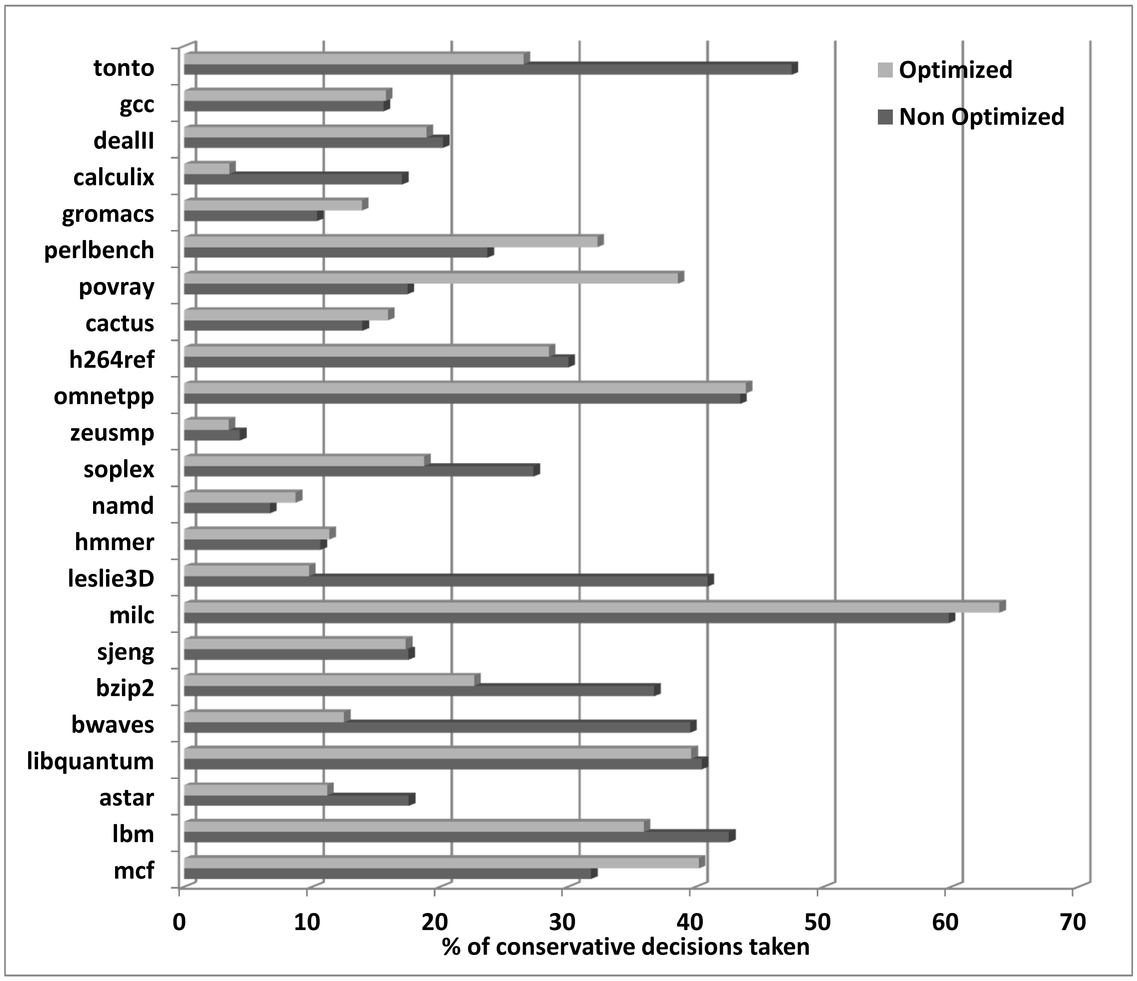


Figure 4.7: Conservative floating point decisions for Linux

vative decision in the subset of the benchmarks that we were able to compile on Microsoft Visual Studio 2010. Figure 4.7 shows the results on Linux binaries. On average, we took the conservative decision 28% of the time for non-optimized executables and 25% of the time for optimized ones. This means we are able to identify the exact floating point arguments and returns for more than 72% of the indirect and external calls on average. We are not aware of any work that identifies such information. Optimized binaries often have less variables than non-optimized binaries which translates to less floating point stack usage and less number of times when the conservative decision is taken. This is true in most of the cases, but in some cases the optimized binaries are challenging because they have fewer control flow edges. One example of this behavior is 'hmmer' where the optimized binary has much less control flow join points and hence much less number of equations and higher conservative decision ratio. The conservative decision is usually taken more often in C++ binaries because they have more indirect calls than C and Fortran binaries.

4.8 Related Work

We are not aware of any work done to recover floating point stack variables except Hex-Rays [31]. Hex-Rays produces inline assembly in case it cannot resolve the variables which is not acceptable for our goal. There is no published work on the details of their techniques as well as how often it fails to identify variables from low level stack accesses.

None of the static and dynamic binary rewriting tools like PIN [48], BIRD [52], ATOM [28], PLTO [60], Boomerang [26], Jakstab [36], UQBT [16], Bitblaze (BAP) [9] and CodeSurfer/X86 [50] decodes the x86 floating point stack into variables. None of those tools employ a compiler level intermediate format, like LLVM IR or similar; rather they define their own low-level custom intermediate format.

Chapter 5: Recovering Memory Allocated Variables and Data Types

5.1 Introduction

In this chapter, we present static analyses that can recover source level variable and type information from x86 binaries as large as millions of instructions in a few minutes. The produced information is as accurate as the current state of the art x86 binary analysis systems with much faster and scalable analysis. The recovered information is represented in a high level compiler IR that is completely functional and produces a correct rewritten executable when recompiled. Our static techniques combine functionality, precision and scalability; features that collectively do not exist in today's binary analysis tools.

This chapter presents an important step towards a system that rewrites executables into a functional high-level program representation and incorporates as much source level information as possible in a scalable manner. This chapter has the following contributions:

- It presents a highly scalable mechanism for identifying variables and types which is orders of magnitude faster than current analysis techniques. Our techniques do not rely on symbol or debug information to be present in binaries.

- It presents practical techniques to emit the recovered data types from binaries into a high level IR. The emitted types include scalars, pointers, arrays, structures and recursive data structures.

- It is evaluated and shown to recover accurate and precise information from C, C++, and Fortran binaries obtained from the SPEC2006 benchmarks suite; compiled using two different compilers in a reasonable amount of time.

This chapter is divided into eight sections. Section 5.2 gives an overview about the variables and data types identification problem. Section 5.3 represents our variables recovery technique. Section 5.4 presents our data type recovery techniques for the variables discovered in the binary. Section 5.5 shows how we emit data types into the recovered IR including pointer and recursive data types. Section 5.6 discusses the correctness of the IR recovered and the algorithms termination guarantees. Section 5.7 shows how our variable recovery techniques can still work for inaccurate function boundaries. Section 5.8 shows a detailed evaluation of the techniques presented and section 5.9 presents a literature review about the variables and data types recovery from binaries.

5.2 Variable and Type Recovery - Challenges and Intuitions

Variable and type recovery from executables is a hard problem because symbol tables are absent. Every memory-allocated variable access in the source code is represented by a memory store or load in the executable. Those memory accesses are either direct accesses to locations represented by constant addresses, or indirect memory accesses to locations represented by some register value.

Direct memory accesses can be used to infer variable information by examining the constant memory address being accessed, but indirect memory accesses are unknown accesses and need more advanced memory analysis to reveal the underlying memory locations. That is why pointer analysis is important while recovering variables and data types from executables since it reveals what are the possible memory locations an indirect memory reference can possibly access.

Researchers in this field know this and the best known variable identification technique from executables (DIVINE [4]) uses an advanced memory analysis technique called value set analysis [5], which is a generalized form of alias analysis for binaries built on top of the aggregate structures identification algorithm [54]. DIVINE presents accurate variable identification that detects 88% of the memory-allocated variables in executables. The problem with DIVINE is that it is not scalable and requires a very long time to analyze even small programs. Our aim is to present techniques with the same accuracy as DIVINE, but run orders of magnitudes faster.

