Accurate Recovery of Functions in COTS Binaries

A Dissertation presented

by

Rui Qiao

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

May 2017

Stony Brook University

The Graduate School

Rui Qiao

We, the dissertation committe for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Dr. R. Sekar - Dissertation Advisor
Professor, Computer Science Department

Dr. Michalis Polychronakis - Chairperson of Defense
Assistant Professor, Computer Science Department

Dr. Nick Nikiforakis - Committee Member
Assistant Professor, Computer Science Department

Dr. Aravind Prakash - External Committee Member
Assistant Professor, Computer Science Department
Binghamton University

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Accurate Recovery of Functions in COTS Binaries

by

Rui Qiao

Doctor of Philosophy

in

Computer Science

Stony Brook University

2017

Binary analysis and instrumentation play a central role in COTS software security. They can be used to detect and prevent vulnerabilities, mitigate exploits, enforce security policies, and so on.

Many security instrumentations work at the granularity of functions. However, unlike high-level languages, functions in binaries are not clearly demarcated. To complicate matters further, functions in binaries may have multiple entry points and/or exit points. Some of these entries or exits may not be determined simply by instruction syntax or code patterns. Moreover, many functions are reachable only through indirect control transfers, while some may be altogether unreachable.

In this dissertation, we present an approach that overcomes these challenges to accurately identify function boundaries, as well as calls and returns. Our approach is based on fine-grained static analysis, relying on precise models of instruction set semantics derived in part from our previous work.

In the later part of the work, we expand our investigation to recover the next crucial piece of information that is lost in high-level language to binary translation: the types and numbers of function parameters. We propose an approach that uses fine-grained binary analysis to address this problem. We evaluate this technique by applying it to enforce fine-grained control-flow integrity policies. While our approach is widely applicable to all binaries, when

combined with recovered C++ semantics, it provides significantly improved protection.

Contents

List of Figures

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisor, Professor R. Sekar, without whom the journey would have not been possible. I greatly appreciate his guidance, support and help throughout my PhD, especially during the time I faced difficulties and setbacks. His critical thinking and insights always inspired me; his enthusiasm and attitude for research also deeply influenced me. I thank him for his efforts and patience on my research.

Second, I would like to thank my committee members. I am grateful for the advices Professor Michalis Polychronakis has given on different parts of my work. I also thank Professor Nick Nikiforakis, Professor Aravind Prakash and Professor Rob Johnson for being in my thesis or prelim committee and providing constructive feedback.

I am fortunate to have two great internships during the PhD. They not only gave me opportunities to work on interesting problems and improve technical skills, but also brought me two wonderful summers. I would like to thank my two mentors, Mark Seaborn at Google and Pieter Hooimeijer at Facebook, for their help.

I would like to thank my colleagues in Secure Systems Lab and RiS3 Lab for the collaboration, discussion and company: Niranjan Hasabnis, Mingwei Zhang, Laszlo Sekeres, Wai-Kit Sze, Alireza Saberi, Tung Tran, Riccardo Pelizzi, Daiping Liu, Yarik Markov, Yaohui Chen and many more. I would like to thank my friends at Stony Brook, California, and many other places for their support.

Most importantly, special thanks go to my family. My parents have constantly supported me to pursue my goals and encouraged me during tough times. The hard-working and persistent spirit they taught me has been a lifelong gift to me. My greatest thanks go to my wife. I am heartily grateful for her support during the PhD.

Lastly, I would like to acknowledge support from the following grants: AFOSR grant FA9550-09-1-0539, NSF grant CNS-0831298 and CNS-1319137, and ONR grant N000140710928 and N00014-15-1-2378.

1 Introduction

Program analysis is the process of analyzing programs for various properties while program instrumentation concerns with changing program behavior by inserting additional code. Program analysis and instrumentation play a central role in software security. They are used in various tasks such as vulnerability detection and prevention, exploit mitigation, security policy enforcement, and malware analysis.

Program analysis and instrumentation can be applied to either source code, or native binaries. Source code based techniques can leverage abundant information that is available from high level languages, such as functions, variables and types. These information are unavailable for COTS binaries because they are either discarded during the compilation process, or stripped off before software distribution. Despite the lack of high-level information, binary based techniques are attractive because they are more widely applicable. For example, they can be used on proprietary software, third-party libraries, or even malware that are only present in binary form. Moreover, working with binaries is advantagous because a later and more faithful form of the program is used. This is in contrast with source code which is yet to be transformed by compilers and linkers that could alter various aspects of the program.

Binary analysis and instrumentation are only possible or most effective if high level abstractions and program constructs are available. Therefore, a recurring task is to recover various high level information from COTS binaries. In this report, we focus on a fundamental requirement for binary analysis and instrumentation: recovering functions. In the rest of this section, we discuss the motivation, involved tasks, challenges and existing techniques for function recovery, and also summarize our contributions.

1.1 Functions at Binary Level

Functions are among the most common constructs in programming languages. They are also the very basic building blocks for composing program logic. In source code, function declarations or definitions are available, which can be readily used by compilers or other source code transformation tools for analysis and instrumentation purposes. However, this is not the case at binary level.

Functions in binaries are just byte streams. And unlike high-level languages, functions in binaries are not clearly demarcated. Actually, there is no such requirement for binaries to be able to execute. At runtime, the bytes of functions are interpreted by the CPU as instructions, and control can be transferred into or out of a function, without the knowledge of function boundaries.

Other high level information are missing as well. Function types, just as variable types, are discarded before binaries are generated. Note that function type is defined as the number, location, and types of a function's arguments and return values.

However, function information is required by many binary analysis and instrumentation applications: various tools are designed to operate on functions. Therefore, when function information is not available, an essential step of many tools is to recover them. While the common task is to recognize functions in binaries, i.e., identifying their boundaries as well as entry/exit points, other higher level information such as function types are also often needed.

Function recovery needs to be *accurate*. As an early step in many analysis applications, function recovery significantly affects analysis accuracy due to the quality of its output. Moreover, demanding applications such as instrumentation can only be supported with highly accurate function information. Note that when supplied with inaccurate results, instrumentation techniques may generate programs that would crash or raise false alarms, significantly limiting their adoption and usability.

1.2 Challenges for Accurate Function Recovery

Although function recovery is essential and critical for many applications, producing accurate results for COTS binaries is difficult due to several challenges.

- **Missing abstractions.** Many abstractions are introduced in high level programming languages to cope with software complexity. However, after compilation, these abstractions are missing at the binary level. We already discussed function abstraction is not available and needs to be recovered. However, other abstractions (such as variables and types) that may facilitate the task of function recovery, are unavailable as well.

- **Stripped binaries.** Metadata for high level program constructs may be generated along with the compilation process. These include debug, symbol and relocation information. However, these information are often *stripped* off from COTS binaries, due to concerns such as their potential use for reverse engineering.

- **Indirect control flows.** In contrast to *direct* control flows whose targets are statically known, the targets for *indirect* control flows are determined at runtime. In general, it is an undecidable problem to statically resolve indirect control flow targets. Therefore, functions that are only *indirectly* reachable cannot be easily recognized. Note that the number of such functions can be significant, and it is necessary for many applications that these functions are accurately recovered.

- **Complex data flows.** In low-level code, registers and memory locations are basic units of keeping data. Program memory is further divided into global, stack and heap regions. However, all these regions are *physical*, instead of *abstract* as in high-level langauges. On the other hand, functions extensively operate on different regions (especially stack) for computation and data tranfers.

- **Compiler optimizations.** To squeeze performance gains, compilers may generate code in unusual ways. For instance, contrary to the high level abstraction that a function has a single entry point, a function in a binary may have multiple entries. Moreover, instead of being entered via a call instruction, tail call optimizations results in the use of jumps to enter a function. Because of these, intra- and inter-procedural control flow transfers cannot be easily distinguished by the instructions used, or a simple model of the target.

- **Hand-written assembly.** Hand-written assembly is necessary in implementing some low-level functionalities. For instance, `setcontext` and `getcontext` functions of libc are such cases because direct manipulation of stack pointer register is required. Furthermore, hand-written assembly can be useful in producing optimized code. One example is the `memset` and `memcpy` functions of libc. By implementing the functionality directly using carefully arranged assembly code, superfluous instructions that might otherwise be emitted by compilers are avoided. Moreover, additional CPU features such as non-temporal instructions [56] can be directly used for faster execution. However, "clever" uses of instructions in hand-written assembly often deviate from compiler generated code patterns, and can break assumptions made by binary analysis and instrumentation tools. Therefore, in order to be compatible with real-world and low-level applications, binary tools need mechanisms to detect and handle non-standard use of instructions.

1.3 Existing Function Recovery Techniques

There are three major techniques for function recovery, namely static analysis, dynamic analysis, and machine learning. Static analysis is the most widely used approach, which works by analyzing the binary module without program execution. However, techniques for indirect branch and memory reference resolution often lead to overapproximation. Moreover, imprecise modeling of instruction semantics and shalow analysis employed by many existing approaches result in inaccuracies in recovered functions.

Dynamic analysis is the technique of analyzing programs by executing them. By collecting execution traces and runtime data, some functions can be recovered. Although the problem of indirect branch and memory reference resolution is side-stepped because the data value is known for a particular run, the main drawback of this approach is its limited coverage. This results from the difficulty of determining the inputs (and possibly other environment factors) to excercise all parts of the program. Path explosion problem could pose further challenges. Due to limited code and path coverage, the recovered constructs are often incomplete and constrained in terms of applications.

Machine learning is a technique that can be applied to many problem domains. Two phases are involved. In the *training* phase, a dataset is consumed by the training program to build a model. This model is then used in the *testing* phase to produce results. Currently, machine learning is only adopted for function boundary identification, but not yet other aspects of function recovery. Based on the proposed techniques and observations of the evaluation results, current machine learning techniques have not effectively used deep program construct information or global evidence.

1.4 Contributions

Taking the completeness, applicability and accuracy requirements into account, we consider static analysis a more favorable approach than the other two. We utilize *deep* static analysis for function recovery: with precisely modeled instruction semantics, our approach uncovers critical properties associated with functions, and identifies useful constructs. This deep analysis is essential for robust support of demanding applications such as instrumentation, even on large and complex software.

In this dissertation, we present deep analysis techniques for accurate function recovery, with an emphasis on applications to the security domain. Our approach is different from existing work in the following aspects:

1. **Precise modeling of instruction semantics.** Due to the complexity of modern instruction sets such as x86, it is nontrivial to precisely model instruction semantics. Many analyses only loosely capture semantics of a subset of instructions, therefore not able to support large and complex software. Our approach utilizes precise model of instruction semantics,

which can not only be manually developed, but also automatically extracted from retargetable compilers.

2. **Detailed reasoning of the stack.** Program stack is an important memory region that is frequently accessed by functions. To recover function constructs or properties, the interaction of a function with stack should be captured and analyzed. While some existing approaches solely focus on registers [94], our analysis performs detailed reasoning of the stack. Therefore, not only stack variables values are tracked, entities such as return addresses and stack arguments can be determined.

3. **Accurate recovery of function constructs with deep analysis.** Compared with coarse-grained analysis [12] as well as machine learning [15, 87], our approach achieves significantly more accurate results. This is due to the fact that comprehensive analysis techniques have been leveraged to capture both local and global evidences, as well as deeper semantics. Relying on program semantics instead of instruction syntax or code patterns, our analysis is also able to recover constructs that are originated from hand-written assembly, which are easily missed by manual analysis or pattern matching.

4. **Robust enforcement of recovered constructs.** Many prior approaches [2, 62] recover function constructs for reverse engineering and program understanding purposes, therefore the results are not applicable to security enforcement. On the contrary, the recovered constructs from our analysis either directly support demanding applications such as instrumentation (in addition to more permissive applications such as analysis and reverse engineering), or possess desired properties which make them more amenable for further refinement. Moreover, different from binary rewriting oriented analysis [10, 37], our approach emphasize on robust enforcement of security policies based on recovered constructs. Therefore, a different set of trade-offs are made to support our use cases, and the analysis could be tailored for specific needs.

Specifically, we apply our analysis techniques to various problems, and make the following contributions:

- **Accurate discovery of function returns.** We present a technique for accurate function return discovery. A principled return-oriented programming (ROP) defense is developed as an application of our analysis. By systematically enforcing the inferred returns, our system can achieve stronger protection, better compatibility, and better performance as compared to previous research.

- **Accurate recognition of functions.** We present a systematic function recognition approach that is based on function start address enumeration and function interface checking. An in-depth evaluation shows that our system produces better results than state-of-the-art machine learning based approaches. Moreover, the recovered functions are more amenable for many analysis and instrumentation applications.

- **Accurate function type analysis.** We present techniques to accurately infer function types. A more fine-grained control-flow integrity (CFI) policy can be derived and enforced, based on the recovered function types. When combined with recovered C++ semantics, our analysis leads to stronger protections for virtual dispatches — a C++ construct commonly abused by attackers.

1.5 Dissertation Organization

The rest of this dissertation is organized as follows: Section 4 describes our technique for accurate function return discovery, as well as a principled ROP defense mechanism that is based on inferring and enforcing returns. Section 5 presents our approach for function boundary identification, which is a fundamental step for many binary analysis and instrumentation applications. Section 6 presents our techniques for accurate function type analysis and its application to fine-grained control-flow integrity enforcement. And lastly, Section 7 concludes.

2 Background and Related Work

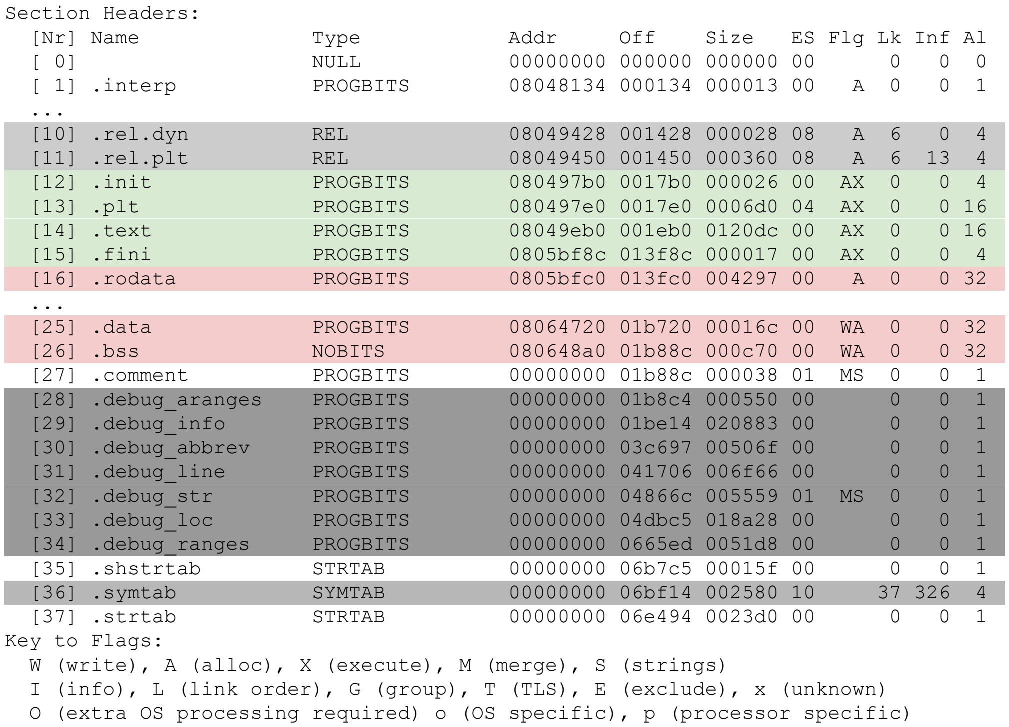
In this chapter, we discuss the background and related work for this dissertation. After describing existing analysis techniques related to function recovery in the first three sections, we shift our focus to their applications on security. Specifically, we introduce the common forms of code-reuse attacks in Section 2.4, and then discuss how these attacks can be mitigated with a general technique, namely control-flow integrity (Section 2.5).

2.1 Binary Organization and Disassembly

Program binaries are organized into *sections*. Each section may contain code, data, metadata, or other auxiliary information. Figure 2.1 gives an example for the organization of a Linux Executable and Linkable (ELF) binary. In this figure, a code section (e.g., `.init`, `.plt`, `.text`, `.fini`) consists of a sequence of bytes which is interpreted by the CPU as instructions and gets executed at runtime. A data section can either be read-only (e.g., `.rodata`), or have read-write permissions (e.g., `.data`, `.bss`). There may be metadata about the code sections (and data sections), most notably the *symbol*, *debug* and *relocation* information. However, they are normally stripped off before COTS binaries are distributed.

Disassembly is the process of translating bytes of code sections of a binary into decoded instructions. It is usually the first step for any binary analysis. There are two major techniques for disassembly: linear sweep and recursive traversal [85]. Each of these techniques has some limitations: linear sweep may erroneously treat embedded data as code, while recursive traversal suffers from completeness problems due to difficulties in statically determining indirect control flow targets.