Scalable source-level pointer analysis techniques (like Steensgaard's analysis [65]) cannot be used on executables since executables lack variables and data types information. A custom pointer analysis technique has to be implemented for this purpose.

Our key insight that enables scalability is that efficient variable detection and type recovery do not require a sound pointer analysis. Unsound pointer analysis usually means incomplete points-to sets. As an example, if variable `x` points to `y` and `z`, an unsound pointer analysis might report `x` points to `y` only. Usually unsound pointer analysis is unacceptable, but variable detection from executables is a best-effort analysis and no method claims to detect 100% of the variables. If we are going to miss some variables anyways because of the nature of the problem we are solving, then we can sacrifice the soundness of the analysis at the expense of losing some variable information – as losing variable `z` in the given example above, but with the gains of having a practical analysis that scales well for large executables.

The correctness of the recovered IR, while missing some variables due to the unsound pointer analysis, comes from the fact that the relative ordering between variables in the memory layout is maintained in the recovered IR. For example, if we detect two integer local variables at offsets 0 and 20 on a stack frame of size 24 bytes, we will lay out those variables in a structure which has the following three members: a) An integer in the range [0-3]. b) A generic array of bytes in the range [4-19]. c) An integer in the range [20-23]. Preserving the layout of the variables in such a structure maintains the correctness of any indirect memory access to this region. The arrays inserted fill the unknown gaps between variables and maintain the memory layout. This representation helps understanding what variables are detected along with their types, and at the same time maintains the functionality of the rewritten program.

We introduce the concept of a best-effort pointer analysis; where the identified points-to set of each pointer may not be complete, but we terminate the analysis in a certain amount of time nevertheless to prevent it from taking too long even before it converges. This analysis is not correct given the usual criteria for correctness, but suffices in the way we use it to identify as many discrete variables as possible. Our best-effort pointer analysis is a flow and context insensitive data flow analysis that has the following properties:

- It limits the cardinality of the points-to sets to a fixed number.
- It does not track interprocedural information via indirect calls.

- The number of analysis iterations is set to a fixed number.

Having the above relaxations makes our analysis much faster at an extremely small loss in precision. The intuition behind this is as follows: a) A flow and context sensitive pointer analysis is not needed since the variables usually have the same size and type in all flows and contexts of a program. Some exceptions to this might happen which is not common in the programs. b) Limiting the cardinality of points-to sets does not affect the precision that much since only few variables will have large points-to sets. c) Propagating interprocedural information through indirect calls will only affect functions which are only called indirectly. Those functions are still analyzed, but their arguments will have unknown points-to sets. Given that there are relatively few such functions in executables, skipping their arguments propagation is not a big loss. d) Limiting the total number of iterations will only affect longer chains of pointers. For example, the first iteration will always reveal some pointers. The second will reveal two-level (double) pointers. Subsequent iterations reveal more pointer levels. Usually most variables do not have more than four level pointers, which means subsequent iterations will only reveal very little information.

5.3 Best Effort Static Variable Recovery

We show in this section how a simple best-effort pointer analysis can be used for identifying variables. This pointer analysis should be suitable to run on executables where no variables are identified yet. We could have modified current memory analysis schemes on executables like [5] to fit our needs, but we show a simpler analysis with similar precision and much better scalability.

Before we begin the analysis, we identify all base memory regions in the executable. An executable has the following three base memory regions.

1. The global memory region where global variables are located.

2. The stack memory region where local variables inside functions are located. Stack regions are allocated at the beginning of a function and deallocated at the end of the function.

3. The heap memory region where dynamically allocated variables are usually located. Those are identified by detecting calls to functions like `malloc` and `new` in the executable.

Every detected memory-allocated variable is represented by an abstraction called *ALoc* which stands for Abstract Location. The name is similar to the name used by DIVINE [4]. An *ALoc* contains an offset inside a base memory region and a size representing the variable size. Variables allocated to registers are represented by IR symbols which represent the SSA form of those registers.

Our pointer analysis conservatively assumes that every detected variable can be a pointer. We assign points-to sets to every IR symbol and detected ALoc. When the analysis is done, the actual pointers are identified by tracking if the corresponding points-to sets are not empty.

We implement the points-to sets using the efficient LLVM sparse bit vector data structure. For every base memory region, we assign it a series of unique bits where the number of bits equals the size of the region in bytes. If the size of the base memory region is not known (usually in heap allocated arrays), we assume an arbitrary size. This allows us to detect variables with offsets up to that size. Whenever an access is detected beyond that arbitrary size, we do not track it. This is an important part of our best-effort analysis that allows us to recover a subset of the variables on un-sized base memory regions instead of totally giving up on them as the case in DIVINE [4]. Whenever a symbol or an ALoc points to some variable in a certain memory region, the bit corresponding to the starting address of the variable will be set to one. The number of bits set to one equals the number of variables pointed to by a symbol or an ALoc.

Table 5.1 shows our detailed propagation rules for the best-effort pointer analysis as well as for detecting the variables. We introduce the following definitions to ease the understanding:

1. PtSet(x): takes an ALoc or an IR symbol x and retrieves its points-to set 'bit-vector'.

2. ALocs(x): takes a bit-vector x and retrieves the set of ALocs starting at the addresses that correspond to the set-bits in the bit vector x.

3. UpdateALocs(x,y): takes a bit-vector x and a size y and creates ALocs starting at the addresses corresponding to the set-bits in the bit-vector x with the given size y. If existing ALocs overlap the new ALocs, the new and old ALocs will be split into smaller ALocs to avoid the overlap.

4. UpdateStructure(x,y): takes a bit-vector x and a number y. It defines a set of structures starting at the addresses corresponding to the set-bits in the bit-vector x. Each

store y, x (store value y to location x of size S)	$\forall\ z \in$ ALocs(PtSet(x)) : PtSet(z) $\cup$ = PtSet(y) **Variables:** UpdateALocs (PtSet(x), S)
y = load x (load location x of size S to y)	$\forall$ z $\in$ ALocs(PtSet(x)) : PtSet(y) $\cup$ = PtSet(z) **Variables:** UpdateScalar (PtSet(x), S)
y = x	PtSet(y) = PtSet(x)

y = x + z , PtSet(x) is not empty	**if** z is a constant **then**
	  PtSet(y) = PtSet(x) $>>$ z
	Variables:
	if z is a constant **then**
	  UpdateStructure (PtSet(x), z)
	else if z has SCEV bounds and stride **then**
	  UpdateArray (PtSet(y), stride, bounds)

Table 5.1: Points-to sets propagation and variable detection rules

structure has its last member at offset y. If a structure already starts at one of the starting addresses, its last member offset will be updated with the maximum of the existing offset and the new one (y).

5. UpdateArray(x,y,z): takes a bit-vector x, a number y representing a stride, and another number z representing the upper bound of the array. It defines arrays starting at the addresses corresponding to the set-bits in the bit-vector x. Each array has a maximum size z. The arrays will be declared to have an element size y. Existing arrays will be merged with the new declared ones and the element size will be set to one if overlapping arrays have conflicting element sizes.

Here we describe briefly the propagation rules in table 5.1. For a store instruction, the points-to sets of the ALocs pointed to by the pointer operand will be unioned with the points-to set of the value stored. This is called a weak update in the domain of pointer analysis. A load will set the loaded value points-to set to whatever is pointed to by the pointer operand. Stores and loads will create ALocs as they are resolved using the UpdateALocs function described earlier. For pointer arithmetic, the points-to sets will be shifted right according to the positive constant added. If the constant is negative, the shift will become to the left. Adding a constant to a pointer is a hint about the existence of a structure where the pointer address is the start address, and the constant represents one field offset inside the structure. We use this hint and declare a structure identified by the starting address and the last member offset. The structure's last member offset might be updated in subsequent pointer arithmetic operations that start from the same base. The structure's last member offset will eventually be the maximum observed constant that was added to the pointer in the program. Adding a non-constant value is an indication that an array exists. An array will be declared in this case. We use the Scalar EVolution (SCEV) analysis by LLVM to deduce the bounds and the stride of the arithmetic and use this information to describe the array. If such information is not present, we do not declare an array.

The more pointer analysis rounds done, the more ALocs, structures and arrays are identified in all base memory regions. More pointer analysis rounds help identifying multi-level pointers since the first round will always reveal single level pointers. The second round will propagate the points-to sets for those ALocs and identify their points-to sets leading to the identification of two level pointers. More rounds will reveal more levels.

After all iterations are done, collected information about arrays gets resolved. For every base memory region, we fill in the gaps between ALocs using arrays. The bounds and stride information are available from our earlier propagation. If no bounds are available, previously defined ALocs are used as bounds. If no stride information is available, a stride of one is used which means the array is an array of bytes. Overlapping arrays are combined into one bigger array as described earlier.

At the end of this process, a structure hierarchy is created based on the structure information calculated for every base memory region. Using the starting and ending offsets previously calculated for every structure, we construct nested hierarchy structures. We define inner and outer structures such that any outer structure must have its starting address less than any starting address of any nested inner structure, and its ending address larger than any ending address of any nested inner structure. This nested structure hierarchy is used to emit structure data types in the IR as we will explain later in this chapter.

5.4 Data Type Recovery

Data type recovery aims at representing every symbol in the IR with a meaningful type. It declares a map between every symbol in the IR and the corresponding detected data type. It uses this map to rewrite the complete IR such that the instructions use the detected types instead of the generic types that are used by SecondWrite.

Without integrating type recovery with some pointer analysis, detected types will be less accurate because of two reasons: 1) Instructions like memory loads and stores will usually be untyped since there is no memory tracking possible. 2) Multi-level pointer types will not be detected because there is no way to track them without having some sort of pointer analysis.

To achieve the goal of typing memory accesses and IR symbols; and detecting multi-level pointer types, we integrate our best-effort pointer analysis and variable recovery techniques described above with our type recovery system. Any other pointer analysis like [5] can be theoretically used, but will be orders of magnitude slower which makes it less practical in large executables. That is the disadvantage of TIE [43] which is the state of the art binary type recovery technique.

Integrating our variable identification system with type recovery makes the type recovery simpler because it will need only recover scalar types like integers, floats and doubles. Structures and arrays are detected as part of the variable identification. A pointer is detected if the points-to set of the corresponding ALoc or IR symbol is not empty. In this case, we get the ALocs pointed to by that pointer and type them according to our rules. We keep doing this for longer pointer chains as needed.

A = call foo (arg_1, ..., arg_n)
 	
foo has the known prototype:
 	
$retType$ foo ($type_1$, ..., $type_n$)
	$\forall$ x $\in$ [1,n]
	
	setType(arg_x, $type_x$)
	
	setType(A, $retType$)

A = B op C $op \in \{+, -, *, /, \%, >>, <<\}$ op has type: $opType$ A, B, C has empty points-to sets	setType($\{A, B, C\}$, $opType$)
A = load B store A, B	unifyType(A, ALocs(PtSet(B)))
op_1 = ϕ (op_2, ..., op_n) op_1 = typecast op_2 to $type$	unifyType($\{op_1$, ..., $op_n\}$)

Table 5.2: Typing rules

Table 5.2 shows the most important typing rules we have. There are two main type sources. a) Known external function calls like standard C/C++ library calls. For those, we set the types of actual arguments passed to be the same as the known argument types from the prototypes and we do the same thing for the return value. b) Arithmetic operations with non-pointers: in this case the type is deduced from the semantics of the operation itself – whether it is an integer or a floating point operation –. We use the function *setType* to update the type of the symbol or the ALoc in the type map we declare. For pointer types, we type the ALocs represented by the points-to sets of the corresponding variables.

For the other operations in the table, we propagate the types using the function *unifyType*. This function attempts to set the data type of all the given symbols and ALocs to be the same. At least one of the symbols or the ALocs given to that function should be typed. Whenever this function finds conflicting types, it gives up and does not update any types. It is used for copy operations like type casts and phi nodes. It is also used to propagate types through memory as shown in the rules for stores and loads. Interprocedural information is propagated by unifying the formal and actual arguments types at a call instruction. The return value data type at the call site is unified with all the data types of all return values appearing in the return statements inside the called function body.

5.5 IR Data Types Emission Algorithm

After recovering variables and data types information from the techniques presented in the previous two sections, we proceed to express this information in the IR such that end users can use this information right away.

The data type emission process we present in this section aims at producing an IR which has the following three properties:

1. The IR is readable with as many recovered data types expressed in the IR as possible.

2. The IR is correct which means that users can take the IR, recompile it and still produce a correct rewritten binary that produces the same answers as the input

binary.

3. The IR contains recursive data types if applicable.

Achieving all three goals together is challenging. The readability goal usually conflicts with the correctness goal since our recovered types might not be 100% accurate. Recursive data structures are hard to emit in the IR. For example, if an original binary has a linked list, the recovered information will be a structure with two fields, the first field is the element and the second field is a pointer to the same structure. Before emitting the complete structure, the type emission will need to refer to the same structure that is being emitted which might result in an infinite loop if the type emission algorithm does not specifically account for this case. In fact, this is one reason why the state of the art type recovery system called TIE [43] does not recover such recursive data types.

It is important to mention that we do not recover any shapes of the data types. We cannot distinguish a linked list from a tree or a graph. We only recover structures and pointers that might refer to the same structures. Other shape analysis techniques can run on our recovered IR to detect this kind of information.

The pseudo code in figure 5.1 shows how we choose a data type to emit for a certain point to set *bitset*. For an IR symbol or an ALoc, we first obtain its points-to set and then execute the algorithm in figure 5.1. If an IR symbol or an ALoc does not have a points-to set, we either get the recovered scalar type or return a generic type if no scalar type is recovered. The initial execution of the function *getEmittedType* shown in figure 5.1 will have *ignoreStructs* set to false and *chainedBitSet* set to an empty bit vector.

Lines (1) and (2) of the code returns the cached type of the current bit vector if any.

```
Type* getEmittedType (bitset, ignoreStructs = false, chainedBitSet = empty) {
     (1)  if (!ignoreStructs && cachedType(bitset))
     (2)       return getCachedType(bitset)
     (3)  if (chainedBitSet ∩ bitset is not empty) //Circular pointer
     (4)       return getGenericPointerType ()
     (5)  else
     (6)       chainedBitSet = chainedBitSet ∪ bitset
     (7)  if (!ignoreStructs && isStartOfStruct(bitset)) //Structure
     (8)       returnType = getStructType(bitset, structHierarchy(bitset))
     (9)  else if (isStartOfArray(bitset)) //Array
     (10)      returnType = ArrayType (getScalarType(bitset))
     (11) else if (hasScalarType(bitset)) //scalar type
     (12)      returnType = getScalarType (bitset)
     (13) else if (P2Set(bitset) is not empty) //pointer type
     (14)      returnType = PointerType (getEmittedType (P2Set(bitset),
               ignoreStructs, chainedBitSet)
     (15) else
     (16)      return getGenericScalarType () //non-identified type
     (17) cahcedType (bitset) = returnType //store to cache
     (18) return returnType
}
```

Figure 5.1: The type emission algorithm

We use type caching for two reasons: 1) It speeds up the type emission process. 2) It is necessary for recovering recursive data structures as we show later in this section. The cache is updated at line (17) in the algorithm.