Recent advances have shown that robust disassembly can be achieved with linear disassembly [11] and error correction mechanisms [106]. More specifically, the disassembly algorithm works by first linearly disassembling the binary, and then checking for errors such as (1) invalid opcode; and (2) direct control transfer outside the current module or to the middle of an instruction. These errors arise due to embedded data and are thus corrected by identifying data start and end locations so that disassembling can skip over them. Robust disassembly has been achieved by these techniques for a wide range of complicated and low-level binaries [106, 11], so we rely on the same techniques in this dissertation.



```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048134 000134 000013 00   A  0   0  1
  ...
  [10] .rel.dyn          REL             08049428 001428 000028 08   A  6   0  4
  [11] .rel.plt          REL             08049450 001450 000360 08   A  6  13  4
  [12] .init             PROGBITS        080497b0 0017b0 000026 00  AX  0   0  4
  [13] .plt              PROGBITS        080497e0 0017e0 0006d0 04  AX  0   0 16
  [14] .text             PROGBITS        08049eb0 001eb0 0120dc 00  AX  0   0 16
  [15] .fini             PROGBITS        0805bf8c 013f8c 000017 00  AX  0   0  4
  [16] .rodata           PROGBITS        0805bfc0 013fc0 004297 00   A  0   0 32
  ...
  [25] .data             PROGBITS        08064720 01b720 00016c 00  WA  0   0 32
  [26] .bss              NOBITS          080648a0 01b88c 000c70 00  WA  0   0 32
  [27] .comment          PROGBITS        00000000 01b88c 000038 01  MS  0   0  1
  [28] .debug_aranges    PROGBITS        00000000 01b8c4 000550 00      0   0  1
  [29] .debug_info       PROGBITS        00000000 01be14 020883 00      0   0  1
  [30] .debug_abbrev     PROGBITS        00000000 03c697 00506f 00      0   0  1
  [31] .debug_line       PROGBITS        00000000 041706 006f66 00      0   0  1
  [32] .debug_str        PROGBITS        00000000 04866c 005559 01  MS  0   0  1
  [33] .debug_loc        PROGBITS        00000000 04dbc5 018a28 00      0   0  1
  [34] .debug_ranges     PROGBITS        00000000 0665ed 0051d8 00      0   0  1
  [35] .shstrtab         STRTAB          00000000 06b7c5 00015f 00      0   0  1
  [36] .symtab           SYMTAB          00000000 06bf14 002580 10     37 326  4
  [37] .strtab           STRTAB          00000000 06e494 0023d0 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

Figure 2.1: Sections of an *unstripped* Linux ELF binary. Sections marked in green are code sections, and the ones in red are data sections. Sections marked in (different levels of) grey are symbol, debug, and relocation sections, which can be distinguished using their section names. When the binary is stripped, some of the grey sections are removed.

2.2 Discovering Code Pointers

There are different kinds of code pointers in a binary. Some are function pointers, while others are pointers to internal basic blocks of functions, and used for intra-procedural control flow transfers.

In this section, we first describe a general and conservative technique for identifying *all* code pointers, and then move to two specific code pointer structures, namely jump tables and virtual tables.

2.2.1 Conservative Code Pointer Analysis

Although it is undecidable whether a constant value in a binary represents a code pointer, conservative analysis techniques have been developed that identify a superset of possible code pointers. One recent approach [106] is to scan all constants in the binary, and select the subset that (a) fall within the range of code subsections within the binary, and (b) target a valid instruction boundary. Our function recognition technique (Section 5) starts from this conservative set, and prunes away almost all non-functions. As shown in our experiments, our analysis reduces the number of valid function pointers by a factor of 3.

2.2.2 Jump Table Analysis

A jump table is an array of addresses that are possible targets for an indirect jump (which we refer to as a *table jump*). Jump tables are generally used to implement intra-procedural `switch-case` statements in high-level languages. Existing work has developed analysis techniques to identify jump tables as well as their targets [26, 67]. The basic idea is to perform a backwards program slicing from each indirect jump instruction, and then compute an expression for the jump target. If the expression matches commonly used table jump patterns, the indirect jump is recognized as a table jump. The address of jump table can be extracted from the same expression, and its bound is obtained based on constraints imposed on the index variable. Finally, jump targets can be collected from the identified jump table.

2.2.3 Virtual Table Analysis

A virtual table (or VTable in short) is a data structure in C++ binaries used for virtual function calls. Since the main component of a virtual table is an array of virtual function pointers, the identification of virtual tables is helpful for code pointer analysis.

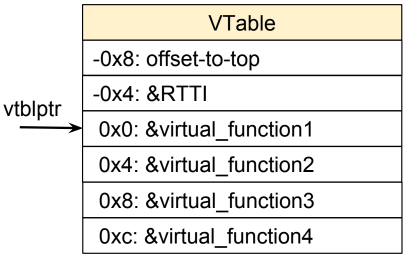


Figure 2.2: Layout of an Itanium C++ VTable

Although VTables can be structured in different ways, modern compilers follow system ABI (e.g., Itanium or MSVC) when generating VTable. Figure 2.2 presents the layout of an example VTable according to Itanium ABI [4].

In this figure, the first field "offset-to-top" represents the offset of a *vtblptr* field in an object. It is a constant that is either 0 (for single inheritance) or a small number (in case of multi-inheritance). The second field is a pointer to Run-Time Type Information (RTTI) [4]. However, it is 0 if RTTI is not available in the binary. The final part is an array of (virtual) function pointers. Note that *vtblptr* points to the start of this array, instead of the "offset-to-top" field.

The basic approach for virtual table identification is a set of heuristics [77, 103, 74] relying on system ABI. Since VTables locate in the read-only section of a binary, a constant found in disassembled instructions is identified as a candidate VTable pointer if its value corresponds to an address in the read-only section. Further checking is performed on the data surrounding the potential *vtblptr* address. Specifically, the data value at offset -4 should either equal to 0 or point to read-only section, and the data value at offset -8 should either equal to 0 or a small number. The values at positive offsets are scanned in a pointer-sized stride. Since they are supposed to be function pointers, the scanning stops when the value does not point to the `.text` section any more. Therefore, the potential VTable end is detected.

This technique has been shown to be effective to recover VTables without false negatives but only a few false positives [77, 74]. Moreover, the recovered VTables may be larger than the correct ones, but not smaller. As will be shown, this is desired and won't cause false alarms for security enforcement. Therefore, we leverage the same technique for VTable analysis.

2.3 Function Recovery

A function is essentially a body of code that operates on input data, and possibly returns values and/or exhibits side effects. Functions also have their own variables to keep track of internal state.

Depending on the application, there are many aspects to recover for functions in a binary. However, a first task is usually to recognize them. In the following sections, we discuss related work on recovery of important constructs, properties, as well as original source code for functions.

2.3.1 Function Recognition

Many tools recognize functions using call graph traversal and function prologue matching. Examples include CMU BAP [17], angr binary analysis platform [88], and the Dyninst instrumentation tool [50]. However, function prologue patten matching is not robust. Similarly, IDA [2] uses proprietary heuristics and a signature database for function recognition. Its problems include that it underperforms for different compilers and platforms, and the overhead of maintaining an up-to-date signature database.

Rosenblum et al. first proposed using machine learning for function start identification [81]. The precision and runtime performance have been greatly improved by recent work from Bao et al. [15] and Shin et al. [87], due to adoption of different machine learning techniques such as weighed prefix trees and neural networks. However, machine learning relies on a good training set, and potentially subtle parameter tuning for producing accurate results. Compared to analysis based approaches, they may also require an expensive training phase.

Necleus [12] is a recent effort of function recognition based on control flow analyis. Their approach builds a global control-flow graph (CFG) based on accurate linear disassembly. After removing direct call edges, the nodes with no inwards edges are considered as function starts. An interesting finding of the paper is that the dataset used by prior machine-learning techniques [15, 87] are biased because a lot of equivalent functions are present in the dataset. And a re-evaluation of those approaches on different datasets have shown significantly decreased performance.

Different from existing work, we develop a novel static analysis for function recognition (Section 5). Our approach is based on comprehensive function interface checking of control flow and data flow properties for potential functions. We demonstrate that *fine-grained* static analysis [79, 78] can recognize functions with much greater accuracy, and has the potential to support demanding applications such as automated analysis and instrumentation.

2.3.2 Function Type Recovery

Function type is defined as the number, location, and types of a function's arguments and return values. Since function type captures important aspects of function interfaces, with which functions interact with each other, it is critical for many applications.

The problem of function type recovery has been studied in the context of binary rewriting [10, 37]. To ensure the rewritten function receives all its original arguments, the number of arguments is over-approximated when static analysis is limited. However, this makes the technique less applicable to other applications.

TypeArmor [94] identifies the number of register arguments for both functions and indirect callsites, in order to enforce a finer-grained control-flow integrity (CFI). However, their technique relies on specific restrictions on the x86-64 platform, and does not detect stack arguments.

Complementary to TypeArmor, our work recovers *stack* arguments for functions and callsites. Based on a conservative analysis strategy and policy design, our technique can also be used for fine-grained CFI enforcement. Note that stack arguments information can be used on its own, or combined with register argument information, for stricter CFI policy.

2.3.3 Function Decompilation

Other than recovering certain constructs, a more ambitious goal for function recovery is full decompilation: converting functions to source code. Two critical tasks are involved: (1) variable and type recovery; and (2) control flow structure recovery.

Due to adoption of carefully chosen abstract domains, prior work based on abstract interpretation [28] has made significant progress on variable recovery [14, 10, 37]. However, they still cannot detect all variables. For type recovery, although both dynamic and static analysis based approaches have been proposed, the recovered types may either contain errors [63], or present in the form of type intervals rather than precise, single types [62].

Control flow structure recovery [84, 98] concerns with recovering high-level, structured control flow constructs such as loops, if-then-else branches, and switch-cases. A program is considered *structured* if it is free of `goto`s. Control flow structure recovery can assist program understanding and facilitate program analysis.

Because of above challenges and limitations of discussed techniques, existing decompilers are either designed for helping human audits, due to possible errors in their outputs [49, 38], or only tested with small programs [84], and with no demonstration of scalability.

In this dissertation, our focus is robust recovery of function constructs for large and complex software, with applications to security analysis and instrumentation. Therefore, full decompilation is not needed hence out of scope of this work.

2.4 Code-Reuse Attacks

Programs written in C/C++ are not memory safe. Vulnerabilities such as buffer overflow, heap overflow and use-after-free can be exploited by attackers to execute code of their choice. Traditionally, attackers inject payload (called shellcode) into the address space of a victim process, and redirect control to this code. However, with widespread deployment of Data Execution Prevention (DEP), injected code is no longer executable, so attackers have come to rely on code reuse attacks. The idea is to repeatedly redirect control flow to existing benign code in the address space. Since the target code snippets are carefully chosen by attackers, malicious computations can be performed.

Based on abused control flows, code-reuse attacks can be roughly classified as return-oriented programming (ROP), call-oriented programming (COP), and jump-oriented programming (JOP). These techniques can be combined to be more evasive [47, 32, 48, 20, 19, 39]. Since JOP is less utilized due to the relative small number of indirect jumps in a program, we focus our discussion on the first two forms.

2.4.1 Return-Oriented Programming

Return-oriented programming (ROP) [86] is the most prevalent form of code-reuse attacks. It makes use of "gadgets," i.e., existing code snippets ending with return instructions. A gadget used in a ROP attack could either be an intended code sequence, or unaligned code, which refers to unintended instruction sequence beginning from the middle of an (intended) instruction. This is possible on variable-length instruction-set architectures such as x86.

To carry out ROP, attackers first get control of the stack. Since return addresses used by gadgets originate from the stack, attackers prepare the stack in such a way that execution of the selected gadgets are chained one after another. The stack is also used to supply necessary data to the gadgets.

2.4.2 Call-Oriented Programming

Another form of code-reuse attacks is call-oriented programming (COP) [47, 39]. In this case, indirect calls, rather than returns, are abused by attackers. Generally speaking, COP is more difficult than ROP due to the need to repeatedly corrupt code pointers that may reside at different locations. Therefore, COP may sometimes only be used to bootstrap ROP [47].

Counterfeit object-oriented programming (COOP) [83, 29] is a special form of COP for C++ programs. By crafting counterfeit objects and abusing virtual function calls, attackers can repeatly redirect control to selected virtual functions (called "vfgadgets"). COOP is practical because it takes advantage of code patterns commonly found in object-oriented languages such as C++.

2.5 Control-Flow Integrity

Control-flow integrity (CFI) is a low-level security policy that constrains program control flows [6]. The basic idea is to pre-compute a static control flow graph (CFG) of the program, and enforce that the indirect control flow transfers are compliant with the CFG. CFI on its own can be used for defenses against code reuse attacks, or it can serve as a primitive to ensure non-bypassability of other inline reference monitors [6, 105, 101].

CFI restricts indirect control flow transfers, either *forward* or *backward* ones. Forward control flow transfers include indirect calls and indirect jumps, while backward ones refer to returns. Since in a CFG the instructions are represented as nodes and control flow transfers are edges, we also call these transfers forward and backward *edges*.

The strength of CFI depends on the precision of the CFG. A perfect CFG would only allow *intended* control flow transfers. However, as CFG is inherently static and the analysis that computes it usually *overapproximates* allowed indirect control flow targets to be conservative, the result policy is often overly permissive. Although function type information can be leveraged to further restrict forward edges, backward edges are usually difficult to constrain as a return site may legitimately target many locations. Based on the relative precision of CFG, CFI schemes can be roughly classified as *coarse-grained* or *fine-grained*.

CFI is a general technique that can be applied to any native code. However, CFI precision can be greatly improved if advanced language features can be recovered. In the next sections, we first describe language-agnostic CFI techniques that are widely applicable, and then discuss how C++ virtual calls can be more precisely protected with recovered C++ semantics.

Calling convention	Arch.	Arguments passing
System V ABI	x86-64	RDI, RSI, RDX, RCX, R8, R9, then stack
Microsoft x64	x86-64	RCX, RDX, R8, R9, then stack
cdecl	x86-32	stack
stdcall	x86-32	stack

Figure 2.3: Arguments passing for different calling conventions

2.5.1 Language-Agnostic Approaches

To compute the CFG, programs need to be statically analyzed. They also need to be instrumented for runtime enforcement. These tasks can be performed either by a compiler on source code, or by some low-level tool on binaries. Therefore, CFI can be classified as compiler- and binary-based approaches.

Compiler based approaches can leverage source code information such as function types [71, 92, 73, 66], therefore providing a better precision. However, they are not applicable to COTS software or third party libraries. This limitation can be significant: even if a single module is not protected, it could become the weakest link and render all other defenses useless.

On the other hand, although binary-based protection can be more complete, enforcing fine-grained CFI on COTS binaries is challenging due to lack of high-level information. As a result, the first binary level schemes sacrifices security for robustness, and used a relaxed policy that a large target set is allowed by each indirect control flow transfer type [106, 104]. However, these systems are bypassable [47, 32]. To provide stronger protection against determined adversaries, later research efforts have combined CFI with techniques such as randomization [68] or safe loading [107], or also consider context information [93].

TypeArmor [94] is a technique inspired by source-level type-based CFI enforcement. By analyzing indirect callees and callsites, it derives an under-approximation of the number of arguments passed to each callee, and an over-approximation of the number of arguments prepared by each callsite. It enforces a policy that a callsite can only call a callee if the prepared argument count is not smaller than that of the callee. Policy for return values is also in a similar, conservative fashion.

A limitation of TypeArmor is that it focuses only on register arguments. Although for x86-64/Linux it is effective (because the first 6 arguments are passed though registers), the technique does not apply to x86-32 where most arguments are passed through stack, and its precision significantly decreases

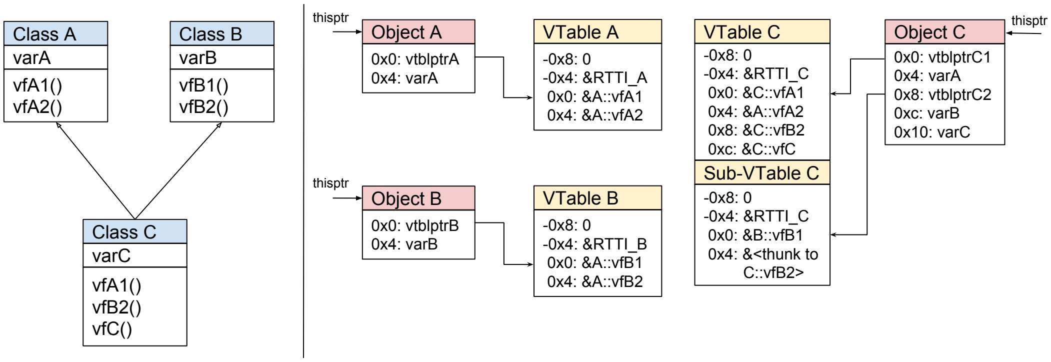


Figure 2.4: Layout for polymorphic objects

on x86-64/Windows (only the first 4 arguments are passed through registers and the rest are through stack). A summary of register passing for common calling conventions is presented in Figure 2.3. Our function type analysis complements this by focusing on memory arguments. It not only directly works with x86-32, but can also improve the CFI precision for x86-64 when combined with TypeArmor.

2.5.2 CFI for C++ Virtual Calls

As discussed in Section 2.4.2, C++ programs give attackers additional opportunities to develop advanced exploit techniques, such as those leveraging virtual function calls [83, 29]. On the other hand, C++ semantics available from source code can be used for stronger protection. Specifically, object type information as well as class hierarchies can be effectively leveraged for highly precise CFI enforcement [57, 92, 72, 16, 102].

For COTS binaries, however, these semantics are mostly unavailable. To illustrate related concepts, we use an example in Figure 2.4. The left half of the Figure depicts three classes A, B, and C. C is the subclass of both A and B, therefore it is a case of multiple inheritance. The right half shows the object layout of these polymorphic classes. For each object instance, the first field is a VTable pointer, while the next ones are member variables. Each object is associated with the VTable according to its class. Specifically, the VTable pointer of each object points to the virtual function array inside the corresponding VTable. Note that an object of class C has two VTable pointers, as it inherits from two classes. The first VTable pointer is still the first field of the object and points to $VTable\ C$, while the second VTable pointer is at a larger offset and points to $Sub\text{-}VTable\ C$.

In Figure 2.4, the RTTI pointers in VTables either point to RTTI data structures, or are 0 if such information is not available. Although RTTI information [4] (which encodes class hierarchies) may present in *some* stripped binaries, it is mostly adopted for reverse engineering [42, 99], and difficult to apply to enforcement. Techniques for deriving class hierarchies *without* RTTI information have also been developed [43, 58]. However, the recovered hierarchies are still not precise enough for CFI enforcement without the help of dynamic profiling [74].

Due to the challenges of accurately recovering class hierarchies as well as object types, existing C++-aware CFI techniques rely on a subset of C++ semantics and constructs. Specifically, only virtual callsites and virtual tables (VTables) are identified and used for generating a CFI policy [103, 77, 46]. Although these policies are not as precise as those from source code based techniques, they still represent a significant improvement over approaches not considering C++ semantics.

Since we discussed VTable identification in Section 2.2.3, next we focus on identification of virtual callsites.

Identification of Virtual Callsites Prior works have developed techniques to identify virtual calls in binaries [46, 77, 103]. The basic approach is to slice and transform code so that each indirect call target as well as its first argument are represented as expressions; if these expression satisfy certain constraints, the corresponding indirect call is recognized as a virtual call. Specifically, since the target of a virtual call should be derived from a VTable, the target must be a dereference of form $*(\texttt{ptr1} + \texttt{offset})$. Note that `offset` should be a multiple of pointer size for the underlying architecture, and is possibly 0. Moreover, for `ptr1` to be a VTable pointer, it has to be the first field of an object. In other words, `ptr1` should derive its value from another dereference, and its expression should be of form $*(\texttt{ptr2})$. If `ptr2` is passed as the first argument of the indirect call, a virtual call is identified, and `ptr2` is recognized as the `this` pointer.

Note that this technique also works for multiple inheritance and virtual inheritance, as the `this` pointer is adjusted by the compiler before invoking virtual calls. For example, in Figure 2.4, before any virtual function from *Sub-VTable C* is invoked, the `this` pointer is adjusted by adding 0x8 to point to $vtblptrC2$. Therefore, although `ptr2` is an expression that can take different forms, a virtual call is recognized as long as the two dereferences described above and the argument passing are identified.

Evalutions on this technique have shown that although it may miss some virtual calls (false negatives), there are no false positives [77]. Since the missing virtual callsites would not result in false alarms but only slightly decreased security, we utilize the same technique for virtual call identification.

CFI Policies Since virtual calls must target virtual table entries (virtual functions), a basic policy is to enforce this invariant, and therefore no other targets are allowed. Earlier work has not developed VTable identification techniques, therefore a relaxed policy is used [46]. Specifically, since VTables must present in read-only sections, a virtual call is allowed if the target is found in those sections. VTint [103] relocates VTables into a separate read-only section, and instruments virtual callsites to check that the target VTable is read-only. However, VTint is not able to prevent VTable reuse attacks.

vfGuard [77] proposed a stronger CFI policy with accurate detection of virtual calls and VTables. Specifically, since each virtual call uses a constant offset to index into a VTable and retrieve virtual function pointer (`offset` in the expression $\texttt{call} * (*(\texttt{thisptr}) + \texttt{offset}))$, this constant can be statically determined and effectively used. For example, a virtual call with offset 16 can only target entries of offset 16 of all VTables. If a VTable is small and offset 16 is out of its bounds, then this VTable cannot be a candidate VTable for the virtual call. vfGuard further refines the target sets based on other properties such as calling conventions.

2.5.3 Shadow Stack

As discussed, backward edges cannot be well protected by CFI as the CFG computed by a static analysis is not precise. Therefore, a shadow stack is typically incorporated with coarse-grained CFI [7].

Shadow stack schemes [44, 24] were first proposed as a defense for stack smashing attacks. However, a shadow stack alone is not effective against ROP attacks, as only legitimate returns were checked and ROP attacks using unintended returns are possible. CFI enforcement, which prevents the use of unintended instructions, provides one way to block this attack avenue. A second approach, used in DBT-based techniques (e.g., ROPdefender [33]), is to instrument *all* returns before their execution.

The practical deployment of shadow stack has been limited by the prevalence of non-standard returns that violate shadow stack checks. While RAD [24] addressed the cases of `longjmp` and signals, ROPdefender [33] identified two other non-standard uses: C++ exceptions and lazy-binding of calls to shared library functions. It handled them by manually identifying instructions that save a return address on the stack, and pushing a copy on the shadow stack.