Lines (3) through (6) of the code are inserted to avoid infinite loops while emitting IR data types. Infinite loops come when a circular pointer is detected where some pointer type is detected to point to itself, or to point to some other chain of pointer types among which one of them points back to the first pointer type. We call this a circular pointer data type. We do not allow emitting circular pointer data types except for recursive data structures where pointers point to detected structures not to scalars as we show next. If a circular pointer is detected, we return back a generic pointer data type (we still know it is a pointer, but lose the information about what it points to).

The code in figure 5.1 then proceeds to emit different type categories accordingly. The code is showing type emission for memory locations, but a very similar technique is used to emit types for IR symbols representing use points. We show every emitted data type category below.

Scalar Data Types: Lines (11) and (12) emit a scalar data type (like int, float, double, char, ...). The data type is already recovered from the techniques in the previous section. We only emit a scalar data type if the bit vector is not detected to point to an array or a structure.

The function *getScalarType* returns a recovered scalar type if all the ALocs pointed to by the bit vector *bitset* have the same scalar type. If there is a conflict, a generic scalar type is returned.

Pointer Data Types: Lines (13) and (14) emit a pointer data type if the bit vector *bitset* is detected to have a non-empty points-to set. We run the function *getEmittedType* recursively for the points-to set and return a pointer to the returned data type.

Array Data Types: Lines (9) and (10) emit an array data type if the bit vector *bitset* corresponds to a single array ALoc, or if all set bits in the bit vector *bitset* refer to isomorphic array ALocs (those with the same size and stride). This information is stored for us during the best-effort pointer analysis discussed in the previous sections. For the sake of simplicity, we only show scalar arrays. Arrays of pointers and arrays of structures can be emitted by recursively applying the *getEmittedType* function on the array ALoc element.

Structure Data Types: Lines (7) and (8) emit a structure data type in the array. A bit vector refers to a structure if: 1) It has only a single set bit, and 2) There is a structure hierarchy defined starting at this set bit as per our discussion in the last paragraph of section 5.3. If both conditions are true, we get the structure data type as shown in the pseudo code in figure 5.2.

```
StructType* getStructType (bitset, H /*Structure Hierarchy*/) {
     (1)  if (cachedType(bitset)) return getCachedType(bitset)
     (2)  returnType = createOpaqueStruct ()
     (3)  cachedType(bitset) = returnType
     (4)  startOffset = offset = LowerBound(H); maxOffset = UpperBound(H)
     (5)  while (offset <= maxOffset) {
     (6)    if (innerStructExist (offset, H))
     (7)      currentType = getStructType (bitset
                                    , getInnerStructHierarchy (offset, H))
     (8)    else
     (9)      currentType = getEmittedType (bitset, (offset==startOffset))
     (10)   addFieldToStruct (currentType, returnType)
     (11)   offset = offset  + size (currentType)
     (12)   bitset = bitset >> size (currentType)
     (13) }
     (14) return returnType
}
```

Figure 5.2: The structure data type emission algorithm

To emit a structure data type, we first check if this structure data type has already been cached in line (1) of figure 5.2. If not cached, we create an opaque structure and cache it in lines (2) and (3).

An opaque structure is a structure with no body defined. Creating an opaque structure is very similar to using forward declarations in C and C++. Creating an opaque structure and caching it is one key point in supporting recursive data structures. The reason is that once a field of some structure is declared to point to the beginning of the same structure, the cached opaque version will be returned instead of redefining the same structure again and again.

The algorithm in figure 5.2 handles the case of emitting aggregate types containing other aggregate and non-aggregate types to any nesting depth. The aggregate hierarchy recovery process was discussed before at the end of section 5.3. Lines (5) through (12) of the algorithm iterate over the structure elements. If an inner structure exists in the hierarchy, it is declared by calling *getStructType* recursively. If not, we call the *getEmittedType* recursively to recover the non-structure data type.

The reason we add the *ignoreStructs* argument to the *getEmittedType* function is that once an opaque structure is created, its corresponding bit vector will be cached to that structure type. The first element of that structure will have the same bit vector (the first iteration of the loop starting at line (5) of figure 5.2 will have the same *bitset* that was cached to the opaque structure in line (3). If the *ignoreStructs* flag is not set in this case, the function *getEmittedType* will return the cached opaque structure, not the first element type of the structure.

It is clear from the discussion above that the two algorithms in figure 5.1 and 5.2 enable the emission of recursive data structures. The two key points enabling this is the caching and the opaque structure creation mechanisms. These two algorithms do not allow circular pointers to scalar elements on the other hand as discussed before.

5.5.1 Practical Considerations

The algorithms discussed in the previous section for type emission show the basic idea of type emission. Some practical details are not included in the algorithms to simplify the discussion. We discuss these practical details here.

Emitting data types in the recovered IR works fine as long as the emitted data type has the same size as the underlying ALoc recovered from the pointer analysis. There are two situations where the size of the emitted data type can be different from the ALoc size. The first situation is related to type recovery inaccuracies, and the second is related to data structure alignment. We discuss these below.

Sometimes, the type detection is not accurate and there is no recovered type for certain ALoc(s). In this case, the algorithm in figure 5.1 returns a generic type. If we use the same generic type for all unknown ALoc types, they might mismatch the ALoc(s) size(s) and hence create problems in the rewritten binary execution. We make sure we choose the correct generic type that exactly matches the size of the underlying ALoc(s). If the points-to set *bitset* refers to more than one ALoc, they have to have the same size or otherwise the algorithm emits a data type that matches the semantics of the corresponding instruction in the IR without considering the ALocs at all. The actual algorithm implemented in SecondWrite takes care of that with extra added checks.

Another reason why a size mismatch happens is related to structure alignment issues. If the recovered types inside of a structure in the IR do not match exactly the types in the original source code, the backend of the compiler used to generate the rewritten binary (LLVM in this case) might introduce extra alignments inside the structure that makes the actual size of the structure in the rewritten binary different from its size in the IR.

As an example of when this alignment issue might occur, consider the original source code structure in figure 5.3-a and the corresponding recovered IR structure in figure 5.3-b. The structure in figure 5.3-a is compiled into the structure in figure 5.3-c after adding one byte padding to ensure that the short is aligned on a two bytes boundaries.

```
      Original         IR Recovered
      Structure          Structure

struct {            struct {
     char x;             char x;
     short y;            char y;
     short z;            int z;
}
                    }
        (a)                 (b)
```

Memory Layout (Original Binary)	Memory Layout (Rewritten Binary)
x (1 byte)	x (1 byte)
Padding (1 byte)	Y (1 byte)
y (2 bytes)	Padding (2 byte)
Z (2 bytes)	y (4 bytes)

(c) (d)

Figure 5.3: Structure alignment problem example

The total size of the recovered IR structure in figure 5.3-b (when adding up the individual field sizes) exactly matches the input binary structure in figure 5.3-c. Because the type recovery is not 100% accurate, the type recovery data flow analysis combined the two shorts y and z into one single integer z. This can happen if the original binary initializes both y and z simultaneously using a single store instruction. Character y in the recovered IR represents the padding in the original binary structure.

The problem is when we try to compile the code in figure 5.3-b, the compiler adds 2 bytes padding between the second character and the third integer since integers have to be aligned on 4 bytes boundary. The resulting memory layout is shown in figure 5.3-d. This layout is different than the input binary layout shown in figure 5.3-c since only one byte padding is required to be added. This will result in a mismatch in the sizes between the original and the rewritten structures which causes wrong behavior in case pointer arithmetic is used.

To solve this problem, we detect if the compiled IR structure size can be different from what is emitted and in such cases we mark the IR structure as being `packed`. A packed attribute for a structure instructs the backend not to emit any extra alignments at all.

5.6 IR Correctness and Analysis Termination

In this section we discuss the correctness guarantees our variable and data type recovery as well as the type emission techniques provide for the rewritten IR. We also prove that our type emission techniques will never go into infinite loops (always terminate).

The correctness of the recovered IR comes from the fact that the if the memory layout in the rewritten binary exactly resembles the memory layout in the original binary, then every memory access either being a direct access or an indirect reference using pointer arithmetic will always land on the correct (abstract) memory region in the rewritten binary and hence the rewritten binary memory references execute correctly.

The reason a compiler can change some memory layout is if individual variables are provided in the code that is being compiled. Compilers cannot provide any guarantees about the order of the variables in the binary. It can allocate them in any random order.

The only case when a compiler has to respect some memory layout is when the code instructs the compiler to do so. One way the code can do that is by having arrays and/or structures in the IR. The array and structure fields have to remain in the same order in memory.

We show here that for every memory region in the original binary, the corresponding abstract memory region in the rewritten binary is always surrounded by either an array or a structure in the IR that keeps the same layout.

In any case when a stack location is accessed in the original binary, it is accessed by adding some constant (or non-constant) offset to the stack pointer to get to the desired stack location. Following the last rule in table 5.1, this automatically creates a structure or an array for this particular memory region. The same argument applies for global and heap regions.

Regarding the type emission termination, we prove below that the algorithm in figure 5.1 always terminates. In this proof, we use the fact that the type detection techniques along with the best effort pointer analysis techniques always terminate (either after convergence, or after a certain number of iterations).

The only cases when the algorithm can go into an infinite loop is when it is calling itself recursively. This can only happen in line (8) and line (14).

At line (8) of the algorithm, the code is recovering a structure data type. The function *getStructType* might call itself (in case an inner structure exists), or calls the *getEmittedType* for other non-structure fields. Since the pointer analysis terminates, the structure hierarchy is finite, and hence when the function *getStructType* calls itself, it will keep calling itself until the hierarchy is done (which means a finite number of times) bounded by the number of inner structures. When the function *getStructType* calls *getEmittedType*, it tries to get a non-structure data type which is discussed below.

Line (14) of function *getEmittedType* calls itself for getting the type a certain pointer points to. Since we prevent circular pointers using the checks in lines (3) and (4), and since the original pointer analysis terminates, then the pointers chain has a finite length which leads to finite number of recursive calls at line (14) bounded by the chained pointers length.

5.7 Effects of inaccurate function boundaries and spurious functions

This chapter presented scalable techniques to recover variables and data types from binaries. These techniques use pointer analysis on memory locations in the binary to reason about memory allocated variables and their data types.

These memory allocated variable and data type recovery techniques will still work correctly for spurious functions and functions with inaccurate boundaries provided that arguments are passed correctly to all functions including spurious ones. For register arguments, they will flow correctly using the techniques presented in chapter 3. For memory arguments, they will flow correctly if the stack translation rules in section 3.4.1 are used to construct the IR before running the pointer analysis pass presented in this chapter. The same discussion applies to data types since they also use the same pointer analysis.

5.8 Results

In this section, we present the results showing the effectiveness of our schemes to identify variables and data types. We first show results on the overall variable and data type detection process and then we show specific in-depth results for floating point variables and function prototypes. We evaluate our techniques on the SPEC2006 benchmark suite which represents C, C++ and Fortran executables using different optimization levels and compiled using two different compilers (GCC 4.3 for Linux, and Visual Studio 2010 for Windows). We use a machine with an Intel Core i7 3.33GHz processor with 24 GB of

RAM.

All the recovered code in all the experiments was recompiled using LLVM 3.0, linked using GCC (Linux) and MinGW (Windows), and then tested on the ref and test inputs provided by the SPEC2006 test suite. All rewritten executables worked successfully and produced the correct answer as provided in the test suite.

For the experiments presented in this section, we compile C benchmarks from SPEC2006 with all debugging information present and only use them for comparison. We currently do not support reading complete debugging information for C++ and Fortran, yet we collected results on those benchmarks without comparing with source code.

The first experiment shows the quality of the recovered variables using the same metrics DIVINE [4] used for comparison purposes. DIVINE [4] compares recovered variables in the binary to corresponding variables in the source code of those binaries to determine how well it did. It defines four variable categories as a result: 1) a matched variable is a recovered variable whose exact size and position matches the variable from the source code. 2) An over refined variable is when the source code variable is divided into more recovered variables; for example, an integer identified as four characters. 3) Under refined variables which are recovered as part of a larger source code variable ; for example, an un-identified structure member. 4) An unknown variable is a variable which is not one of those mentioned categories.

As shown from figure 5.4, an average of 86% of the variables are matched to the debugging information. We run this experiment on programs ranging from 2,149 instructions (mcf) to 934,292 instructions (gcc). DIVINE [4] reports an average of 88% matched variables on programs ranging between 252 to 5,371 instructions. This shows that our

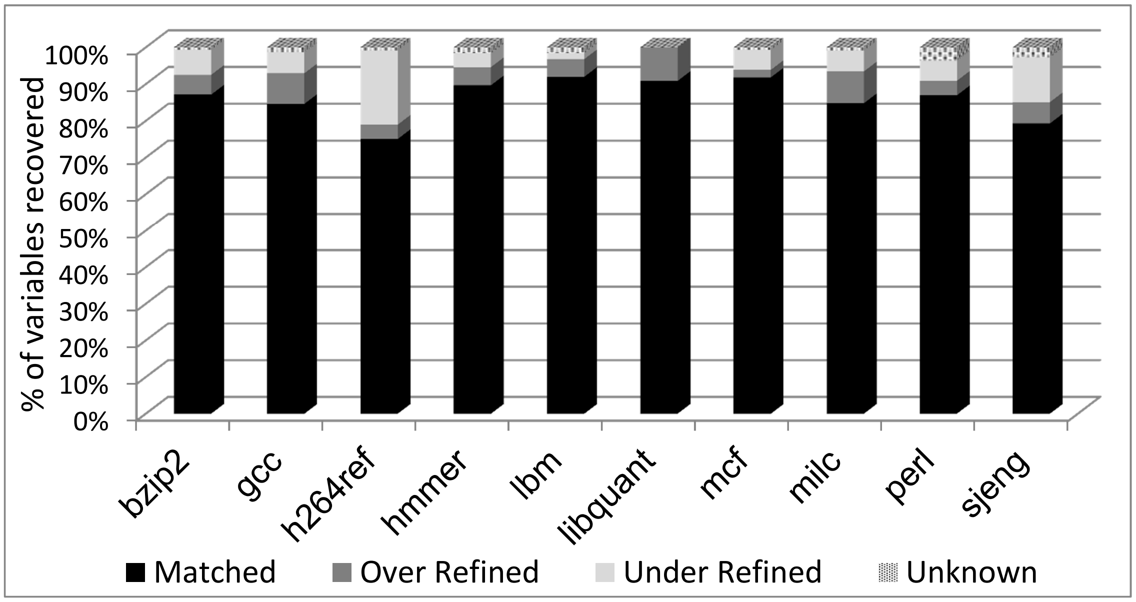


Figure 5.4: Accuracy of variable detection

schemes has comparable precision to DIVINE [4] but on much bigger benchmarks. The largest benchmark they report variables results on is *deltablue* with 5,371 instructions.

The scalability of the variables and type detection is shown in figure 5.7. Our analysis scales linearly with program size for larger binaries. The detailed benchmarks sizes were previously shown in figures 2.8 and 2.9 in chapter 2. The analysis takes around 6 minutes to analyze *tonto* which is a Fortran benchmark whose size is 1.3 million instructions. The average analysis speed is 1.7 seconds per 10000 instructions compared to 10 minutes per 10000 instructions in DIVINE. Thus our method is 352X faster than DIVINE on average. As mentioned before, the underlying reason for our much-faster analysis is using an underlying best-effort pointer analysis that is not guaranteed to have complete points-to sets. We consider that while recovering the IR to maintain correctness as we discussed earlier in section 5.2. dealII is the only program (out of 25) that did not scale well. dealII has very large number of procedures. The interprocedural data flow propagation took most of the time in dealII. Still, it is finishing in around 13 minutes given that it has 766,555 instructions.