A drawback of ROPdefender was its significant runtime overhead. Zhang et al [105] discuss how dynamic binary instrumentation techniques, while displaying good performance on SPEC benchmarks, tend to perform poorly on large, real-world applications. Being based on static instrumentation, Zhang et al were able to achieve significantly better performance than ROPdefender.

Lockdown [75] is a recent effort combining shadow stack and CFI in dynamic instrumentation, while focusing on reducing runtime overhead. However, they do not focus improving compatibility.

Dang et. al surveyed existing shadow stack systems and designed a "parallel shadow stack" scheme [31] to eliminate the need for shadow stack pointer save and restore. They avoided register clobbers in their intrumentation, applied peephole optimizations, and achieved great performance. However, this comes with some trade-offs on security. In fact, StackDefiler [27] describes an attack that leaks shadow stack address.

2.5.4 Comparison with Other Defenses

The most comprehensive defense for memory corruption attacks is based on bounds-checking [59, 97, 100, 9, 69]. Unfortunately, these techniques introduce considerable overheads, while also raising significant compatibility issues [91]. LBC [51] achieves lower overheads while greatly improving compatibility by trading off the ability to detect non-contiguous buffer overflows. Code pointer integrity [61] significantly reduces overheads by selectively protecting only those pointers whose corruption can lead to control-flow hijacks. However, these solutions all require source code.

3 Static Analysis Approach

3.1 Intermediate Representation

To analyze binaries, a representation of binary code is required. Disassembly is a necessary step because it discovers code bytes and decodes them into instructions. However, modern instruction sets such as x86 are complex and the disassembled instructions are not amenable to direct analysis.

To address this problem, instructions are usually transformed into an intermediate representation (IR). The IR is much simpler in that it only consists of a small number of operations, and all semantics are represented explicitly with IR instructions. Since the IR is usually in a flat three address code form, the data movement sources and destination are also explicit. Due to these properties, IR can be directly consumed by analysis engines.

An IR can be a newly designed language, examples include the Valgrind IR [70], BAP (Binary Analysis Platform) IR, and REIL (Reverse Engineering Intermediate Language) IR. Alternatively, an existing IR, typically used in compilers, can be adopted. For example, SecondWrite [10] chose LLVM IR, while other works have used the GCC RTL (Register Transfer Language) IR [55, 54, 80].

The benefit of using a new IR is that it can be designed to satisfy specific requirements. However, it requires significantly more work. On the other hand, using an existing compiler IR is advantageous in that many existing software components, such as optimization passes of a compiler, can be adapted or directly used. Moreover, it also facilites support for multiple platforms, if an IR from a retargetable compiler (such as GCC and LLVM) is used.

In the next sections, we describe the two tasks involved in obtaining an IR form of a binary, namely instruction semantics modeling, and IR transformation.

3.1.1 Precise Modeling of Instruction Semantics

To make sure the transformed intermediate representation faithfully represent the semantics of original program, precise modeling of instructions is required. Earlier works manually specify instruction semantics, based on the instruction set architecture (ISA) manual from the CPU venders [70, 17, 35, 10, 60]. However, since modern instruction sets are complex, this task requires huge human efforts and is prone to error.

Recent research has proposed techniques for to alleviate this. The key observation is that compilers, which transform source code into binaries, already have a model of instructions. Therefore, different approaches have been developed to automaticaly extract instruction semantics from compilers. LISC [55] is a learning-based system that trains a model with GCC RTL IR and assembly pairs obtained from the compilation of a large number of packages. While EISSEC [54] takes a different approach by symbolically executing the code generators in the GCC compiler.

To validate the correctness of recovered instructions semantics, testing approaches can be leveraged [52].

3.1.2 IR Transformation

To transform a binary into IR form, the disassembled instructions are used as inputs to a "lifter" program. The lifter transforms each assembly instruction into a number of IR statements. The original instruction opcode has been represented with IR operations, while the operands are translated to their counterparts in the IR. Note that the lifter makes use of an instruction specification, either manually specified or obtained using automatic approaches, so that the generated IR has the same semantics as original program.

Our lifter is built on top of LISC [55], which transforms assembly instructions into GCC RTL IR. However, since RTL is tree-structured, it is not suitable for direct analysis. We therefore further transform RTL into a flat, and simple IR that is in three-address form.

The core algorithm, `flatten`, is a recursive procedure that takes any RTL tree node as input. For each RTL operation encountered, it outputs a corresponding IR statement, and temporaries are used as desired. Therefore, a single RTL instruction is translated to multiple IR statements, which as a whole captures the instruction semantics. These IR statements consist of a handful of operations, such as arithmetic operations, memory dereferencing, and assignment, therefore they can be easily used in analysis.

3.2 Data-Flow Analysis

Data-flow analysis is a set of techniques that gather information about the flow of data along program execution paths [8]. Data-flow analysis is widely used in compilers as a basis for optimization. For binary programs, data-flow analysis is also the foundation for capturing important program properties.

The task of a data-flow analysis is to associate each program point with a *data-flow value* — an abstraction of all possible program states that can be observed from that point. Usually, only data-flow values at basic block boundaries are considered. This is because with such information it is easy to compute the data-flow value before and after any internal instruction of a basic block.

Different data-flow analysis concerns with different data-flow values. Moreover, data-flow analyses can also be classified as *forward* and *backword* ones, based on the directions of concerned data flows.

Suppose we use $IN[B]$ and $OUT[B]$ to respectively represent data-flow values at the entry and exit of a basic block B, for a forward data-flow analysis, the following *data-flow equations* are generated:

$$OUT[B] = trans_B(IN[B])$$
$$IN[B] = join_{P \in pred_B}(OUT[P])$$

In these equations, $trans_B$ is the *transfer function* for basic block B. It represents the transformations of data-flow values based on the semantics of instructions of B. The *join* operation combines exit data-flow values from all B's predecessors.

For backward data-flow analysis, the equations are the opposite:

$$IN[B] = trans_B(OUT[B])$$
$$OUT[B] = join_{S \in succ_B}(IN[S])$$

Note that $IN[B]$ derives new value from $OUT[B]$, and the *join* combines entry data-flow values from all B's successors.

To get output for a data-flow analysis, data-flow equations need to be solved. This is usually done with an iterative algorithm. Specifically, all the IN states are first initialized (typically with $\emptyset$), and then the OUT states are updated, based on the equations. This process continues until a *fixpoint* is reached. The IN and OUT states at this point represent the final output of the analysis.

In the following sections, we introduce two specific data-flow analyses that have been used in our function recovery tasks.

3.2.1 Reaching Definition Analysis

Reaching definition analysis is a forward data-flow analysis that determines for each program point which definitions may reach it. The data-flow equations for a reaching definition analysis is as follows:

$$REACH_{out}[B] = GEN[B] \cup (REACH_{in}[B] - KILL[B])$$
$$REACH_{in}[B] = join_{P \in pred_B}(REACH_{out}[P])$$

In above equations, we used $REACH_{in}[B]$ and $REACH_{out}[B]$ to represent the reaching definition data-flow values at the entry and exit of a basic block B. Note that the second equation is no different from the one shown in last section. Intuitively, this means that a definition that can reach the exit of any predecessor basic block reaches the current block.

The right-hand side of the second equation corresponds to the transfer function for reaching definition analysis, and it consists of some new elements. Specifcally, $GEN[B]$ is the set of definitions inside B that are visible immediately after B, while $KILL[B]$ is the union of all definitions killed in B. Note that a "kill" means that the variable is overwritten, by a new definition of the same variable.

$GEN[B]$ and $KILL[B]$ are derived from $GEN[I]$ and $KILL[I]$, where I is any instruction (or equivalently, any IR statement) of B. Note that this requires instruction semantics, which can be obtained using approaches described in the first section of this chapter.

3.2.2 Liveness Analysis

Liveness analysis is a backward data-flow analysis that computes at each program point the variables that may be potentially read before the next write, i.e., variables that are *live*. In simpler words, a variable is live if it may be needed in future of the execution. A variable is said to be *dead* if it is not live.

The data-flow equations for a liveness analysis is as follows:

$$LIVE_{in}[B] = GEN[B] \cup (LIVE_{out}[B] - KILL[B])$$
$$LIVE_{out}[B] = join_{S \in succ_B}(LIVE_{in}[S])$$

The second equation is no different from the general equation for backward data-flow analysis. For the first equation, note that $GEN[B]$ and $KILL[B]$ are defined differently as in reaching definition analysis. Specifically, $GEN[B]$ represents the set of variables that are used in B before any assignment, while $KILL[B]$ represents the set of variables that are assigned a value in B.

3.2.3 Static Single Assignment

To simplify data-flow analysis, IR can be first transformed into a special form called static single assignment (SSA). In SSA form, each variable is defined *exactly once*, and each variable is defined before used. The same variable defined at different places are usually denoted with different subscripts.

For example, consider the following piece of IR code:

$x = 3$

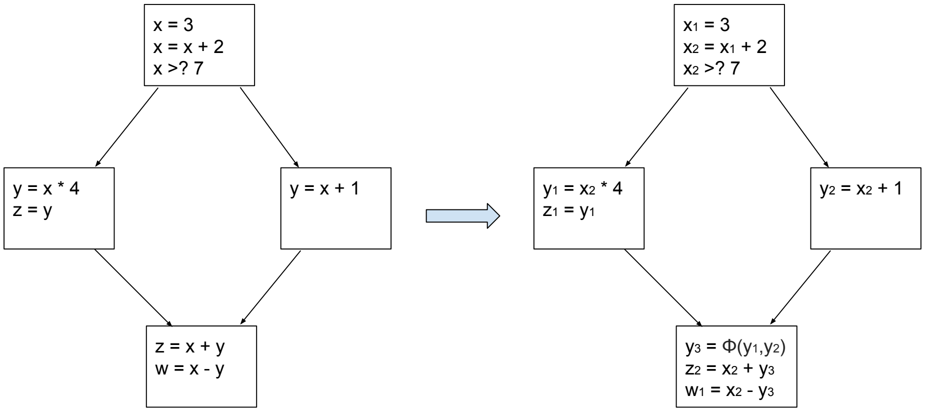


Figure 3.1: IR code and its SSA form

$y = 4$
$x = y + 5$

It can be transformed to the following SSA form:

$x_1 = 3$
$y_1 = 4$
$x_2 = y_1 + 5$

Note that at line 3, since x is redefined, a new version (x_2) of the variable is created. The same statement uses variable y, which corresponds to its definition at line 2, hence the same version (y_1) is used.

An advantage of SSA, as can be seen from above example, is that the def-use chain is explicit. This simplifies further analysis and optimizations on the IR.

Figure 3.1 presents some IR code that consists of several basic blocks (on the left), as well as its SSA form (on the right). Note that when multiple definitions of the a variable reach the same location, a ϕ (Phi) function is used to indicate the definition may be from different sources. As shown in the figure, y at the last basic block is defined either in the second or third basic block, therefore a special IR statement $y_3 = \phi(y_1, y_2)$ is inserted.

Algorithms have been developed for transforming IR code into (and out of) SSA form [30]. At the binary level, both registers and memory locations can be transformed [95].

3.3 Abstract Stack Analysis

One common task for analyzing a function is to compute its effect on registers and memory. Since stack frames are used to store local variables and pass

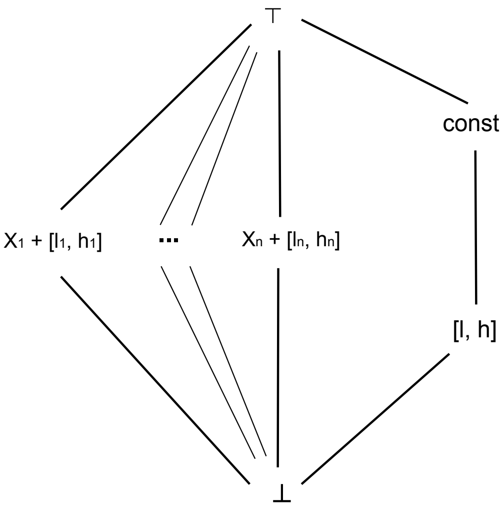


Figure 3.2: The abstract domain for stack analysis

arguments, a function frequently interacts with them. To get accurate results, it is important that stack memory is precisely modeled.

We develop an abstract interpretation [28] based *abstrack stack analysis* (ASA) for this purpose. For each function, it computes an overestimation of values for registers and stack locations, based on semantics of lifted IR of a function. The abstract domain we have used is similar to that of Reference [82], and it is depicted in Figure 3.2.

As shown in the figure, each abstract value is in the form of $Base + [l, h]$, where $Base$ and h are optional. $Base$ is a symbolic value that represents the value of a register or stack location at function entry. A unique symbolic value is associated with each distinct register or stack location. A missing $Base$ is treated as 0, and a missing h is treated as the same as l. The values l and h are (possibly negative) integers. Note that a special $Base$, $BaseSP$ is used to represent the initial value of the stack pointer `esp`. Moreover, stack memory locations are referenced with respect to this base value — typically with an offset to $BaseSP$.

An abstract value $X + [l, h]$ indicates that the concrete value will be in the range of $X + l$ to $X + h$ (inclusive). The value $[l, h]$ denotes a concrete value from l to h. A special abstract value *const* represents an unknown concrete value, with one limitation: when used as an address, it cannot reference local memory.

Figure 3.3 describes the abstract interpretation in ASA. For each IR statement, the abstract store A is updated, based on statement semantics. In this

IR Statement	Abstract Store (A)
$R := c$	$Upd(A, [R \mapsto [c, c]])$
$R := R'$	$Upd(A, [R \mapsto A[R']])$
$*(R) := R'$	$Upd(A, [x \mapsto A[R']])$, if $A[R] = x$ $Upd(...(Upd(A, [x_1 \mapsto A[x_1] \cup A[R']])...$ $...), [x_n \mapsto A[x_n] \cup A[R']])$ if $A[R] = x_1, ...x_n, n > 1$
$R := R_1 + R_2$	$Upd(A, [R \mapsto \cup_{r_1 \in A[R_1], r_2 \in A[R_2]} r_1 \oplus r_2])$
$call(f)$	$Upd(...(Upd(A, [x_1 \mapsto applysum(A[x_1], f, x_1)])...$ $), [x_n \mapsto applysum(A[x_n], f, x_n)])$ where $ModifiedNonLocal(f) = x_1, ...x_n$

Figure 3.3: Abstract interpretation for stack analysis

figure, R (possibly with a subscript) denotes a register. The abstract store associates up to k (where k is a small constant) abstract values to a register or memory location. The notation $A[l]$ means the abstract value stored at location l for abstract store A, and *Upd* is used for updating the abstract store. Note that there are two cases for stack memory updates. If the location is a singleton, then the abstract value is replaced with a new one. However, if multiple locations are involved, each possible location is updated with its original value, combined with the new value. Moreover, note that unlike stack memory, there is only a single location representing global and heap memory, and it is initialized as *const*. This captures the assumption that stack variables are created *after* a function is invoked, and no references to stack variables present in global memory, unless they are *escaped*.

For an arithmetic operation '+', its abstract operation is denoted as '$\oplus$'. The result of abstract value $a + b$ depends on their forms. If a is of form $X + [l_1, h_1]$ and b of form $[l_2, h_2]$, then the result is $X + [l_1 + l_2, h_1 + h_2]$. The result is less precise if both bases present. To add $X + [l_1, h_1]$ and $Y + [l_2, h_2]$, if neither X nor Y is $BaseSP$, then the result is *const*, and is $\top$ otherwise. Note that if an operation causes the number of abstract values for a location to exceed k, a generalization is used to reduce the count.

To handle function invocations, a *function summary* is computed for the callee. Note that there is only one summary for each function, regardless of its calling context. Depending on the analysis application, a function summary may consist of the following information:

- *Stack pointer change.* The stack pointer may change due to invocation of a function. Although for most of the cases, the value of `esp` change is

0, but sometimes it could also be a constant or $\top$.

- *Activation record range.* The stack memory region accessed as a result of function invocation is denoted as $BaseSP+[l,h]$. $BaseSP$ is the original stack pointer value upon the function entry, while $l, h \in (-\infty, \infty)$.

- *Change of registers and stack location values.* Since ASA captures register and stack location values with respect to their initial value, the changes are explicit with the end states of a function.

Given these summaries, ASA uses a function *applysum* to update the abstract store to reflect value changes of registers and stack locations specified in the summary. If the summary indicates a location l is unmodified or changed to one of values $x_1, x_2, ..., x_n$, then *applysum* assigns l with new abstract value $A[l] \cup \{x_1, x_2, ..., x_n\}$.

To analyze a function, ASA uses an iterative algorithm to traverse the CFG and perform abstract interpretation. For branches, each target basic block is handled. For merges, ASA takes the union of abstract stores for all incoming edges as the new state. When an *indirect* call is encountered, one option is to assume that ABI is followed. Therefore, a special *applysum* is used: callee-save registers as well as `esp` are preserved, but scratch registers are clobbered.

4 Accurate Recovery of Function Returns

Although function returns may seem trivial to identify, they are actually not. In this section, we present a technique for accurate return discovery. The most important motivation for this work is to apply it for ROP defense. We therefore illustrate how accurate inference and enforcement of returns can be useful.

4.1 Motivation and Approach Overview

As discussed, ROP is the most powerful and versatile among code reuse attacks. Its power stems from the pervasiveness of returns in binary code. As a result, there are sufficient gadgets in a reasonably large binary to perform Turing-complete computation. Although variants such as COP and JOP have been proposed, ROP remains by far the most dominant code reuse attack, and the only kind used repeatedly in real-world attacks. For this reason, we focuse on ROP attacks in this section, and develop a principled approach for defeating them.

ROP relies on repeated subversion of returns in the victim program. Since CFI is a general technique for limiting all control-flow subversions, it can be used as a defense mechanism. CFI defeats most ROP attacks since they tend to violate the statically computed CFG. However, determined attackers can overcome CFI [47, 32] — specifically, coarse-grained CFI that is based on simple static analyses can be defeated. In fact, researchers have shown that a Turing-complete set of gadgets is available on sufficiently large applications even when coarse-grained CFI is enforced [32].

We note that although techniques for more precisely constraining returns have been known for well over a decade [24, 25], they have not seen wide deployment due to compatibility and performance concerns. Next we discuss these in more detail.

The essential characteristic of ROP is the repeated use of return instructions. Thus, techniques for constraining returns can be very effective in defeating ROP attacks. The primary approach for confining returns is the *shadow stack,* which relies on a second stack that maintains a duplicate copy of every return address. Each call instruction is modified so that it stores a second copy of the return address on the shadow stack. Before each return, the return address on the top of the stack is compared with that atop the shadow stack. A mismatch is indicative of an attack, and program execution can be aborted before a successful control-flow hijack. However, previous shadow stack solutions suffer from one or more of the following drawbacks:

- *Incompleteness.* Many shadow stack schemes are based on compilers [24, 31]. They do not protect returns in hand-written assembly code from low-level libraries such as `glibc` and `ld.so` that are invariably present in every application. Also left unprotected are third-party libraries made available only in binary code form. Moreover, unintended returns (See Section 2.4.1) could be used in ROP, and these won't be checked against the shadow stack.

- *Incompatibility.* In most complex applications, returns don't always match calls. If these exceptional cases are not correctly handled, they lead to false positives that deter practical deployment of shadow stack approaches.

- *Lack of systematic protection from all ROP attacks.* None of the previous approaches provide a systematic analysis of possible hijacks of returns, and how these attempts are thwarted. Indeed, most previous approaches incorporate exceptions to the shadow stack policy in order to achieve compatibility. A resourceful adversary can exploit these policy exceptions to carry out successful ROP attacks.

In this section, we develop a new defense against ROP that overcomes these drawbacks. We provide an overview of our approach below.

Approach Overview Our approach is based on the following simple policy:

> *Return instructions should transfer control to* ***intended*** *return targets.*

With a static interpretation of "intention", many existing coarse-grained CFI schemes can be seen as enforcing this policy. However, as discussed before, a static interpretation affords far too many choices for return targets, allowing successful ROP attacks to be mounted. We therefore take a dynamic interpretation of intent. Specifically:

- The ability to return to a location is interpreted as a *one-time use capability.* These capabilities are inferred from and associated with specific parts of the program text, e.g., a call instruction, or, a move instruction that stores a function pointer on the stack, with the intent of using this pointer as the target of a return. A *return capability* is issued each time this program text is executed.

- These return capabilities must be used in a last-in-first-out (LIFO) order. As the term “capability” suggests, not every intended return needs to be taken. Unexercised returns arise naturally due to exception unwinding, thread exits, and so on. However, we require that those return capabilities that are exercised do follow a LIFO order.

The LIFO property of return capabilities means that they can be maintained on a stack, which we will refer to as the return capability stack (RCAP-stack). Our system therefore consists of two key components:

- *Static analysis to handle non-standard returns:* While the intended returns of call instructions are obvious, nontrivial applications include many non-standard returns that don’t match any calls. Unlike previous approaches that relied on manual annotations to handle them, our automated static analysis technique identifies (a) non-standard returns, and (b) the intended targets of these returns, which can be used for enforcement.

- *Enforcement of inferred backward edges:* Our instrumentation based enforcement mediates all returns using the inferred backward edges. The need for “whitelisting” return instructions is avoided, and therefore our policy is strict.

4.2 Background and Threat Model

4.2.1 CFI Platform

Although CFI itself cannot sufficiently confine returns, it can be used as an important primitive for *secure* static instrumentation, as it can limit control flows so that instrumentation cannot be bypassed [6]. Specifically, control flows cannot go to (a) middle of instructions, or (b) an instruction within (or immediately following) an inserted instrumentation snippet. For this reason, we build our defense, which is based on static binary instrumentation, on a platform that already implements CFI, specifically, the P𝕊I platform [105].

4.2.2 Threat Model

We assume a powerful remote attacker that can exploit memory vulnerabilities to read or write arbitrary memory locations, subject to OS-level permission settings on memory pages. We assume the attacker has no local program execution privilege or physical access to the victim system.

We assume DEP is enabled on the victim system and therefore ROP is a necessity for payload construction. We also assume that ASLR is deployed, but attackers can use memory corruption vulnerabilities to leak the information needed to bypass it without resorting to brute-force.

4.3 Inferring Intended Control Flow

As discussed earlier, we focus exclusively on return instructions. We do not attempt to further improve the (coarse-grained) BinCFI policy [106] enforced on the remaining branch types by our implementation platform, namely, P𝕊I [105].

The first task in enforcing a stronger policy on returns is to precisely infer program-intended control flow for each of them. We develop a static analysis for this purpose. Specifically, our analysis identifies instructions that push addresses that may later be used as the target for a return instruction. As a fallback option, static analysis may be augmented with manual annotations, but we have not had to do this so far.

Based on the results of static analysis and/or annotations, instrumentation is added to update RCAP-stack to keep track of the return capabilities acquired by the program by virtue of executing these instructions.

4.3.1 Calls

Most call instructions are used for function invocations and therefore express an intent to return to the next instruction. However, it is up to the callee to decide whether the return is actually exercised. For example, a call to `exit()` will never return. Moreover, the call instructions themselves may be used for purposes other than calling functions. For instance, position-independent code (PIC) on x86 uses call instructions to get the current program counter, from which the base of the static data section is computed.

Unintended calls do not lead to compatibility problems since we do not require all return capabilities to be used. However, issuing unneeded capabilities can increase an attacker's options. To avoid this, we use a static analysis of target code to determine if the return address generated by a call is definitely discarded before being used by a return instructions. In the simplest case, a discard will happen through the use of a `pop` instruction that pops off the return address at the top of the stack. More generally, the location containing the return address may be overwritten, or the stack pointer incremented to a value greater than this location. If one of these properties holds on every