In order to evaluate our type analysis techniques, we calculate the same metrics that TIE [43] uses. TIE defines a type range for every variable recovered from the executable. An ordering between basic types is specified by a type lattice shown in their paper. The first metric they define is the *distance* which is the difference between the lattice heights of the upper and lower bounding types for each type range. The smaller the *distance*, the more accurate the identified types are. The maximum distance is 4. They also define their detected type range to be *conservative* if the actual source code type falls inside the detected range.

In order to compare with TIE [43], we define a range of types for every variable we detect where the lower bound is the single detected type by our analysis and the upper bound is the generic `regx_t` type they define in their lattice, where x represents the number of bits of the underlying ALoc or register. Based on that range, we calculate our distances and conservativeness rates. Since the TIE paper is not clear about how to define conservativeness for structure and array types, we set their distances to 4. We also added floating point types to their lattice the same way the integer types are added. As an example, floats are added in the following order:

$$\top \texttt{ :> reg32\_t :> float :> } \bot$$

.

In addition to the distance and conservativeness, we define our own metric that measures the precision of multi-level pointers detection. TIE metrics do not show how multi-level pointers are precisely typed since all pointer types have the same height on their lattice [43]. Our precision metric is defined as the ratio between the correctly recovered pointer levels to the source level pointer levels. For example, if a variable has a double pointer to integer type (`int**`) in the source code and we identified it as a single pointer to an integer (`int*`), then we identified one level only out of the three levels in source, which are *pointer* to *pointer* to *integer*. Our precision in this example will be 33%.

Figure 5.5 shows the conservativeness as well as the precision of our detected types. The conservativeness rate is 96% on average which is slightly higher than 90% that TIE reports. Our precision metric shows that we detect 73% of the pointer levels on average. The average distance detected for our type recovery system is 1.7 which is slightly better than the distance of 2 that TIE [43] reports. Figure 5.6 shows the distance measured per benchmark.

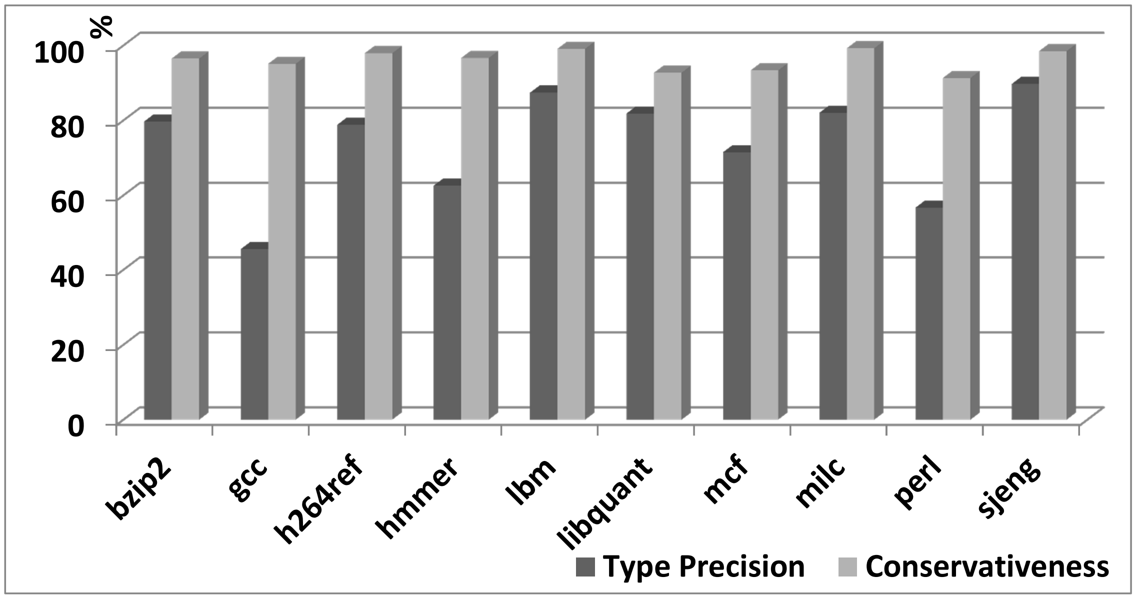


Figure 5.5: Accuracy of type detection

Some of the larger binaries have lower type precision than other smaller ones. This is expected since larger programs tend to have more higher level pointers than smaller ones and those are usually hard to detect since they rely on the effectiveness of the underlying pointer analysis. The conservativeness and distance measures used by TIE do not capture this fact as it is clear from figure 5.5.

It is worth mentioning that our variable and type recovery are integrated together in our system. The scalability shown in figure 5.7 is capturing both the variable recovery and the type analysis.

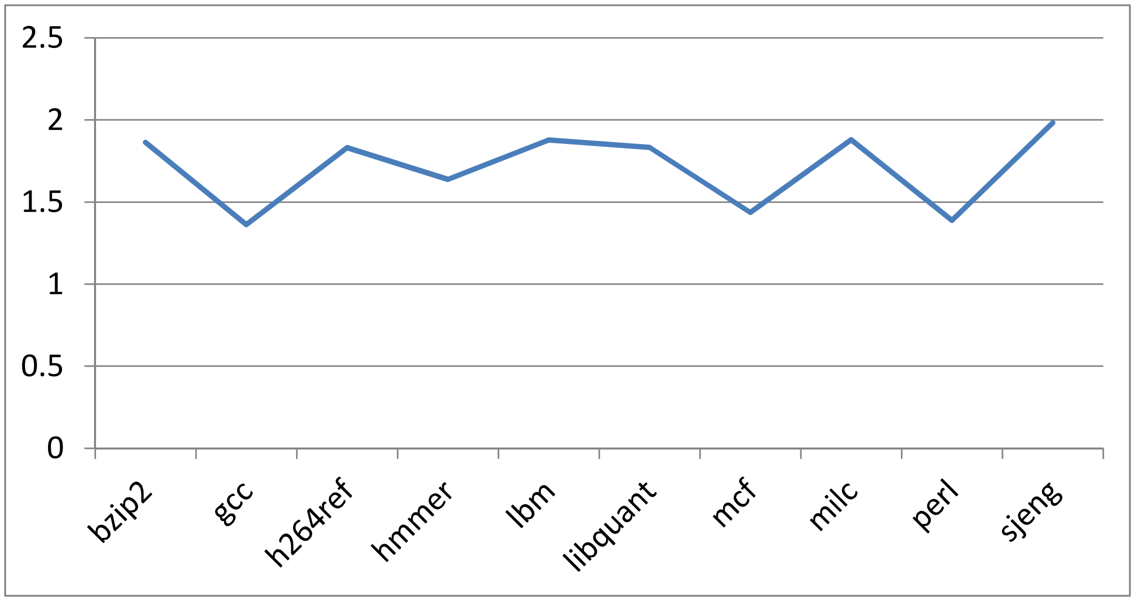


Figure 5.6: Distance of type detection

The recovered IR after our type analysis is usually of a higher quality than the one before our techniques. To evaluate this, we calculate the percentage of the IR symbols that have a non-generic type after our techniques. Results show that 91% of the IR symbols are typed in Fortran binaries, 88% of them are typed in C and C++ binaries, and 81% of them are typed in the visual studio binaries. Those binaries were compiled using the

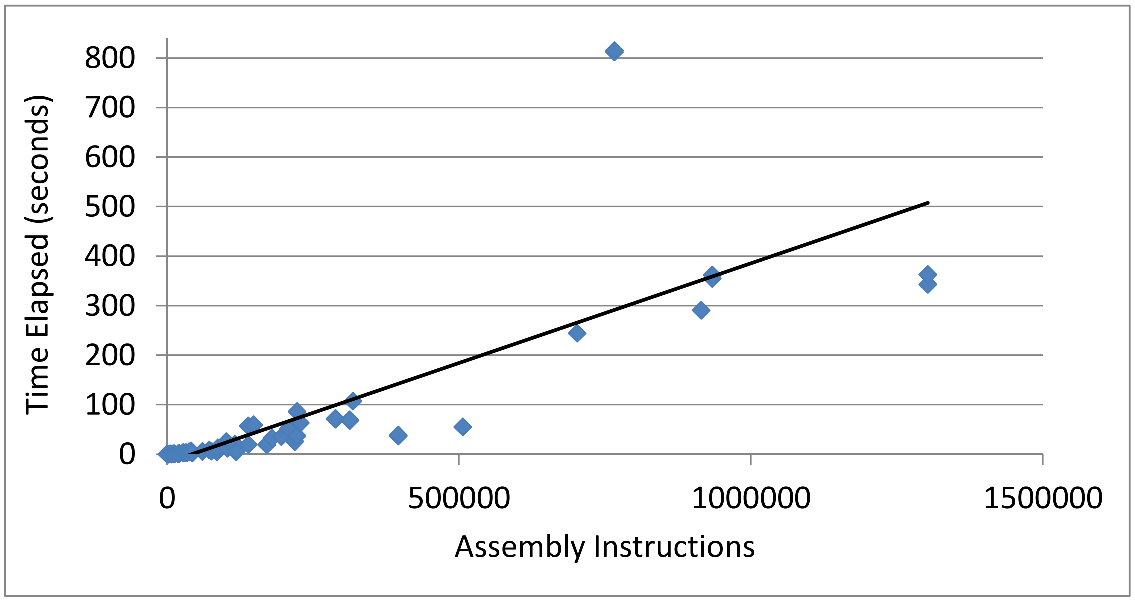


Figure 5.7: Scalability of variable and type detection

maximum compiler optimization level while conducting this experiment.

5.9 Related Work

Throughout this chapter, we compared our work with the most recent work done in the areas of variable and type recovery [4, 43] and function prototypes identification [10]. In this section, we discuss other work that is relevant to our techniques.

Binary rewriting has been considered by a number of researchers. There are two main categories when talking about binary rewriters, dynamic binary rewriters and static binary rewriters. Dynamic binary rewriters rewrite the binary during its execution. Examples are PIN [48], BIRD [52] and others. None of the dynamic binary rewriters found produce high-level compiler IR. Examples of existing static binary rewriters include ATOM [28], PLTO [60] and UQBT [16]. None of those binary rewriters employ a compiler level intermediate format, like LLVM IR or similar; rather they define their own low-level custom intermediate format. They do not detect high level features such as floating point stack variables, register arguments to functions and data types.

Boomerang [26] is an open source decompiler. It has very limited capabilities and cannot handle large binaries. Register arguments have to be specified manually. It does not detect any floating point stack operations. Emmerik mentions in his PhD thesis [69] some type recovery techniques based on SSA which are partially implemented in Boomerang. They have very limited memory tracking capabilities which are very important in recovering variables and data types as we stated earlier in this chapter.

Cifuentes et al. present techniques to recover high level C code from SPARC binaries [13]. There is no discussion on how to detect variables along with their data types. The paper is more towards recovering the high level constructs of the C language like conditionals, loops, etc.

Saxena et al. present an efficent binary instrumentation technique [58]. For their technique to be effective, they perform memory analysis similar to VSA [5] but limited only to stack memory to detect escaped local variables. They assume complete knowledge of accurate function boundaries.

REWARDS [44] presents a dynamic type recovery technique; TIE [43] shows better precision than REWARDS. We already compared to TIE [43] in our results. A technique to automatically reconstruct data types from binaries is presented in [22]. It is used in a tool that aims to produce C code from binaries; however no actual C code generation is demonstrated. One main disadvantage in their work is they do not track memory. As we have shown, tracking memory is very important in identifying accurate types. The analysis they produce is intraprocedural which limits its accuracy. Their algorithm is used by Torshina et. al. [68] in another attempt to reverse engineer data types in a tool named TyDec for program decompilation. An early work on type construction from binaries is by Mycroft [51]. It tries to construct C code from binaries with correct type information. However, it does not actually show results producing C code. The algorithm does not track memory locations and it is not clear if it can produce valid IR or C output code.

Many static custom memory analysis techniques in binaries exist. None of them recovers variables or data types. The VSA analysis by Balakrishnan and Reps [5] is used by them to implement various analyses. One of them is called DDA/x86 [6] which is used to detect bugs in device drivers. Device drivers were also analyzed using the Jakstab tool [36] using a modified version of the VSA technique [37].

A low level pointer analysis was proposed by Guo et al. [32]. It is a context sensitive, flow insensitive analysis detecting accurate points-to sets of registers and memory locations. They do not recover variables or data types in their analysis. Their technique can be used in place of ours, but as we show in this chapter, our technique is simple, scalable and sufficient for the application we are presenting.

Alias analysis on binaries was proposed by Debray et al. [21]. It detects aliases between registers using address descriptors. No real memory tracking takes place. The same problem is present in the static slicing technique on binaries by Cifuentes and Fraboulet [12].

Other types of alias analysis on executables were proposed. Speculative alias analysis of executables was proposed by Fernandez and Espasa [29] which increases aliasing information precision by introducing unsafe speculations at analysis time which might result in wrong analysis results in rare cases. Another probabilistic alias analysis for executables was proposed by Lu and Chen [47] which estimates the probability of two registers referring to the same memory address.

Chapter 6: Conclusion

In this dissertation, we presented a set of techniques that are essential for the core of any binary analysis and rewriting system. Our techniques can disassemble the complete executable binary code, produce accurate function interfaces, recover function APIs, recover variables and data types. Our techniques guarantee the correctness of any high-level IR recovery process based on the recovered code. Our techniques can handle stripped binaries without symbolic, relocation, or debugging information.

In chapter 2, we presented function boundaries recovery techniques that achieve 100% complete code coverage for most application code. The function boundaries are almost 100% accurate. We presented techniques to reduce the amount of disassembled spurious functions up to 4%. Our techniques perform better than all other binary analysis tools aiming at disassembling binaries and achieving functions with accurate boundaries.

In chapter 3, we defined APIs for the recovered functions. We presented precise techniques to recover accurate register arguments and returns information from binaries. Our techniques guarantee that external function calls present in the binary without any known prototype can still work correctly in our recovered IR.

In chapter 4, we presented techniques to recover variables that are allocated on the x86 floating point stack. These variables are often missed in most of the tools recovering variables from executables. Our techniques are sound and will always have a functional recovered IR for most application binaries.

In chapter 5, we extend the variables recovery process to include all memory allocated variables in executables. We also present techniques to recover data types for the recovered variables. The recovered data types include scalar, pointers, aggregates and recursive data types. Our techniques are 352X faster than current techniques which enables the analysis of very large binaries (up to a million instructions).

All the techniques presented were tested on the SPEC2006 benchmarks suite. The recovered IR was recompiled and the rewritten binaries worked correctly giving the same output as the original binaries. We presented in the dissertation detailed metrics showing the quality of the recovered IR.

Bibliography

[1] *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, April 2010.

[3] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 295–308, New York, NY, USA, 2013. ACM.

[4] G. Balakrishnan and T. Reps. DIVINE: Discovering variables in executables. In *Verification, Model Checking, and Abstract Interpretation*, pages 1–28. Springer, 2007.

[5] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 5–23. Springer Berlin Heidelberg, 2004.

[6] Gogul Balakrishnan and Thomas Reps. Analyzing stripped device-driver executables. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 124–140, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[8] Davidson R Boccardo, Arun Lakhotia, A Manacero Jr, and Michael Venable. Adapting call-string approach for x86 obfuscated binaries. *Simpósio Brasileiro em Segurança da Informaçao e de Sistemas Computacionais*, 2009.

[9] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In *Network and Distributed System Security Symposium* [1].

[11] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[12] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Software Maintenance*, pages 188 –195, oct 1997.

[13] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to high-level language translation. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 228–237, Nov 1998.

[14] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*, pages 192–199, 1999.