```
0x146b4 mov %eax,(%esp)
0x146b7 ...
0x146bb ret $0xc
```

Figure 4.1: A non-standard return from `ld.so`

execution path starting at the target of the call, then we conclude that the return address will not be used as the target of a return.

After identifying unintended calls, the remaining calls are instrumented for storing the return address on RCAP-stack. For unintended calls, the RCAP-stack is left unchanged.

4.3.2 Returns

Returning to a code location is permitted only if the program possesses the capability to do so. We check RCAP-stack for this capability. Typically, this capability originates at the most recent call instruction, but there are instances where the return address is pushed by other means. We call such returns as *non-standard returns.*

One example of a non-standard return is shown in Figure 4.1. The return instruction (at `0x146bb`) uses a return address generated by a `mov` instruction (at `0x146b4`) rather than a call. This code snippet is taken from GNU's dynamic loader, and the non-standard return is used for dynamic function dispatch after resolving a symbol. Specifically, when a function from another module is called for the very first time, its execution traps to the dynamic loader for symbol resolution. After the loader has resolved the address for the function and cached the result in original module's Global Offset Table (GOT), control should be directed to the called function. This is achieved by first moving the function address (stored in `eax` register) to the top of stack using a `mov`, and then issuing a return[1], as shown in Figure 4.1.

Note that while this non-standard return achieves the effect of an indirect jump, it does so without using any register (other than the stack pointer), and moreover, deallocates the memory location used to store the target address.

As discussed in the next section, there are a number of such non-standard returns, scattered in different modules. Moreover, unlike a call, whose intended return address is its successor, the intended target of non-standard return is not immediately obvious. These factors motivate the static analysis described below.

[1] The argument `0xc` to the `ret` instruction specifies the number of additional bytes that should be popped off the stack.

4.3.3 Static Analysis of Non-standard Returns

The distinction between a standard and non-standard return is the return address being used. The return address used by a *standard return* is pushed by the call instruction in its caller, and not modified in the callee. In contrast, return address used by a *non-standard return* is written to the return address stack slot by a non-call instruction. Based on this observation, we develop a static analysis that consists of four main steps as discussed below.

Candidate snippet extraction After a binary module is disassembled, we build its CFG. We then perform a backward scan on the CFG starting from each return instruction, and going back by n instructions, with $n = 30$ in our implementation. These snippets are our candidates for analysis.

Each such snippet may contain multiple execution paths to the return instruction. We analyze each path separately, as this enables more accurate analyis. In particular, this approach avoids approximations that result from least upper bound operations needed to handle path merges. However, this approach introduces two problems. First, loops can lead to an unbounded number of paths. We only consider paths corresponding to zero and one iteration of such loops. As a result, we may fail to discover some instances where an instruction inside a loop pushes a return address on the stack. In theory, this could lead to a compatibility problem, but in practice, it is very unlikely that such instructions occur within a loop body. The second difficulty is that it is theoretically possible for a single instruction I to participate in two distinct paths such that in the first path, I pushes a value on the stack that would be used by the return instruction at the end of the snippet, while it does not do so in the second path. Note that this (unlikely) scenario does not lead to an incompatibility: if the second execution path were to be taken at runtime, the return capability pushed by I would simply not be used.

Semantic analysis The second step is to analyze the semantics of each snippet by performing an abstract interpretation using ASA (Section 3.3). At the beginning of each snippet, each register is assigned a corresponding initial symbolic value. The program state is updated based on the semantics of each executed instruction. At the end of each instruction, the abstract value of each register (or memory location) will consist of simple expressions consisting of constants and initial register values. Since we are analyzing each execution path separately, these expressions rarely involve approximations. Our analysis includes a simple procedure for maintaining these expressions in a canonical form, thereby enabling equivalent expressions to be recognized in most instances.

Non-standard return identification The next step is to identify non-standard returns. After semantic analysis, the value of stack pointer register before the return instruction can be determined by an expression. Since it is the pointer for the return address slot, if there is any memory write to that location, a non-standard return is identified.

Intended control-flow inference The last step of the analysis is to infer the intended control flow for the non-standard return. To that end, we need to first identify the non-call instruction that stores the values used by the return. We call such an instruction as an *RAstore*. Such an instruction can be identified from the contents of memory and registers computed by our static analysis after each instruction in the snippet.

In the following part of this section, we describe real-world non-standard return examples identified by our analysis.

Non-standard return examples Previous shadow stack solutions rely on manual identification and ad-hoc instrumentation to support non-standard returns [33, 105, 31]. However, manual approaches are not scalable, and/or can lead to false positives on large and complex software. Figure 4.2 illustrates some of the more prominent real-world non-standard returns identified by our static analysis. In this figure, upper case register names (e.g., EAX) denote initial symbolic values, while lower case ones (e.g., eax) denote the current contents of registers or memory. For easier illustration, each code snippet is simplified to only include the last basic block. We omit the effects on floating point registers and segment registers. Note that our analysis results do not change when the full code snippets are used and when effects to non-general purpose registers are captured.

The first example is the same one as shown in Figure 4.1. Our analysis indicates that the return address comes from `eax`. The analysis discovers the highlighted instruction as the one that pushes the return address.

The second example comes from `setcontext(3)` function of glibc. The single argument of `setcontext` is a pointer to `ucontext_t` structure, which is loaded to `eax` at the first instruction. Since the user context structure contains all saved register information, most of the snippet code performs the job of register restores. Particularly, the program counter placed at offset `0x4c` of `ucontext_t` was loaded to `ecx` at loction `0x3fa81`. And the push instruction

Code Snippet	Semantics Equations
;; #1 /lib/ld-2.15.so	eax = *(ESP+8)
0x146b0 popl %edx	edx = *ESP
0x146b1 movl (%esp),%ecx	ecx = *(ESP+4)
0x146b4 movl %eax,(%esp)	esp = ESP + 4
0x146b7 movl 0x4(%esp),%eax	*(ESP+4) = EAX
0x146bb ret $0xc	ra = *esp = *(ESP+4) = EAX
;; #2 /lib/i386-linux-gnu/libc.so.6	
0x3fa73 movl 0x4(%esp),%eax	eax = *(*(ESP+4)+64)
0x3fa77 movl 0x60(%eax),%ecx	edx = *(*(ESP+4)+56)
0x3fa7a fldenvl (%ecx)	ecx = *(*(ESP+4)+60)
0x3fa7c movl 0x18(%eax),%ecx	ebx = *(*(ESP+4)+52)
0x3fa7f movl %ecx,%fs	esi = *(*(ESP+4)+40)
0x3fa81 movl 0x4c(%eax),%ecx	edi = *(*(ESP+4)+36)
0x3fa84 movl 0x30(%eax),%esp	ebp = *(*(ESP+4)+44)
0x3fa87 pushl %ecx	esp = *(*(ESP+4)+48)-4
0x3fa88 movl 0x24(%eax),%edi	*(*(*(ESP+4)+48)-4)
0x3fa8b movl 0x28(%eax),%esi	= *(*(ESP+4)+76)
0x3fa8e movl 0x2c(%eax),%ebp	ra = *esp
0x3fa91 movl 0x34(%eax),%ebx	= *(*(*(ESP+4)+48)-4)
0x3fa94 movl 0x38(%eax),%edx	= *(*(ESP+4)+76)
0x3fa97 movl 0x3c(%eax),%ecx	
0x3fa9a movl 0x40(%eax),%eax	
0x3fa9d ret	
;; #3 /lib/i386-linux-gnu/libgcc_s.so.1	
0x154cb movl %esi,%ecx	eax = *(EBP-20)
0x154cd movl %edi, 0x4(%ebp,%esi,1)	edx = *(EBP-16)
0x154d1 addl $0x10,%esp	ecx = ESI+EBP+4
0x154d4 leal 0x4(%ebp,%ecx,1),%ecx	ebx = *(EBP-12)
0x154d8 movl -0x14(%ebp),%eax	esi = *(EBP-8)
0x154db movl -0x10(%ebp),%edx	edi = *(EBP-4)
0x154de movl -0xc(%ebp),%ebx	ebp = *EBP
0x154e1 movl -0x8(%ebp),%esi	esp = ESI+EBP+4
0x154e4 movl -0x4(%ebp),%edi	*(ESI+EBP+4) = EDI
0x154e7 movl 0x0(%ebp),%ebp	ra = *esp
0x154ea movl %ecx,%esp	= *(ESI+EBP+4)
0x154ec ret	= EDI
;; #4 /usr/lib/libunwind-setjmp.so	eax = EDX
0x674 pushl %eax	esp = ESP-4
0x675 movl %edx,%eax	*(ESP-4) = EAX
0x677 ret	ra = *esp = *(ESP-4) = EAX

Figure 4.2: Code snippets and their analysis results

at `0x3fa87` pushes it as return address onto stack, which is consumed by the return instruction at the end of the snippet. This non-standard return and the RAstore at `0x3fa87` are identified by our static analysis. Another similar case in function `swapcontext(3)` from the same module, was also identified (not shown in figure).

The third example is a snippet from one of the stack unwinding functions in libgcc_s.so.1. The code first stores `edi`, the address of landing pad (handler code) which is previously computed, to the return address slot of next frame (`0x154cd`). Therefore, the following return will redirect control to the landing pad. This example also demonstrates the power of the analysis: the store to `0x4(%ebp,%esi,1)` at `0x154cd` does not "look" like a return address overwrite, however our static analysis is able to detect it. This is also an example why simple pattern matching based non-standard return identification would not work well.

Our last example is from an unwinding library libunwind. The snippet is simple, and similar to the first example, but used for implementing longjmp.

We note that although the non-standard return compatibility problem has been recognized by many in the literature [33, 31], only the first and third of these four examples have seen manual handling [33]. In contrast, our static analysis systematically identifies all of them, and serves as a basis for automatic instrumentation.

4.3.4 Discussion

Since that our static analysis is local, it can fail to identify non-standard returns when the RAstore instruction is far away from the return. If this assumption were to be violated, we can address it by strengthening the analysis, or using manual annotations. As mentioned before, we have not had to do this so far in our implementation.

4.4 Enforcing Intended Control Flow

In this section, we describe our approach for enforcing intended control flow using static binary instrumentation. We also describe the protection of the RCAP-stack to ensure that the same mechanisms used to corrupt the main stack cannot corrupt the RCAP-stack.

4.4.1 Instrumentation-based Enforcement

Intended control flow enforcement is realized by instrumenting calls, RAstores and returns. Both calls[2] and RAstores are instrumented in the same manner: a copy of the address being stored on the main stack is also pushed on RCAP-stack.

Return instructions are instrumented to check the RCAP-stack for the corresponding capability. Note that due to normal program behaviors such as stack unwinding, the required return capability may not always be located at the top of RCAP-stack. Similar to previous shadow stack proposals, our design also pops non-matching capabilities from the top of RCAP-stack until a capability that matches the target location of the return is encountered. If such a capability is never found, then a policy violation is reported and program execution aborted.

4.4.2 RCAP-stack Protection

Since return capabilities are generated and consumed for control flow authentication, their integrity needs to be ensured. In other words, RCAP-stack which stores return capabilities should be protected. Otherwise, determined attackers could use vulnerabilities to corrupt both the program stack and RCAP-stack for control flow subversion.

We used the same approach as described in CFCI [107], which has also been implemented on our platform P𝕊I. In short, the protection mechanisms are architecture-dependent. For x86-32, we rely on segmentation for efficient protection, and for x86-64, a randomization based approach is used. The randomization approach ensures that the location of RCAP-stack cannot be leaked.

4.5 Implementation

4.5.1 Static Analysis

The first step of static analysis is to extract candidate snippets. We utilized P𝕊I [105] for this purpose. Specifically, P𝕊I has a disassembly engine that is based on objdump, and adds a layer of error detection and correction over it. It also builds a CFG for the code disassembled. We traversed the CFG backwards from each return instruction to collect code snippets that were 30 instructions long.

[2] As discussed earlier, we avoid instrumenting calls that are determined never to return.

For our static analysis, we need to accurately model the semantics of each instruction. Specifically, we utilized a tool by Hasabnis et al [53, 55] that lifts assembly to GCC's intermediate language called RTL. Since RTL is a tree-structured language, it is further transform into a simple flat IR, as described in Section 3.1.2.

Our static analysis is performed on the simple IR statements. Since we analyze single execution paths, the main step in the static analysis is to substitute each register or memory location by the expression representing its previously computed value. This expression is maintained in a canonical form by defining an ordering on variables, and by performing constant-folding and other arithmetic simplifications.

4.5.2 Binary Rewriting based Enforcement

Our shadow stack instrumentation is based on P𝕊I [105] and was implemented as a plugin. We chose P𝕊I primarily for two reasons. First, shadow stack needs to be built on top of CFI to be effective against ROP attacks, and P𝕊I offers CFI as a primitive. Second, P𝕊I is a platform for COTS binary instrumentations and works on both executables and shared libraries, and therefore aligns with our goal of instrumentation completeness.

Protecting the Dynamic Loader Since the dynamic loader `ld.so` is an implicit dependency for all dynamically linked executables, it is also instrumented to prevent returns from being misused. We ensured that memory protection for RCAP-stack is set up before it is used by instrumentation.

Signal Handling The static analysis discussed in Section 4.3.3 is able to identify non-standard returns that consume return addresses stored by program code. However, return addresses can sometimes originate from the operating system. This is the case for UNIX signals. Once the OS delivers a signal to a process, it invokes the registered signal handler by switching context so that the user space execution starts at the first instruction of the signal handler. Prior to that, the OS puts the address of the sigreturn trampoline on the stack, which is to be used as the return address for the signal handler. Therefore, signal handler will "return" to the sigreturn trampoline, whose purpose is to trap back to the kernel. The kernel can proceed and revert user program execution with saved context. Since the returns for signal handlers (which are just normal functions) are also instrumented, if the corresponding return capabilities are not pushed onto RCAP-stack, signal delivery would cause false positives.

Fortunately, P𝕊I [105] already has a mechanism for signal handler mediation. The platform intercepts all signal handler registrations (using `signal` and `sigaction` system calls) and registers wrappers for the signal handlers. Once a wrapper function is invoked by the OS, it transfers control to the real signal handler after resolving its address. We use an updated version of wrapper code so that it pushes the corresponding return capabilities to RCAP-stack. (The wrapper code is not instrumented, and the CFI policy configured to ensure that it cannot be invoked by the application.)

4.5.3 Optimizing Returns

Our shadow stack is built on top of a binary instrumentation system that requires code pointer translation. In particular, code pointers point to original code section, while the instrumented code resides in a different section. As a result, code pointer values need to be translated to the corresponding code locations in the instrumented code. This step, called address translation, is a significant source of runtime overhead because it requires a hash table lookup. To improve the performance, we performed an optimization that has also been used in some previous research works [76]: push both the original address and translated address on the shadow stack for each call. At the time of return, we first compare the return address on the main stack with the original address on the shadow stack, and if they match, return to the translated address on the shadow stack.

For calls, the translated return address is simply the address of the instruction following the call instruction. However, RAstores push code pointers on the stack, so there is no way to avoid address translation for them. Rather than eagerly performing address translation at the RAstore, we simply push a null value as the translated address. At a return instruction, if the translated address has a null value, we perform address translation at that point.

4.6 Evaluation

We evaluated the key aspects of our system using a wide range of software on Linux and FreeBSD operating systems. Below, we present our findings and results.

4.6.1 Compatibility

In this section, we evaluate the compatibility improvement offered by our approach. We first present statistics on the identification of non-standard

Directory	Linux NSR #	Linux NSR module #	FreeBSD NSR #	FreeBSD NSR module #
/lib	9	4	7	2
/usr/lib	41	23	0	0
/bin	6	1	7	2
/sbin	6	1	4	1
/usr/bin	26	7	0	0
/rescue	N/A	N/A	182	91
/opt	28	7	N/A	N/A
total	116	42	213	98

Figure 4.3: Non-standard return (NSR) statistics

Module	OS	NSR Count
/lib/ld-2.15.so	Linux	2
/lib/i386-linux-gnu/libc.so.6	Linux	2
/lib/i386-linux-gnu/libgcc_s.so.1	Linux	4
/usr/bin/cpp-4.8	Linux	4
/usr/bin/g++-4.8	Linux	4
/usr/bin/gcc-4.8	Linux	4
/lib/libc.so.7	FreeBSD	3
/lib/libgcc_s.so.1	FreeBSD	4
/usr/bin/clang	FreeBSD	5

Figure 4.4: Non-standard returns in common modules

returns, together with an explanation for their prevalence. We then demonstrate the improved compatibility by testing our instrumentation on low-level and real-world software.

Non-standard Return Statistics We ran our static analysis tool on executables and shared libraries from an Ubuntu 12.04 32 bit Linux desktop distribution, and a FreeBSD 10.1 32 bit desktop distribution. We have identified hundreds of non-standard return instances from different modules. Figure 4.3 shows the number of non-standard return instances and the modules containing them for different directories of Linux and FreeBSD.

To better understand the impact of non-standard returns to shadow stack compatibility, we need to further zoom in and see if they exist in widely used binary modules. Figure 4.4 shows the prevalence of non-standard returns in some of the widely used modules.

Non-standard Return Summary In this section, we summarize some of the most common reasons for the prevalence of non-standard returns, based on an analysis of our static analysis results.

1. Programming language design and implementation

   In addition to subroutine abstraction, return instructions can also be used to implement other control flow abstractions such as coroutines or light-weight threads. Under these situations, they are used to transfer control between contexts, and therefore do not match calls.

2. Operating system design and implementation

   Operating systems also provide programmers various abstractions to ease their job. These abstractions may use return to implement control flow behavior across OS boundary. UNIX signals, as discussed, are probably the most prominent example in this category.

3. Optimization "tricks"

   In the engineering of some software constructs, programmers tend to make "clever" uses of assembly instructions. This also happens to return instructions.

Testing Low-level and Real-world Software In order to further evaluate the compatibility of our approach, we tested it with some low-level libraries and real-world software. For each binary module tested, we first ran our static analysis to identify non-standard returns and RAstore instructions. The results are then fed into our instrumentation module to generate hardened binaries. The instrumented software is finally executed for testing. For multi-threaded programs used in this evaluation, we used Pin [65] for our testing.

Figure 4.5 shows the low-level and real-world software we have tested, and how we tested them. The "Size" column specifies the total mapped code size (in MB) of all modules of the program. No incompatibilities were found on any of these programs, demonstrating that our approach works well even on low-level software. The total size of all software tested in this evaluation is almost 200MB.

4.6.2 Protection

Security Analysis Our system instruments all software modules including executables, shared libraries, and dynamic loader. Moreover, it protects all backward edges including both standard and non-standard returns. Return

Type	Software	Size	Description
Low-level	libunwind	1.9	Run a test program unwinding its own stack based on libunwind API
Low-level	libtask	2.0	Run a tcp proxy that uses user level threads API provided by libtask
Real-world	scp	2.1	Copy 10 files to server
Real-world	python	6.7	Run pystone 1.1 benchmark
Real-world	latex	7.8	Compile 10 tex files to dvi
Real-world	vim	9.1	Edit text file, search, replace, save
Real-world	gedit	22	Edit text file, search, replace, save
Real-world	evince	26	View 10 pdf files
Real-world	mplayer	46	Play 10 mp3s
Real-world	wireshark	58	Capture packets for 10 min

Figure 4.5: Low-level and real-world software testing

capabilities greatly restrict the scope of attacks possible. A coarse-grained CFI permits any return to target any of the instructions following a call in a program. In contrast, our approach limits return to one of the return addresses that are already on the RCAP-stack. Moreover, each time an attack makes use of a return address other than the top entry on RCAP-stack, the intervening entries are popped off, thus further reducing the choice of possible targets for the next return.

Note that although JOP and COP gadgets can be used in advanced code-reuse attacks, the vast majority of them still rely on ROP gadgets [47, 32, 20, 48], and therefore can be defeated by our system.

Stack Pivoting In ROP attacks, controlling the stack is the most important goal of the attacker. This is because, (a) fake return addresses need to be prepared on stack so that control flow can be repeatedly redirected in the manner chosen by the attacker, and (b) the stack supplies the data used in ROP computation.

Attackers basically have two choices to control the stack. The first is to *corrupt* the stack, usually through a stack buffer overflow. The second is to *pivot* the stack, i.e., hijack the stack pointer to point to attacker controlled data. Among these two, stack pivoting is more versatile because vulnerabilities other than buffer overflow could be used. It is also more convenient because the entire stack could be controlled, without being limited by factors such as the location of the vulnerable buffer, or the maximum length of overflow.

Our system readily defeats ROP based on both stack corruption and stack pivoting. As the effectiveness for stack corruption is clear, we focus on the latter. Specifically, in a single RCAP-stack scheme, stack pivoting based ROP is blocked because the required return capabilities won't be present on RCAP-stack. When multiple RCAP-stacks are used, although stack pivoting could cause new RCAP-stack creation, this does not compromise security as the new RCAP-stack starts out with zero return capabilities on it.

Note that RCAP-stack protection is critical for defeating stack pivoting. This is because in addition to stack pivoting, the attacker could also craft and pivot an RCAP-stack by corrupting the RCAP-stack pointer. While previous solutions may be vulnerable to such attacks [33, 31], our system is resistant because RCAP-stack pointer resides in protectd memory as well.

TOCTTOU Threats For standard returns, our instrumentation pushes return capability onto RCAP-stack at the time of a call, i.e., the instant that return capability is issued. This is different from schemes that push return capability at function prologue [24, 31], and hence provide a (narrow) window for TOCTTOU attacks.

However, we note that our instrumentation does have a delay to store return capability in the case of a non-standard return: i.e., it happens at RAstore instruction, rather than return address generation instruction. This is due to limited data flow tracking of our analysis, and is not an issue when annotation is possible.

Storing return capability at a later time may give some window for attackers, because they can modify the generated return capability before its store on both stacks. However, attacker capabilities for utilizing non-standard returns is greatly limited because of the following two reasons. First, CFI is still enforced as our base policy. Even if return capabilities for non-standard returns can be altered by attackers, it has to satisfy CFI at least, and therefore the forged capability can only grant transfer to instructions after calls. Second, as shown in Section 4.6.1, there are limited number of non-standard returns. Repeatedly corrupting return capabilities before store, effectively chaining such limited gadgets and bypassing CFI would be very difficult.

Experimental Evaluation of ROP Defense We evaluated the effectiveness of our approach using two real-world ROP attacks. Our first test was the ROPEME attack [64], which exploits a buffer overflow vulnerability in a test program. The attack is two-staged. In the first stage, the attack uses a limited set of gadgets in non-randomized executable code to leak out the base address of `libc`. This enables the attacker to bypass ASLR as it relates to targeting gadgets in `libc`. In the second stage of the attack, ROPEME uses a payload

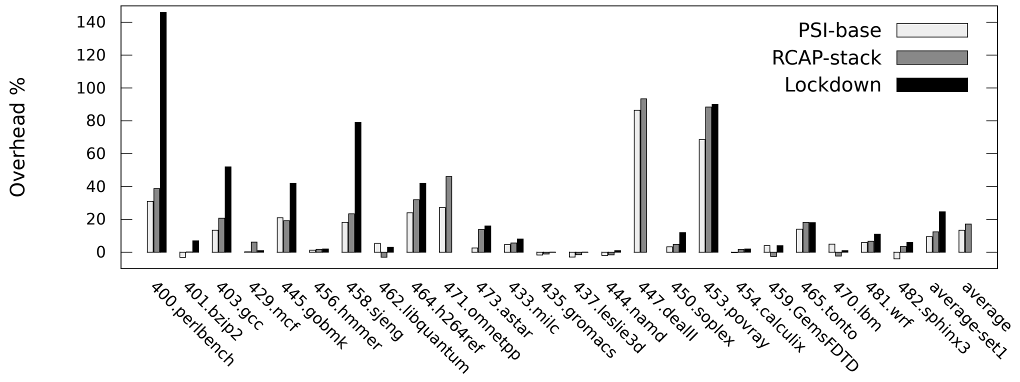


Figure 4.6: CPU overhead of shadow stack systems on SPEC 2006

consisting of a chain of `libc` gadgets. The stack is pivoted to this payload, and control is transferred to `libc` gadgets. Our defense blocked the attack at the first stage, because a backward edge control flow violation was identified when the vulnerable function returned with an overwritten return address.

Our second test was to protect a vulnerable Linux hex editor: HT Editor 2.0.20. A specially crafted long input could overflow the stack and lead to ROP attack [3]. As with the first attack, we detected the very first control flow violation and successfully defeated this ROP attack as well.

4.6.3 Performance Overhead

We have measured the CPU overhead of our instrumentation on SPEC 2006 benchmark. We tested on a x86-32 Linux machine because it is the only environment currently supported by PSI [105]. For all the benchmarked programs, we transformed all involved executables and shared libraries. The results are presented in Figure 4.6, where an empty bar in the histogram indicates an unavailable performance number.

We compare our performance with that of our base platform PSI [105] and Lockdown [75], a recent dynamic instrumentation based shadow stack implementation. From Figure 4.6, we can see that the performance overhead of our system is about 17% on average. Our optimization (Section 4.5.3) accelerates several control-flow intensive benchmarks such as 429.mcf and 447.dealll by 9% and 403.gcc, 458.sjeng, 471.omnetpp, and 453.povray by 5%. For the common set of programs we had with Lockdown, our overhead is 13% while theirs is about 24%.

Parallel shadow stack [31] achieves lower overhead by employing a variety of optimizations. They report overheads in the range of 3.7% to 4.6%. Their approach does not operate on binaries, but instead, on the assembly code produced by a compiler. As a result, they avoid the overhead of address translation. In addition, they do not enforce CFI. Considering these are the two major source of overhead for the P𝕊I platform we used, our added overhead of 4% makes our performance comparable to theirs.

5 Accurate Recovery of Function Boundaries

Functions are among the most common constructs in programming languages. While their definitions and declarations are explicit in source code, at the binary level, much information has been lost during the compilation process. Nevertheless, numerous binary analysis and transformation techniques require function information. For reverse engineering tasks such as decompiling [49, 38, 84], function boundary extraction provides the basis for recovering other high level constructs such as function parameters or local variables. In addition, many binary analysis and instrumentation tools are designed to operate on functions. These include binary code search [40, 36, 34, 21], binary code reuse [96], security policy enforcement [25, 82, 22, 93, 94], type inference [62], in-depth binary analysis such as vulnerability detection [90], and more. In fact, a recent survey performed literature study by collecting all binary-based papers published last 3 years at top security conferences, and found that 14 out of 30 works rely on function boundary information [11]. As a result, developers of most existing binary analysis platforms [2, 17, 50, 88] need to design and implement techniques to recognize functions.

Function recognition is a challenging task for stripped COTS binaries since they lack debug, relocation, or symbol information. Although directly called functions can be readily identified from disassembly (e.g., 42c9c0 in `call 42c9c0`), a significant number of functions are only reachable *indirectly*. Since resolving indirect call targets (e.g., all possible values for eax in `call eax`) is an undecidable problem, the fraction of indirectly reachable functions identified through static analysis is usually limited. Although there exist conservative techniques to identify a superset of possible functions [106], they lack precision since they overestimate function starts by a significant factor.

Compiler optimizations further exacerbate function recognition in COTS binaries. For instance, contrary to the high level abstraction that a function has a single entry point, a function in a binary may have multiple entries. Moreover, instead of being entered via a `call` instruction, tail call optimizations result in the use of `jumps` to enter a function. Tail calls are relatively common in optimized binaries, but with previous techniques, it is difficult to reliably distinguish them from normal jumps.

Due to the above difficulties, one cannot rely on the obvious approach of identifying functions by following direct calls. Many previous systems [2, 17, 50, 88] relied instead on pattern-matching function prologues (e.g., the instruction sequence `push ebp`; `mov esp`, `ebp`) and epilogues. Unfortunately, this approach is far from robust, since these patterns may differ across compilers. Moreover, optimizations may split and/or reorder these code sequences.

Other optimizations (e.g., reuse of `ebp` as a general purpose register) may also remove such identifiable prologues/epilogues. As a result, the best existing tools are still unsatisfactory for function recognition [15].

To overcome the limitations of manually identified patterns, machine-learning based approaches have been proposed for function recognition [81, 15, 87]. The idea is to use a set of binaries to train a model for recognizing function starts and ends. Machine learning can build more complete models that work across multiple compilers, while reducing manual effort. As a result, ByteWeight [15] achieved an average F1-score of 92.7% on a benchmark consisting of x86 binaries. Shin et al [87] further improved the accuracy to achieve an F1-score of 94.4% on the same dataset. Unfortunately, error rates of over 5% are still too high for most applications. More importantly, the accuracy of these techniques can be skewed by the choice of the training data. In fact, an independent evaluation of this dataset [13] found many functions to be duplicated across the training and testing sets, thus artificially increasing their F1-score. When evaluated with a different data set, ByteWeight's accuracy degraded to around 60% [13].

In light of these drawbacks of machine-learning based approaches, we propose a more conventional approach for function discovery, one that is founded on static analysis. However, unlike previous techniques that relied on simple control-flow analyses, and were confounded by the above-mentioned complications posed by stripped COTS binaries, our technique incorporates two key advances:

- We develop a *fine-grained analysis* that is based on detailed semantics of every instruction, including their effect on the contents of the registers and memory. As a result, our analysis can reason about the content of the stack, as well as the flow of data between a function and its caller.

- We identify a rich set of *data-flow* properties that characterize function interfaces, such as the use of registers and the stack to pass parameters and/or return values. We present a static analysis to discover these flows, and verify whether a candidate function satisifies these properties.

As a result of these advances *we have achieved a 4-fold reduction* in error rate as compared to the results reported by Shin et al [87]. As compared to Nucleus [13], which relies on a static analysis of control-flows, we achieve an even more impressive error rate decrease of more than 7x.

Contributions We develop a novel, static analysis based approach for function recognition in COTS binaries. Specifically, we make the following contributions:

- *Function identification by checking function interface properties.* We show that function *interface* properties, as compared to function prologue patterns, can provide valuable evidence for function recognition. We identify a collection of such properties and present static analysis techniques to check them. Each of these techniques is shown to be independently effective in our evaluation.

- *In-depth evaluation.* Our evaluation consists of about 2400 binaries resulting from 312 distinct C, C++ and Fortran programs. These binaries have been compiled using 3 different compilers (GCC, LLVM and Intel) for two architectures (x86 and x86-64) at four distinct optimization levels. In contrast with previous work, our evaluation set includes low-level code with hand-written assembly code, in particular, GNU libc.

- *Highly accuracy.* Our approach achieved an average F1-score of 99% across these data sets, much better than the 90% to 95% achieved by previous works [15, 87, 13]. This represents a reduction in error rate by more than 4x.

- *Deeper insight.* Our approach automatically categorizes recognized functions by their reachability such as "tail-called" or "unreachable." As discussed in Section 5.6, such information can be the basis for further tuning and refinement of the analysis in order to support demanding applications such as binary instrumentation that cannot tolerate errors.

5.1 Overview of Approach

5.1.1 Problem Definition

We define a *binary function* as a sequence of bytes in code section that has a single *entry point* to be reached from outside the function; and one or more *exit points* that transfer control from the function to some code outside. These bytes need not be physically contiguous. Note that the entry point is typically reached using *call* instructions, but possibly also *jump* instructions. An exit point may be a *return* instruction, a jump to the entry point of another binary function, or a call to a function that never returns.

Multiple-entry functions are supported: a function with n entry points is treated as if there are n independent single-entry functions. Each is analyzed independently by our method.

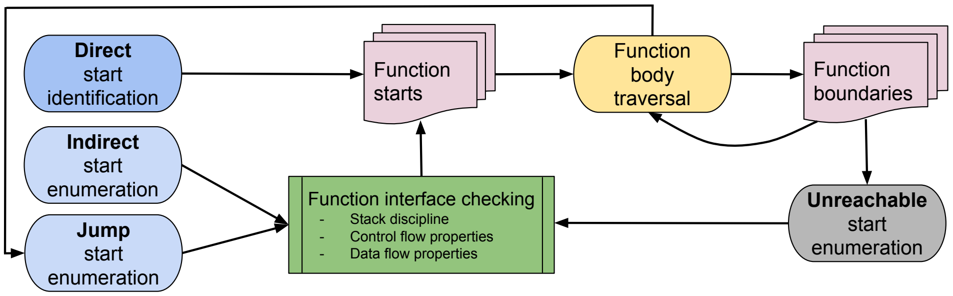

Direct start identification
Indirect start enumeration
Jump start enumeration
Function starts
Function interface checking
- Stack discipline
- Control flow properties
- Data flow properties
Function body traversal
Function boundaries
Unreachable start enumeration

Figure 5.1: Overview of our analysis. `Direct` function starts are identified from call instructions in the disassembly, and require no further confirmation. The remaining function start candidates (`Indirect`, `Jump` and `Unreachable` function) need to pass our function interface checks that eliminate spurious functions. Function body traversal is used to determine function ends, and takes advantage of already identified functions. Function body traversal and boundary information feeds back into the determination of unreachable functions, as well as jump-reached (i.e., tail-called) functions.

Our task is to recover bytes belonging to each function. Similar to prior work [15, 87], correctness is determined by matching the *start* and *end* address for each function with symbol table information[3]. Note that start and end addresses are determined by the smallest and largest address of all *bytes* of the function, respectively.

Scope Our analysis focuses on stripped COTS binaries: no debug or symbol information is available. Our evaluation focuses on Executable and Linkable Format (ELF) binaries from x86-32 and x86-64 Linux platform, although our technique itself is applicable to other platforms and binary formats, such as Windows and the Portable Executable (PE) format. We make no assumptions on the source language, compiler used, compiler switches or optimization levels. However, similar to prior work [15, 87], obfuscated binaries are out of scope.

5.1.2 Approach Overview

The key idea of our approach is that of *enumerating* possible function starts, and then using a static analysis to *confirm* them. The overview of our approach is shown in Figure 5.1.

[3] While our experiments are performed on stripped binaries, we rely on symbol tables in unstripped binaries for the ground truth.

Possible function start addresses are enumerated in different ways. Directly called (`Direct`) function starts are readily obtained from disassembly code. For indirectly reachable (`Indirect`) functions, code addresses buried in all binary sections serve as proper function start candidates, while for unreachable (`Unreachable`) functions, the beginning of unclaimed code regions are considered.

As shown in Figure 5.1, any function that isn't directly reached needs to be confirmed[4] using additional checks. Since functions interact with each other through *interfaces*, our approach identifies *spurious* functions by checking for properties associated with function interfaces, such as the stack discipline, control-flow properties and data-flow properties.

To determine function ends, function body traversal is performed. It takes advantage of already recognized functions, and serves as means for recognizing functions reached only using jumps (i.e., tail-called functions).

In a nutshell, our approach *iteratively* uncovers functions based on how they are reached. `Direct` functions are first identified, and then `Indirect` functions are enumerated and checked. Finally `Unreachable` functions are handled. Note that `Jump` function enumeration and checking happens alongside the body traversal for all other functions. The whole procedure ends when all code regions have been covered.

In the following sections, we describe our techniques for determining function starts (Section 5.2), function boundaries (Section 5.3), and interface checking (Section 5.4).

5.2 Function Starts

5.2.1 Directly Reachable Functions

According to our definition in Section 5.1.1, functions are code sequences that are *called* (or alternatively, reached using *jumps*). Therefore, with the disassembly obtained (Section 2.1), the targets for *direct* call instructions are *definite* function starts [5]. They are first collected.

Although we can obtain direct jump targets in the same manner, it is nontrivial to distinguish whether they are function starts (as in the case of a tail call), or, more likely, intra-procedural targets. We enumerate jump targets as possible function starts if the target is physically non-contiguous with current

[4] In principle, we could require this confirmation for directly called functions as well, but did not do so for performance reasons.

[5] We detect the obvious exceptions such as `call next`; `next: pop reg` as appear in position independent code to retrieve current instruction pointer.

function body (Section 5.3). An analysis of the *jump context* is later performed to confirm or reject the function start.

5.2.2 Indirectly Reachable Functions

As compared to directly reached functions, some functions are only reachable *indirectly*. These include functions that are reached using either indirect *calls*, or indirect *jumps* (i.e., indirect tail calls). To enumerate their starts, constant scanning described in Section 2.2 is used. Since spurious function starts may also be included, the constants need to be confirmed with interface checking.

5.2.3 Unreachable Functions

Other than directly and indirectly reachable functions, there are functions that are not reachable at all. For reasons discussed in the beginning of this chapter, we also try to identify unreachable functions. The basic idea is to analyze the "gap" area, i.e., code regions that are not covered by already identified functions. This procedure is performed *after* the determination of directly *and* indirectly reachable functions.

Because functions may have padding bytes after its end, we consider the first non-NOP instruction[6] in each gap as a potential function start. The corresponding function end is then determined using techniques described in Section 5.3. If this potential function does not take all the space of the current gap, the remaining region is considered as a new gap, and the process continues until all gaps have been analyzed.

Although our gap exploration seems similar to prior work [2, 50, 17], the primary difference is that the identified functions have to pass interface property checking.

5.3 Identifying Function Boundaries

To identify function boundaries, we traverse a function body, starting at its entry point. All possible paths are followed until control flow exits the function. The largest address of any instruction discovered using this process is considered the end of the function. Note that exits may sometimes take place via jumps (tail calls), or calls to non-returning functions. As described below, we discover and handle those cases as well.

[6] We consider an instruction as "NOP" based on its semantics: i.e., the machine state (other than program counter) is *not* changed. For example, other than `nop` itself, `xchg ax, ax` is also a NOP instruction.

Function body traversal works by following all intra-procedural branches until function exits. Specially, for conditional jumps, both branches are taken, while for table jumps, all recovered targets are followed.

For function body traversal, some special control flow transfers need to be taken into account, most notably C++ exception handling. When an exception occurs, either at the current function or some of its callees, control is first directed to C++ runtime, which is responsible for locating the proper handler code (also called a "landing pad"), and during this process, stack unwinding may be performed. If current function is identified to have a landing pad designated for the raised exception, control flow is transferred to this landing pad.

Logically, a landing pad is associated with (the main body of) a function. Indeed, at source code level, it corresponds to the `catch` block inside a function. Since a landing pad is essentially *indirectly* reached from C++ runtime, the control flow transfer is not captured in the disassembly of the analyzed binary. We therefore parse the "call frame information" available in `.eh_frame` sections of ELF binaries, the same metadata used by the C++ runtime to guide exception handling, to recover such control flows and consequently follow them in our function body traversal. Note that exception handling information must be present even in stripped binaries.

Function body traversal stops at function exits. While most functions exit using return instructions, there are special cases that involve calls and jumps. These special cases are described below.

Function exits via calls to non-returning functions. Although most functions do return to the caller, some don't. For example, libc `exit` function terminates the program, and the control flow never returns back to the caller. For invocation of such non-returning functions, function body traversal should not go past the call.

To determine non-returning functions, we perform a simple analysis. First, we collect a list of library functions that are documented to never return. We then analyze each potential function of the binary. If it calls a known non-returning function on each of its control flow paths, it is also recognized as a non-returning function and added to the list, and so on.

Note that function body traversal only stops at a *direct* call to a non-returning function. For an indirect call, the traversal falls through after the call, as the target is unknown and compiler has to be conservative.

Function exits using jump instructions. Tail call is another special type of function exit and is based on *jumps*. If not recognized, tail calls are

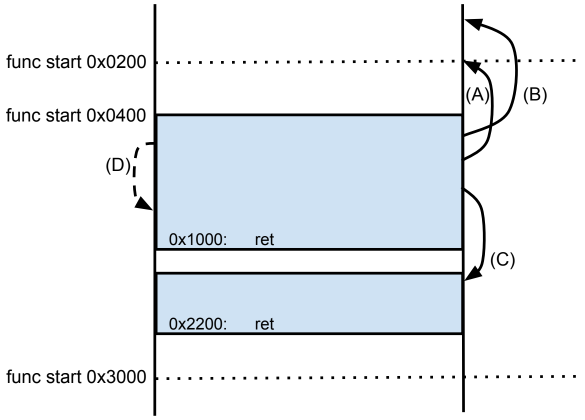


Figure 5.2: Tail call detection

treated as normal intra-procedural jumps, causing errors in identified function ends.

To detect tail calls, we utilize already identified functions to check jump targets, specifically:

1. If the jump target is a known function start or a procedure linkage table (PLT) entry[7], it is recognized as a tail call.

2. If the edge crosses known function boundaries, it is a tail call.

We illustrate with an example. In Figure 5.2, function body traversal begins from identified function start `0x0400`, and `A`, `B`, `C`, `D` are four direct jumps on the paths. Since jump `A` targets a known function and `B` crosses a function start, they are identified as tail calls. For jumps `C` and `D`, there does not seem to be an obvious way to determine if they target a different function, or are intra-procedural jumps. We use a two-step approach to resolve them.

First, our function body traversal *speculatively* follows all jumps and only terminates at definite function exits (e.g., returns). Second, all jumps whose target is not physically contiguous with other traversal-covered instructions are identified as potential tail calls. For example, the colored area in Figure 5.2 represents all traversal-covered instructions, and jump `C` is identified as a potential tail call, as its target is not preceded by any covered instruction. But on the other hand, `D` is not. These potential tail calls may be confirmed

[7] PLT entries are code stubs in ELF binaries that support dynamic linking. The target of a PLT entry is a function in a different binary module, which is patched by the dynamic loader at runtime. Similar mechanisms exist for Windows PE binaries.

with further checks: they are not immediately considered as definite function starts, as functions can be non-contiguous.

A tail call candidate is confirmed if both the potential function exit (the jump) and potential entry (the target) pass function interface checking. While function *entry* interface checking is to be introduced in the following section, we focus on function *exit* interface checking, which is based on stack discipline. Specifically, the stack pointer at the jump should not be lower than its initial value on function entry, as it indicates the local storage has not been deallocated and hence the jump is intra-procedural.

Note that other than determining current function exits, identification of tail calls serves a second purpose: enumerating `Jump` function starts. These functions may otherwise be undetected, if they are not directly or indirectly called.

5.4 Interface Property Checking

For potential functions that are reached only via jumps or indirect calls, we develop a set of static analysis techniques to check if the target is indeed a function. These checks can be divided into control flow and data flow properties, as described further below.

5.4.1 Control Flow Properties

According to the binary function definition in Section 5.1.1, control is transferred *to* and *out of* a function in well-defined forms. Therefore, if control flow reaches or leaves a "function" in a non-conformant manner, the function is identified as spurious.

Function entries. Our verification is based on a simple strategy: intra-procedural control flow targets are not function starts. Particularly, table jumps are intra-procedural control flow transfers as they result from `switch-case` statements. Table jump targets are excluded from function start candidates.

Note that our jump table analysis uses a conservative strategy for identifying jump tables and their bounds. In particular, it is designed to identify targets that are definite table jump targets, and hence it is safe to remove all of them from function start candidates.

Function exits. Our second control flow checking concerns function exits. Specifically, we focus on exits that use returns. Our check determines if the