[15] Cristina Cifuentes. *Reverse compilation techniques*. PhD thesis, Queensland University of Technology, 1994.

[16] Cristina Cifuentes and Mike Van Emmerik. UQBT: Adaptable Binary Translation at Low Cost. *Computer*, 33(3):60–66, March 2000.

[17] Cristina Cifuentes and Doug Simon. Procedure abstraction recovery from binary code. In *Proceedings of the Conference on Software Maintenance and Reengineering*, CSMR '00, pages 55–, Washington, DC, USA, 2000. IEEE Computer Society.

[18] Robert Cohn, David Goodwin, P. Geoffrey Lowney, Norman Rubin, Robert Cohn, David Goodwin, P. Geoffrey Lowney, and Norman Rubin. Spike: An optimizer for alpha/nt executables. In *In USENIX Windows NT Workshop*, pages 17–24, 1997.

[19] M. Cova, V. Felmetsger, G. Banks, and Giovanni Vigna. Static detection of vulnerabilities in x86 executables. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, pages 269–278, Dec 2006.

[20] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1013–1019, 2000.

[21] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL, pages 12–24, New York, NY, USA, 1998. ACM.

[22] E. N. Dolgova and A. V. Chernov. Automatic reconstruction of data types in the decompilation problem. *Program. Comput. Softw.*, 35(2):105–119, March 2009.

[23] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 51–60, New York, NY, USA, 2013. ACM.

[24] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, Jim Gruen, and Rajeev Barua. Scalable variable and data type detection in a binary rewrite. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2014, Under Review.

[25] Khaled ElWazeer, Matthew Smithson, Kapil Anand, Aparna Kotha, , and Rajeev Barua. Scalable variable and data type detection in a binary rewrite. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2014, Under Review.

[26] M.V. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 27–36, Nov 2004.

[27] eSecurityPlanet Staff. eSecurity Planet News, 2011.

[28] Alan Eustace and Amitabh Srivastava. ATOM: a flexible interface for building high performance program analysis tools. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 25–25, Berkeley, CA, USA, 1995. USENIX Association.

[29] M. Fernandez and R. Espasa. Speculative alias analysis for executable code. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 222 – 231, 2002.

[30] GrammaTech. CodeSurfer by GrammaTech. `http://www.grammatech.com/products/codesurfer/overview.html`, 1998.

[31] Ilfak Guilfanov. Idapro Disassembler, Hexrays, 2005.

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

[33] L.C. Harris and B.P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, 2005.

[34] J.K. Hollingsworth, B.P. Miller, and J. Cargille. Dynamic program instrumentation for scalable performance tools. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 841–850, May 1994.

[35] Johannes Kinder and Dmitry Kravchenko. Alternating control flow reconstruction. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 7148 of *Lecture Notes in Computer Science*, pages 267–282. Springer Berlin Heidelberg, 2012.

[36] Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 423–427, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] Johannes Kinder and Helmut Veith. Precise static analysis of untrusted driver binaries. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 43–50, Austin, TX, 2010. FMCAD Inc.

[38] Johannes Kinder, Florian Zuleger, and Helmut Veith. An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 214–228, Berlin, Heidelberg, 2009. Springer-Verlag.

[39] C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 29–44, May 2010.

[40] Christopher Kruegel, William K Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.

[41] Arun Lakhotia, Davidson R. Boccardo, Anshuman Singh, and Aleardo Manacero, Jr. Context-sensitive analysis of obfuscated x86 executables. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, PEPM '10, pages 131–140, New York, NY, USA, 2010. ACM.

[42] Arun Lakhotia, Eric Uday Kumar, and Michael Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Trans. Softw. Eng.*, 31(11):955–968, November 2005.

[43] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *NDSS*. The Internet Society, 2011.

[44] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS* [1].

[45] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on*

Computer and Communications Security, CCS ’03, pages 290–299, New York, NY, USA, 2003. ACM.

[46] LLVM. The LLVM Compiler Infrastructure, 2003.

[47] Yu-Min Lu and Peng-Sheng Chen. Probabilistic alias analysis of executable code. *International Journal of Parallel Programming*, 39:663–693, 2011. 10.1007/s10766-010-0157-y.

[48] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. pages 190–200, 2005.

[49] Jinxin Ma, Zhoujun Li, and Chaojian Hu. Towards extracting control flow abstraction with static disassembly for binary code. In *Computational Intelligence and Communication Networks (CICN), 2012 Fourth International Conference on*, pages 430–435, Nov 2012.

[50] D. Melski, T. Teitelbaum, and T. Reps. Static analysis of software executables. In *Conference For Homeland Security, 2009. CATCH ’09. Cybersecurity Applications Technology*, pages 97 –102, march 2009.

[51] Alan Mycroft. Type-based decompilation. In *Proceedings of the 8th European Symposium on Programming*, volume 1576 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 1999.

[52] Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO ’06, pages 358–370, Washington, DC, USA, 2006. IEEE Computer Society.

[53] Igor V Popov, Saumya K Debray, and Gregory R Andrews. Binary obfuscation using signals. In *USENIX Security Symposium*, pages 275–290, 2007.

[54] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’99, pages 119–132, New York, NY, USA, 1999. ACM.

[55] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. Learning to analyze binary computer code. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 2*, AAAI’08, pages 798–804. AAAI Press, 2008.

[56] Nathan E. Rosenblum, Barton P. Miller, and Xiaojin Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE ’10, pages 21–28, New York, NY, USA, 2010. ACM.

[57] Kevin A. Roundy and Barton P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Comput. Surv.*, 46(1):4:1–4:32, July 2013.

[58] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO ’08, pages 74–83, New York, NY, USA, 2008. ACM.

[59] Benjamin Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 45–54, 2002.

[60] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. PLTO: A Link-Time Optimizer for the Intel IA-32 Architecture. In *Workshop on Binary Translation*, 2001.

[61] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. LLBT: An LLVM-based Static Binary Translator. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES ’12, pages 51–60, New York, NY, USA, 2012. ACM.

[62] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, February 1993.

[63] M. Smithson, K. ElWazeer, K. Anand, A. Kotha, and R. Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 52–61, Oct 2013.

[64] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, ICISS ’08, pages 1–25, Berlin, Heidelberg, 2008. Springer-Verlag.

[65] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’96, pages 32–41, New York, NY, USA, 1996. ACM.

[66] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’09, pages 441–452, New York, NY, USA, 2009. ACM.

[67] H. Theiling. Extracting safe and precise control flow from binaries. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 23–30, 2000.

[68] K. Troshina, Y. Derevenets, and A. Chernov. Reconstruction of composite types for decompilation. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 179 –188, sept. 2010.

[69] Michael James Van Emmerik. *Static single assignment for decompilation*. PhD thesis, The University of Queensland, 2007.

[70] L. Van Put, D. Chanet, B. De Bus, B. De Sutter, and K. De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on*, pages 7–12, Dec 2005.

[71] Giovanni Vigna. Static disassembly and code analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 19–41. Springer US, 2007.

[72] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 260–275, New York, NY, USA, 2013. ACM.

[73] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. Still: Exploit code detection via static taint and initialization analyses. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, pages 289–298, Washington, DC, USA, 2008. IEEE Computer Society.

[74] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 157–168, New York, NY, USA, 2012. ACM.

[75] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 299–308, New York, NY, USA, 2012. ACM.

[76] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Proceedings of the 2011 European Conference on Machine Learning and Knowledge Discovery in Databases - Volume Part III*, ECML PKDD'11, pages 522–536, Berlin, Heidelberg, 2011. Springer-Verlag.

[77] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 487–498, New York, NY, USA, 2013. ACM.

[78] Jingbo Zhang, Rongcai Zhao, and Jianmin Pang. Parameter and return-value analysis of binary executables. In *Proceedings of the 31st Annual International Computer Software and Applications Conference - Volume 01*, pages 501–508, Washington, DC, USA, 2007. IEEE Computer Society.