```
805ce70 <get_date>:
805ce70:     push   %ebp  ; real func start
805ce71:     push   %edi
805ce72:     push   %esi
805ce73:     push   %ebx
...          ...
805d900:     pop    %ebx  ; +4 ; spurious func start
805d901:     pop    %esi  ; +8
805d902:     pop    %edi  ; +12
805d903:     pop    %ebp  ; +16
805d904:     ret
```

Figure 5.3: Incorrect return address is used for a spurious function

```
8081130 <find_connection_moves>:
...          ...
8081136:     sub    $0x533c,%esp
...          ...
8081a10:     mov    %edi,0x4(%esp) ; spurious func start
8081a14:     mov    $0x80e6718,(%esp)
8081a1b:     call   807c8d0
...          ...
```

Figure 5.4: "Return address" is overwritten for a spurious function

return instruction will control transfer to the address that was stored atop of the stack at the function entry point.

We note that although a static analysis may not give conclusive answer in cases like unbounded stack pointer change, this checking is still very effective in detecting spurious functions in general. While a more quantitative study is given in Section 5.5.5, we next illustrate with two examples.

The first example is presented in Figure 5.3. In the code snippet, address 0x805ce70 is a real function start and 0x805d900 is a spurious one. They are both enumerated as potential function starts. However, function [0x805d900, 0x805d904] is detected as spurious by our analysis, as its return address (used by return instruction at 0x805d904) is derived from a location 16 bytes higher than the proper stack slot.

Figure 5.4 shows our second example. In this code, address 0x8081a10 is enumerated as a potential function start. However, since its "return address" is overwritten at 0x801a14, the "function" can never return to the intended return address. As a result, our analysis correctly identifies 0x801a14 as a spurious function start. Note that in the context of the real function starting at 0x8081130, the purpose for instruction at 0x801a14 is actually preparing argument for the upcoming call. However, since the stack allocation at 0x8081136 was not included in the spurious function, the anomaly for "return address" usage is spotted.

Internal Instructions. A function's internal instructions should not be targeted by control flows from outside. An exception arises in the case of multi-entry functions, but even then, these alternate entry points must be targeted by *inter-procedural* control transfers. In contrast, if an internal instruction of a function f is targeted by an *intra-procedural* transfer from another function g, that provides strong evidence that f is likely spurious.

Recall that when we perform interface verification for a function beginning at location f, we start with a traversal of its body at f. The instructions uncovered by this traversal constitute the body of f, and any control transfers using intra-procedural control flow constructs (e.g., table jumps) from outside this body indicate that f is spurious.

5.4.2 Data Flow Properties

Similar to control flows, there are also data flows *into* and *out of* a particular function. This provides us further opportunities to check if a function is spurious.

Data flows between a function and the rest of the program are usually through registers, stack, or global memory. Compared to (stack or global) memory, the usage of registers for function interfaces is usually more constrained. In this section, we explore how register restrictions can help identify spurious functions.

Constraints on the use of registers for inter-procedural data flows are imposed by system ABI and calling conventions [41]. They are summarized in Figure 5.5. In this figure, allowed argument registers are registers that can be used for passing arguments to a function, therefore used for inwards data flow. On the other hand, callee-save registers are those registers whose value need to be saved before being used in a function, and restored before function returns, hence capturing both inwards and outwards data flows.

Note that on x86-32 allowed argument registers are calling convention specific, and the details are presented in Figure 5.6. To compute a reference set, we take the union of all calling conventions. Therefore, the resultant allowed argument registers are eax, edx, and ecx. If dataflow occurs from caller to callee via any other register, such flow is in violation of all calling conventions, thus indicating that the entry point is likely spurious.

Registers	x86-32 Windows	x86-32 UNIX-like	x86-64 Windows	x86-64 UNIX-like
Allowed argument	(See Figure 5.6)	(See Figure 5.6)	`rcx`, `rdx`, `r8`, `r9`	`rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
Callee-save	`ebx`, `esi`, `edi`, `ebp`	`ebx`, `esi`, `edi`, `ebp`	`rbx`, `rsi`, `rdi`, `rbp`, `r12-r15`	`rbx`, `rbp` `r12-r15`

Figure 5.5: Register usage summary for calling conventions on different platforms

x86-32 calling convention	Argument passing
cdecl, stdcall, pascal	stack
fastcall (Microsoft, GNU)	`ecx`, `edx` then stack
fastcall (Borland)	`eax`, `edx`, `ecx` then stack
thiscall (Microsoft)	`ecx` then stack

Figure 5.6: Arguments passing for x86-32 calling conventions

We perform static analysis which operates on each potential function to capture the *actual* data flow through registers and check with the reference (Figure 5.5). Any noncompliance suggests a spurious function.

Static Analysis for Argument Registers. To identify registers used for passing arguments to a "function", we perform a liveness analysis. We use the standard backwards data-flow analysis algorithm by computing GEN and $KILL$ sets for registers and `EFLAGS` (Section 3.2.2).

If a register is *live* at a "function" start, it is potentially an argument register. This is because a live register indicates its *use* is before its *definition* in the "function" body. Consequently, it must have been defined before the function being called and information is passed through it. However, there is an exception: callee-save instructions at function beginning "use" callee-save registers with the purpose of preserving them to stack. Since this does not represent information passing, they should not be considered as *real* uses. Our analysis adopts a simple strategy by not considering register saves on the stack as a use of the register. Specifically, if a register is saved to an address less than the value of the stack pointer at function entry, then it is *not* treated as a use of that register.

Note that a special case for argument register checking is that `EFLAGS` are not used for passing information. Therefore, a live flag also suggests a spurious function.

A concrete example for argument register checking is shown in Figure 5.4.

```
805ce70 <get_date>:
805ce70:     push  %ebp  ; real func start
805ce71:     push  %edi
805ce72:     push  %esi
805ce73:     push  %ebx
...          ...
805d900:     pop   %ebx  ; +4 ; spurious func start
805d901:     pop   %esi  ; +8
805d902:     pop   %edi  ; +12
805d903:     pop   %ebp  ; +16
805d904:     ret
```

Initial state	End state (for “function” [0x805d900, 0x805d904]	End state (for “function” [0x805ce70, 0x805d904]
ebx = EBX	ebx = *(ESP)	ebx = EBX
esi = ESI	esi = *(ESP + 4)	esi = ESI
edi = EDI	edi = *(ESP + 8)	edi = EDI
ebp = EBP	ebp = *(ESP + 12)	ebp = EBP
...	...	...
	ret_addr = *(ESP + 16)	ret_addr = *(ESP)

Figure 5.7: The analysis states of example code

For the spurious function starting at `0x8081a10`, `edi` is live and detected as an argument register. However, since `edi` is not allowed for that purpose, the function is spotted as spurious.

Static Analysis for Callee-saves. To check value preservation for callee-save registers, we keep track of register and memory values by performing an abstract stack analysis (Section 3.3). The analysis produces at the function end the abstract value of each register and memory location, and how it has changed against the initial value.

Figure 5.7 shows the initial and end states from our analysis of an example snippet. In this figure, the capitalized REG is the symbolic value denoting the initial value of `reg` upon function entry. The right two columns show the end states for “function” [0x805d900, 0x805d904] and [0x805ce70, 0x805d904], respectively. For “function” [0x805d900, 0x805d904], since registers `ebx`, `esi`, `edi`, `ebp` do not preserve their values but instead get new values from “return address” (location [ESP + 0]) and “stack argument” region (location [ESP + 4] to [ESP + 12]) , it is recognized as spurious. On the other hand, “function” [0x805ce70, 0x805d904] passes callee-save register usage checking. Note that in case register values end up with $\top$, our analysis conservatively concludes no violations.

Note that in above two examples (Figure 5.3 and Figure 5.4) the spurious functions violate *both* control flow and data flow properties. This is not always the case — sometimes only a single interface checking technique is effective. However, by adopting a comprehensive mechanism, we significantly increase the confidence for correctness once a function passes checking.

5.5 Evaluation

5.5.1 Data Set

We used three data sets for evaluating our system.

Data Set 1. The first data set is the same as that used by machine learning based approaches, namely, ByteWeight [15] and the work of Shin et al [87]. Since our current implementation is limited to Linux ELF binaries, the comparison focuses on the subset of the results for this platform. Note that the vast majority (2064 of 2200) of the binaries in this data set were on Linux, so our results do cover over 90% of the data used in these works. These 2064 binaries correspond to `binutils`, `coreutils` and `findutils`. They are compiled with GNU (`gcc`) or Intel (`icc`) compilers, from no optimization to the highest optimization level for both x86-32 and x86-64 architectures. These binaries include around 600,000 functions in total, with a total size over 280MB.

Despite its size, this data set is found to be skewed by a recent work [13]. Specifically, since many binaries are from the same package, they share a significant number of functions. This gives machine learning techniques advantages because functions used for learning and testing significantly overlap. Nevertheless, we use this data set in order to provide a direct comparison with machine learning approaches.

Data Set 2. Our second data set is the set of SPEC 2006 programs. As compared to the first data set, which are mostly operating system utilities written in C, SPEC programs are more diverse in terms of their applications, as well as the programming languages used (C, C++, Fortran). Moreover, unlike the first data set, SPEC programs rarely share functions with each other. To compile these programs, we used the GCC compiler suite (`gcc`, `g++`, and `gfortran`) and LLVM (`clang`, `clang++`), and compiled with all optimization levels (O0-O3).

Data Set 3. Our third data set is the GNU C library, which is a suite of functionally related shared libraries (24 in total), including `libc.so`, `libm.so`, `libpthread.so`, etc. This is a more challenging data set for two reasons. First, the code is low-level and contains many instances of hand-written assembly. Second, the binaries are in the form of position independent code (PIC). We used GCC -O2 to compile for both x86-32 and x86-64 architectures.

5.5.2 Metrics

We use the same metrics, *precision*, *recall*, and *F1-score* as in previous work [15, 87]. Their definitions are as follows. In these equations, TP denotes the number of true positives for identified functions, FP denotes false positive, while FN denotes false negatives.

$$Recall = \frac{TP}{TP + FN} \tag{1}$$

Note that recall captures the fraction of functions in the binary that are correctly identified by an approach.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

Typically, these two metrics are combined using a harmonic mean into a quantity called F1-score:

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall} \tag{3}$$

5.5.3 Implementation

Our main analysis framework is implemented in Python, and consists of about 4100 lines of code. For the disassembler, we used `objdump` and reimplemented the error correction algorithm from BinCFI [106]. The main framework also includes all major components described, including function start identification, function body traversal, and part of interface checking. Our current analysis engine is based on angr [88].

We used angr mainly because it is a comprehensive binary analysis platform, and supports both x86-32 and x86-64. We built our customized analysis on top of angr, but not using any of its built-in function recovery algorithms. In fact, their accuracy is well under the best published results from machine learning systems (which we compare in the following sections), probably because the primary goal of angr is for offensive binary analysis [88], rather than robust recovery of program constructs.

Tool	x86-32			x86-64			**overall**
	P	R	F1	P	R	F1	**F1**
ByteWeight	0.9841	0.9794	0.9817	0.9914	0.9847	0.9880	**0.9849**
Neural	0.9956	0.9906	0.9931	0.9880	0.9780	0.9830	**0.9881**
Ours	0.9978	0.9920	0.9948	0.9960	0.9948	0.9954	**0.9951**
Error ratio	*2.0000*	*1.1750*	*1.3269*	*2.1500*	*2.9423*	*2.6086*	**2.4286**

Figure 5.8: Function start identification results from different tools

Tool	x86-32			x86-64			**overall**
	P	R	F1	P	R	F1	**F1**
ByteWeight	0.9278	0.9229	0.9253	0.9322	0.9252	0.9287	**0.9270**
Neural	0.9775	0.9534	0.9653	0.9485	0.8991	0.9232	**0.9443**
Ours	0.9865	0.9809	0.9837	0.9912	0.9900	0.9906	**0.9871**
Error ratio	*1.6667*	*2.4398*	*2.1288*	*5.8523*	*7.4800*	*7.5851*	**4.3178**

Figure 5.9: Function boundary identification results from different tools

5.5.4 Summary of Results

Figure 5.8 and Figure 5.9 summarize function *start* and *boundary* identification results for the first data set, respectively. Since the two most recent work [15, 87] has the best published results and outperformed previous tools (such as IDA and Dyninst [50]), we only compare our results with them. Because we tested with the same data set, we directly use the numbers reported by them. In this figure (and following ones), "P" denotes precision, while "R" denotes recall. Note that for each architecture, every number (for P/R/F1) is a mean over the corresponding 1032 binaries. To enable direct comparison with machine learning systems, we followed their practice by using an arithmetic mean[8]. The "error ratio" in the figure is defined as $(1-MAX(ByteWeight, Neural))/(1-Ours)$ for each column. The overall error ratio for function start and function boundary is 2.43 and 4.32. Therefore, our system produced significantly better results.

The results for our second and third data sets are presented in Fig. 5.10. We compare with the most recent work in this field, Nucleus [13], which is also based on static analysis. We followed their way of summarizing results: using an average of geometric means for all optimization levels. Our F1 scores for this data set are consistently above 0.99, significantly higher than those of

[8] When using geometric or harmonic mean to summarize our results, the difference is less than 0.0006.

Dataset & compiler	Tool	x86-32 binaries			x86-64 binaries		
		P	R	F1	P	R	F1
SPEC (GCC)	Nucleus	0.97	0.89	0.92	0.97	0.90	0.93
	Ours	0.9988	0.9869	0.9927	0.9952	0.9861	0.9905
SPEC (LLVM)	Nucleus	0.95	0.88	0.91	0.94	0.86	0.90
	Ours	0.9978	0.9933	0.9955	0.9934	0.9902	0.9918
GLIBC (GCC)	Nucleus	-	-	-	-	-	-
	Ours	0.9846	0.9914	0.9879	0.9804	0.9840	0.9817

Figure 5.10: Function boundary identification results for SPEC 2006 and GLIBC.

Nucleus (around 0.92). We omitted zooming into each optimization level since our results are not sensitive to them: the F1 score differences are within 0.01.

As shown in the last two lines of the figure, *ours is the first work that evaluates with GLIBC.* Despite the challenges posed by PIC-code, hand-written assembly and other low-level features, our techniques achieve an F1-score above 0.98.

5.5.5 Detailed Evaluation

In this section, we present detailed evaluation for our second data set: SPEC 2006 programs. For space reasons, we focus on the most widely used optimization level: -O2. As shown in Fig. 5.11 and Fig. 5.12, for a wide range of applications written in three different languages (C, C++ and Fortran) and compiled with two compilers (GCC and LLVM[9]), our overall F1-scores are no lower than 0.9817 for function boundaries. Note that overall metrics are computed by using the aggregated true positives, false positives and false negatives over the selected fraction of binaries. For many individual binaries, we have achieved perfect (1.0000) precision and recall.

Distribution of Different Call Types. To understand how each step of analyses contributes to the finally identified functions, we list the corresponding results for SPEC 2006 programs in Figure 5.13. To conserve space, only GCC (-O2) compiled programs for x86-32 architecture are shown. Note that x86-64 binaries have similar results to their x86-32 counterparts.

As shown in the figure, the percentage of each function category is largely language and program dependent. For most C and Fortran programs, direct calls contribute to the largest number of identified functions. (Note that this

[9] LLVM does not have an official frontend for Fortran.

Program	Language	Suite	x86-32			x86-64		
			P	R	F1	P	R	F1
400.perlben.	C	int	0.9971	0.9983	0.9977	0.9743	0.9954	0.9847
401.bzip2	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.000
403.gcc	C	int	0.9959	0.9893	0.9926	0.9935	0.9946	0.9941
429.mcf	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
445.gobmk	C	int	0.9996	0.9996	0.9996	0.9980	0.9996	0.9988
456.hmmer	C	int	1.0000	0.9980	0.9990	0.9941	1.0000	0.9970
458.sjeng	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
462.libquan.	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
464.h264ref	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
433.milc	C	fp	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
470.lbm	C	fp	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
482.sphinx3	C	fp	1.0000	1.0000	1.0000	0.9824	0.9911	0.9867
471.omnet.	C++	int	0.9990	0.9946	0.9968	0.9975	0.9853	0.9914
473.astar	C++	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
483.xalan.	C++	int	0.9911	0.9923	0.9916	0.9883	0.9911	0.9897
444.namd	C++	fp	1.0000	1.0000	1.0000	1.0000	0.9904	0.9951
447.dealII	C++	fp	0.9601	0.9577	0.9592	0.9698	0.9494	0.9595
450.soplex	C++	fp	1.0000	0.9764	0.9881	1.0000	0.9443	0.9713
453.povray	C++	fp	0.9974	0.9707	0.9839	0.9913	0.9696	0.9802
410.bwaves	F	fp	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
416.gamess	F	fp	0.9931	0.9951	0.9941	0.9931	0.9979	0.9955
434.zeusmp	F	fp	1.0000	0.9884	0.9941	0.9767	0.9882	0.9824
435.gromacs	F	fp	0.9991	0.9981	0.9986	0.9982	0.9982	0.9982
436.cactus.	F	fp	1.0000	1.0000	1.0000	1.0000	0.9977	0.9988
437.leslie3d	F	fp	1.0000	1.0000	1.0000	0.9667	0.9355	0.9509
454.calculix	F	fp	0.9940	0.9836	0.9887	0.9953	0.9514	0.9728
459.Gems.	F	fp	0.9737	0.9487	0.9610	0.9733	0.9481	0.9605
465.tonto	F	fp	0.9760	0.9598	0.9678	0.9634	0.9634	0.9634
481.wrf	F	fp	0.9961	0.9958	0.9960	0.9972	0.9955	0.9963
C overall	C	both	0.9977	0.9950	0.9963	0.9918	0.9966	0.9942
C++ overall	C++	both	0.9839	0.9808	0.9823	0.9845	0.9757	0.9801
F overall	F	fp	0.9902	0.9846	0.9874	0.9866	0.9826	0.9846
Overall	all	both	**0.9886**	**0.9849**	**0.9868**	**0.9852**	**0.9781**	**0.9817**

Figure 5.11: SPEC 2006 results (GCC compiled binaries)

Program	Language	Suite	x86-32			x86-64		
			P	R	F1	P	R	F1
400.perlben.	C	int	0.9952	0.9940	0.9945	0.9898	0.9886	0.9892
401.bzip2	C	int	1.0000	1.0000	1.0000	0.9867	0.9736	0.9801
403.gcc	C	int	0.9977	0.9982	0.9979	0.9947	0.9942	0.9944
429.mcf	C	int	1.0000	1.0000	1.0000	0.9143	0.9697	0.9411
445.gobmk	C	int	0.9996	1.0000	0.9998	0.9996	0.9992	0.9994
456.hmmer	C	int	1.0000	1.0000	1.0000	0.9958	0.9916	0.9937
458.sjeng	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
462.libquan.	C	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
464.h264ref	C	int	1.0000	0.9981	0.9991	1.0000	0.9981	0.9990
433.milc	C	fp	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
470.lbm	C	fp	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
482.sphinx3	C	fp	1.0000	1.0000	1.0000	0.9819	0.9939	0.9879
471.omnet.	C++	int	0.9974	0.9604	0.9786	0.9963	0.9564	0.9759
473.astar	C++	int	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
483.xalan.	C++	int	0.9958	0.9888	0.9922	0.9920	0.9886	0.9903
444.namd	C++	fp	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000
447.dealII	C++	fp	0.9942	0.9707	0.9823	0.9889	0.9731	0.9810
450.soplex	C++	fp	1.0000	0.9847	0.9923	1.0000	0.9858	0.9929
453.povray	C++	fp	1.0000	0.9566	0.9778	1.0000	0.9573	0.9782
C overall	C	both	0.9982	0.9982	0.9982	0.9949	0.9946	0.9948
C++ overall	C++	both	0.9959	0.9792	0.9875	0.9922	0.9796	0.9859
Overall	all	both	**0.9965**	**0.9847**	**0.9906**	**0.9930**	**0.9840**	**0.9885**

Figure 5.12: SPEC 2006 results (LLVM compiled binaries)

Binary	Total funcs.	Direct call	Direct jump	Indirect	Unreachable
400.perlben.	1742	48.22%	0.46%	40.18%	11.14%
401.bzip2	81	61.73%	1.23%	9.88%	27.16%
403.gcc	4653	68.56%	2.82%	20.89%	7.57%
429.mcf	34	73.53%	0.00%	17.65%	8.82%
445.gobmk	2543	26.27%	0.55%	70.31%	2.87%
456.hmmer	504	54.96%	2.98%	5.56%	36.31%
458.sjeng	146	72.60%	4.11%	8.90%	14.38%
462.libquan.	109	68.81%	3.67%	6.42%	21.10%
464.h264ref	535	79.63%	2.80%	7.29%	10.28%
433.milc	246	79.67%	0.81%	3.25%	16.26%
470.lbm	28	75.00%	0.00%	21.43%	3.57%
482.sphinx	338	70.71%	1.18%	3.85%	24.26%
471.omnet.	2036	27.36%	1.82%	56.93%	13.36%
473.astar	98	75.51%	0.00%	8.16%	16.33%
483.Xalan	13525	33.62%	2.60%	53.84%	9.18%
444.namd	105	45.71%	0.00%	51.43%	2.86%
447.dealII	7242	26.90%	1.20%	30.46%	37.21%
450.soplex	935	43.42%	3.32%	38.93%	11.98%
453.povray	1639	58.88%	1.59%	30.14%	6.47%
410.bwaves	17	58.82%	0.00%	35.29%	5.88%
416.gamess	2898	94.79%	1.04%	0.59%	3.49%
434.zeusmp	86	66.28%	0.00%	6.98%	25.58%
435.gromacs	1100	70.36%	1.09%	4.09%	24.36%
436.cactus.	1311	44.47%	0.76%	14.80%	39.97%
437.leslie3d	32	71.88%	0.00%	18.75%	9.38%
454.calculix	1338	69.43%	2.24%	0.45%	26.83%
459.Gems.	78	78.21%	2.56%	6.41%	10.26%
465.tonto	3851	68.87%	4.05%	0.73%	24.51%
481.wrf	2888	55.40%	3.84%	0.66%	39.89%
Overall	**50138**	**48.06**	**2.16**	**30.83**	**17.70**

Figure 5.13: Functions identified in each step

includes direct calls made within functions that are only indirectly reached.) Some C programs (such as 445.gobmk), however, contains a large number of indirect functions. For many C++ programs, because virtual functions[10] are abundant, there are generally more indirectly reached functions. The fourth column presents the percentage of functions that are reached *only* by direct jumps (i.e., tail called). These functions are not rare in optimized binaries.

Note that for some benchmarks, the percentage of unreachable functions is quite high (average 17% and up to 40%). To verify these results, we selected a subset of these programs, and used Pintools [65] to record the locations reached via calls or jumps. We found that none of these addresses corresponded to functions determined unreachable by our technique.

After checking source code, we found that the unreachable functions are mostly global (i.e., non-static) functions which are neither called directly nor have their addresses taken. Although they are not used, compilers don't generally remove them unless specific actions are taken during the build process to eliminate them. (Note that this is different from static functions whose visibility is within the same compilation unit — it is more common for unused static functions to be removed by default.)

Effectiveness of Interface Checking Techniques. As discussed, function interface checking is critical in pruning spurious functions from the identified candidate set. In this section, we evaluate the effectiveness of each checking mechanism independently. The results are presented in Figure 5.14. Again, only GCC -O2 compiled binaries for x86-32 are shown.

As shown in the figure, each checking mechanism is independently effective in identifying a significant fraction of all spurious ("total pruned" in the figure) functions. However, in general, no single mechanism is able to detect all spurious functions. It is through their combination that we can effectively reduce the number of spurious functions to a very low number. Note that for four of the binaries, no spurious functions are pruned. This is because all the functions enumerated happen to be real functions.

5.5.6 Performance

Our focus so far has been on accuracy rather than run time, and hence we have not made any systematic efforts to optimize performance. Nevertheless, it is useful to compare its performance against previous techniques.

As compared to machine learning based approaches [15, 87], one of the advantages of our approach is that it does not require training, which is ex-

[10] Virtual functions can only be *indirectly* called through a V-table.

Binary	Total	Control flow checking			Data flow checking	
	pruned	entry	exit	internal	argument	callee-save
400.perlben.	1630	91.60%	79.57%	39.26%	57.67%	53.31%
401.bzip2	48	100.00%	33.33%	89.58%	85.42%	87.50%
403.gcc	5845	94.06%	86.35%	38.25%	71.87%	46.14%
429.mcf	0	0.00%	0.00%	0.00%	0.00%	0.00%
445.gobmk	379	61.48%	89.18%	37.73%	46.70%	40.37%
456.hmmer	245	89.80%	81.63%	40.41%	76.33%	60.82%
458.sjeng	130	99.23%	44.62%	45.38%	72.31%	80.00%
462.libquan.	1	0.00%	0.00%	0.00%	100.00%	0.00%
464.h264ref	89	89.89%	73.03%	37.08%	84.27%	66.29%
433.milc	43	88.37%	97.67%	6.98%	69.77%	65.12%
470.lbm	0	0.00%	0.00%	0.00%	0.00%	0.00%
482.sphinx	16	31.25%	31.25%	18.75%	68.75%	12.50%
471.omnet.	130	72.31%	76.92%	22.31%	70.77%	43.08%
473.astar	0	0.00%	0.00%	0.00%	0.00%	0.00%
483.Xalan	1916	46.03%	65.55%	21.14%	80.64%	54.96%
444.namd	2	0.00%	50.00%	50.00%	0.00%	100.00%
447.dealII	1851	18.10%	65.80%	17.77%	64.94%	57.16%
450.soplex	228	84.65%	75.44%	26.32%	62.72%	56.58%
453.povray	1602	91.39%	38.76%	19.48%	75.28%	16.10%
410.bwaves	0	0.00%	0.00%	0.00%	0.00%	0.00%
416.gamess	3088	79.18%	56.99%	35.65%	73.74%	52.91%
434.zeusmp	14	0.00%	50.00%	57.14%	71.43%	50.00%
435.gromacs	360	84.44%	87.50%	34.44%	73.61%	35.56%
436.cactus.	376	95.48%	80.59%	56.65%	84.31%	58.24%
437.leslie3d	2	0.00%	100.00%	100.00%	50.00%	50.00%
454.calculix	269	75.09%	87.36%	53.53%	40.15%	37.17%
459.Gems.	72	80.56%	76.39%	50.00%	43.06%	62.50%
465.tonto	1631	90.74%	88.90%	17.78%	41.02%	40.34%
481.wrf	572	77.45%	93.71%	27.97%	29.55%	36.71%
Overall	**21360**	**77.44**	**73.32**	**31.75**	**67.02**	**47.17**

Figure 5.14: Effects of each checking mechanism

Tool	Experiment setup			x86-32 binaries		x86-64 binaries	
	machine	CPU	RAM	training	testing	training	testing
ByteWeight	desktop	4-core 3.5GHz i7-3770K	16G	293 h_c (estimate)	457,997 s	293 h_c (estimate)	593,170 s
Neural	Amazon EC2 c4.2xlarge	8-core 2.9GHz Intel Xeon	15G	20 h	1,062 s	20 h	1,018 s
Ours	laptop	4-core 1.7GHz i5-4210U	8G	0	47,880 s	0	36,300 s

Figure 5.15: Experiment setup and performance comparison (h_c = compute hours, h = hours, s = seconds)

pensive. The results of our analysis, together with those from ByteWeight [15] and neural network based system [87], are summarized in Figure 5.15. The numbers are based on our first data set, and 10-fold cross validation for machine learning systems.

The neural network based system uses much less time for the testing because it only identifies the bytes where functions start and end, without recovering the function body. As a comparison, ByteWeight and our system follow the CFG to identify function ends, therefore can recognize the exact instructions belonging to the function, and identify physically non-contiguous parts of the function.

Currently, it takes about 40 seconds on average to analyze a binary of our test suite. Although this is already satisfactory for many cases, there are many opportunities for improvement. For example, spurious functions can be immediately spotted if the entry basic block has violating behavior and therefore analysis of the whole function can be avoided. This is in contrast to our current naive implementation that performs complete analysis and checks.

5.6 Case Studies

Since function recognition serves as an essential step for many techniques working on binaries, to understand how well our approach can be used, we analyze several representative classes of applications.

General evaluations have already shown that our approach has better accuracy than state-of-the-art machine learning techniques (Section 5.5), in this section, we focus on binary instrumentation tasks which usually impose more stringent requirements.

5.6.1 Control-Flow Integrity

As discussed, CFI is an effective technique for mitigating code reuse attacks [6, 106, 104, 93, 94]. CFI can be coarse-grained or fine-grained, depending on the precision of the CFG computed by different static analysis techniques. It has been shown that coarse-grained CFI provides less security because it is more easily bypassed.

CFI is sensitive to the *recall* of *indirectly* reachable functions. Specifically, consider a CFI policy that an indirect call must target the entry point of one of the indirectly reachable functions in the program. With a recall rate less than 100%, some legitimate function entry points would not have been identified, and hence CFI enforcement based on such an analysis can lead to runtime failures of legitimate programs.

To evaluate our system in the context of CFI enforcement, we tested with all SPEC 2006 programs that are compiled with GCC (-O2). We focus on indirectly reachable function starts in this case, because function ends are not required for the instrumentation. To get the ground truth for indirectly reachable function starts, we first preserve relocation information during the compilation and linking process. For each relocation entry, if the involved address matches a function start in the symbol table, we consider this address taken and the corresponding function indirectly reachable. Finally, our results are evaluated against this set.

The evaluation shows that for all programs in SPEC 2006, we have achieved 100% recall for *indirectly* reachable function starts. Indeed, the perfect recall matches our expectation because by design our analysis enumerates all indirectly reachable function starts, and then eliminates those spurious ones detected by our analysis.

We note that although a high precision rate is not critical for soundness of CFI instrumentation, it impacts security. Therefore, a strategy that sacrifices precision drastically for perfect recall (e.g., the static analyses used by BinCFI) is unattractive. Instead, we achieve perfect recall, while maintaining a high precision of more than 98% on average. This allows us a 3-fold reduction as compared to BinCFI-allowed targets.

In comparison, machine learning based systems [15, 87] identify function boundaries with a model of surrounding code, and their system cannot be easily tuned towards 100% recall. Therefore, false negatives may occur for indirectly reachable functions (and others). For example, as Shin et al. have observed from their system [87]: "False negatives often occurred when instructions that would typically occur in the middle of functions occurred at the beginning of a function"; and when functions have multiple entries: "Many of the false negatives occurred at the second entry points to functions, given that the instructions before it are not the ones which usually end functions".

5.6.2 Function-based Binary Instrumentation

Many binary instrumentation techniques operate on individual functions as a unit [25, 82, 22]. Compared to CFI which is recall-sensitive, these applications are usually more sensitive to *precision*. This is because, an unrecognized function may only leave the function untouched, while a misidentified function could lead to breaking functionality if the technique assumes correct function boundary information. In this section, we analyze the applicability of our function identification system for these applications.

Since directly called functions are free of errors and unreachable functions

```
0818ba30 <func>:
818ba30:    fld   0x86ed1d0 ;real function start
818ba36:    fstp  0x8ca5af0
...         (similar instructions)
818c784:    fld   0x87202c0 ;spurious function start
818c78a:    fstp  0x8ca5908
818c790:    fld   0x87202c8
818c796:    fstp  0x8ca5910
...         (similar instructions)
818c8d0:    mov   $0xf2,0x8ca5a4c
818c8da:    mov   $0xf6,0x8ca5aec
818c8e4:    ret
```

Figure 5.16: A falsely identified (indirectly reachable) function [818c784, 818c8e4]

are not relevant for correct functionality (as they are not being executed at runtime), imprecision could possibly originate from two sources: indirectly reachable functions and direct jump reached (tail called) functions. We analyze these two cases respectively.

For the first case, an address could be incorrectly identified as an (indirectly reachable) function start if our interface checking mechanism was insufficient. Although our comprehensive checking schemes are generally effective and can remove vast majority of the spurious function starts, such misses do happen. Figure 5.16 shows one example. In this case, since all instructions access global memory and there are no stack or general-purpose register operations, our interface checking could not identify [818c784, 818c8e4] as a spurious function.

Despite these imprecisions, one distinguishing feature of our system is that the real function which encloses the spurious one is always identified. In Figure 5.16 for example, [818ba30, 818c8e4] is also recovered (recall in our model, the recovered functions can overlap or share code). And with this property, different measures could be taken for different instrumentations to cope with the imprecisions.

For RAD [25], no work is required at all because the instrumentation is resistant to such imprecisions[11]. For other more complicated instrumentations [82, 22], the overlapping functions could have their own instrumented version

[11] This is because, at the spurious function start, an extra, unneeded "return address" is pushed to the shadow stack. While this slightly increases attacker's options, it does not break program functionality since at the function epilogue, return addresses is popped repeatedly from the shadow stack until there is a match. Note that the true return address does present in the shadow stack, because the larger, real function is also recovered.

(which are disjoint), and an address translation scheme for indirect branches (commonly adopted by binary transformation systems [65, 106, 105, 89]) could be used. With this technique, an indirect call target is translated at runtime to point to its instrumented version before control transfer. Since the falsely identified function is never called at runtime, the incorrect instrumentation will not be executed.

Our system may also falsely recognize intra-procedural jumps as direct tail calls (the second type of error). Essentially, this is equivalent to splitting the original function into two. However, we note that this will not introduce any correctness problems, as all executed instructions and exercised control flows have been well captured.

Above analysis indicates that our function recognition is effective and only leaves *limited* error possibility. The inaccuracies tend to either have no effect for function-based instrumentation correctness, or can be easily coped with. As a comparison, since machine learning based approaches rely on code or byte patterns, false positives of function starts and ends are much more random and difficult to deal with. Finally, as can be seen in both case studies, our system automatically classifies identified functions based on their reachability property, which can enable more flexible instrumentations.

5.7 Extensions

Special calling conventions. Currently our data flow checking technique is based on well-respected system ABIs and calling conventions, and it can be adapted to other architectures such as ARM [23]. We note although non-standard calling conventions have not appeared in our tests, they could be used in some cases, e.g., function calls within a single translation unit. To deal with this issue, a “self-checking” mechanism can be adopted.

Specifically, note that ABI violations can occur only in the context of direct calls and jumps. (Since a compiler cannot be sure about the target of an indirect control flow transfer, it cannot assume that such a transfer is intra-module.) Since we don’t apply interface checks for direct calls, ABI violations won’t pose a problem in their context. That leaves direct jumps (i.e., direct tail calls) as the only problem case. We develop a self-checking mechanism in this case. Specifically, we can perform interface checks on a subset of directly called functions to determine whether ABI is respected. If not, we identify a relaxed set of conventions that are respected in direct calls, and apply these relaxed checks to tail call verification. (Note that verification of indirectly called functions can continue to rely on ABI.)

6 Accurate Recovery of Function Types

After functions are recognized, another important task is to recover their types. A function type is given by the argument types and return type. At binary level, arguments passing and value return are usually based on stack or registers. However, the numbers and types for the arguments and return values are not present.

Function type recovery is critical for several applications. First, reverse engineering, especially decompilation requires function type information. Moreover, the types for functions as well as function pointers can be used to significantly refine program call graph — useful for many static analyses. Finally, a fine-grained and effective CFI policy can be developed based on function type information. Note that in Section 4 we addressed the problem of backward edge protection by incorporating a compatibility-aware shadow stack, but forward edges are not well protected.

In this chapter, we focus on recovery of function types that can be used for deriving a fine-grained call graph. The call graph can be adopted in static analysis, and most importantly, fine-grained forward-edge CFI enforcement. To that end, we develop novel static analysis techniques to recover function types for indirectly called functions, and also identify arguments used by each indirect callsite. Our analysis is designed to be conservative, so that a CFI policy can be enforced without resulting in false alarms. On the other hand, it is also fine-grained and provides stronger protection than existing CFI schemes. From a high level, our analysis works by accurately analyzing the number and types of arguments for functions and function pointers. And the enforcement policy is such that an indirect callsite can only reach callees with compatible signatures.

6.1 Indirect Callee Arguments

To analyze indirect callee arguments, the first step is to identify these functions. In Section 5.2.2, we have described how indirectly reachable functions are recognized. We utilize the same technique here. Note, however, a slight difference is that all scanned constants that pass interface checking are considered as indirectly reachable functions, although they could be *directly* reachable at the same time.

```
 1 struct A {
 2   int index;
 3   int addend;
 4   int array[4];
 5 };
 6
 7 int f(struct A s) {
 8   return s.array[s.index] + s.addend;
 9 }
10
11 f:
12 movl   4(%esp), %eax
13 movl   12(%esp,%eax,4), %eax
14 addl   8(%esp), %eax
15 ret
```

Figure 6.1: An example function with a `struct` argument

6.1.1 Challenges

For each function, its stack arguments are prepared by the caller and are placed right above the return address. Therefore, an effective approach for inferring stack arguments is to analyze how they are accessed. Specifically, since the stack arguments are located at positive offsets to stack pointer value on function entry, we need to identify memory accesses to those locations.

Obviously, more accurate argument information can lead to more precise CFI policies [92]. However, there are several challenges in obtaining accurate stack argument information through analysis at the binary level.

First of all, a single multi-word argument can be passed through multiple stack slots, the same way as passing multiple word-sized arguments. Therefore, it may be challenging to differentiate between the two cases. Figure 6.1 presents such an example. The source code is shown in the upper part, while the corresponding assembly code is shown in the lower part. In this example, the only argument is of type `struct A` (non-scalar), and is passed by value: all its fields are pushed on the stack. Note that in assembly the fields `index` and `addend` are accessed on line 12 and 14, respectively. However, It is impossible to determine whether they are fields of a struct or two different scalar arguments.

The second issue is aliasing: stack arguments can be accessed using indirect memory references. In Figure 6.1, the elements of `A.array` are accessed with an index variable whose value is not known at compile time. Line 13 shows the corresponding assembly code. In this case, the number of arguments cannot be determined as the value of the index variable is not bounded.

6.1.2 Our Analysis

Recognizing the difficulties in *exact* function argument recovery, we propose to use an alternative approach: a "best-effort" argument recovery scheme. Ideally, this approach should be sound for CFI enforcement, yet does not result in too much loss of precision.

Specifically, for each function, we propose to identify *definite argument region* instead of the accurate *argument list*. Our analysis ensures that any byte inside argument region belongs to some argument (hence the word *definite*), however, there could be extra arguments that are not included in this region. Note that this is essentially an *underestimation* of the memory region used for argument passing. However, as will be shown later, this result is useful and sound for enforcing finer-grained CFI for binaries.

To identify definite argument region, we perform an abstract stack analysis as described in Section 3.3. *Definite arguments* are first recognized if the following two properties are satisfied: 1) the abstract value for the memory access address is a singleton; and 2) the abstract value for the memory access address is of the form $BaseSP + C$, where $BaseSP$ is a symbolic variable that represents the stack pointer value on function entry, and C is a positive constant.

The definite argument region is then determined using the obtained definite arguments. Specifically, the region begins from right above return address, until the last byte of definite argument with the largest offset. Note that there might be bytes inside the definite argument region but does not belong to any definite argument, for the following two reasons. First, some arguments, although passed in, are not actually used by the callee; and second, some arguments may only be accessed through indirect memory references (e.g., `A.array` in Figure 6.1). In both cases, they are not recognized as definite arguments by our analysis.

Note that in above cases, we can still obtain the *exact* argument region unless the unused or indirectly referenced arguments are the last ones.

Function summaries A situation we need to deal with is that the analyzed function may call other functions, which results in two complications: first, the callees may access the current function's arguments in some cases; and second, the stack pointer may change due to the callee invocation.

To address these issues, we utilize function summaries, as described in Section 3.3. Specifically, for direct function calls, we compute a summary for each callee and apply it at the callsite. Note that the function summary not only includes the definite argument region it accesses (using the approach described above), but also the stack pointer (and other register) change as a result of the callee invocation.

To apply a function summary, at each callsite, we check if callee's definite argument region has a part that is above caller's return address on the stack. If so, we combine this part to the the caller's definite argument region. We also adjust the stack pointer (as well as other register) value based on the callee summary. Note that the summaries are applied iteratively for all functions, until a fix point is reached.

Note that for *indirect* calls, the function summaries cannot be computed or applied as the callee cannot be statically determined. In this case, we simply ignore the callee's effects. We next discuss why this does not affect the correctness of our analysis.

We could fail to identify some stack arguments of the caller, if those are only accessed by the (indirect) callee that we have ignored. However, this is relatively rare and will only result in an underestimation of the argument region. As discussed, it is just a precision issue and does not affect correctness.

Another issue is stack pointer change. We note that in some rare cases, indirect callee invocation may not preserve stack pointer. If we ignore this effect, for argument-accessing instructions after the indirect call, our analysis may fail to identify their correct offsets. However, even in this extreme case, the analysis only results in imprecision, but not incorrectness. The key observation is, if a function invocation does not preserve stack pointer, its value could only become larger which is the effect of callee cleanup (assuming stack grows downwards). Therefore, failing to properly increase stack pointer will only make the recognized stack region become smaller, which satisfies our goal of deriving an underestimation of stack arguments.

Variadic Functions Variadic function is a special kind of function that can take a varying number of arguments. We need to make sure that our analysis works correctly for them.

Figure 6.2 shows a typical variadic function. In this case, our analysis recognizes the first argument (`fmt`) because accessed stack address is concrete. On the other hand, the variadic arguments (...) are not recognized because they are accessed using an aliased address[12]. This is actually desired result:

[12] The abstract value for these arguments is $BaseSP + [8, \infty)$. Again, $BaseSP$ represents

```c
int compute(const char *fmt, ...) {
  va_list argp;
  const char *p;
  int tmp, result = 0;

  va_start(argp, fmt);
  for (p = fmt; *p != '\0'; p++) {
    tmp = va_arg(argp, int);
    if (*p == 'a')
      result += tmp;
  }
  va_end(argp);
  return result;
}
```

Figure 6.2: A variadic function

recognizing the normal arguments and ignoring the variadic arguments is conservative, as it provides us an underestimation of the argument region.

6.2 Indirect Callsite Arguments

6.2.1 Challenges

In addition to callee argument analysis, we need to analyze the number of stack arguments that are passed for each indirect callsite. This may seem straight-forward at first sight: just record all slots for stack pushes before the call. However, this simple approach may be inaccurate or even incorrect. This is because, for example, due to compiler optimizations, the argument pushing instructions may not necessarily present in the same basic block as the indirect call.

Figure 6.3 presents such an example. The function pointer takes exactly two arguments, however, in basic block starting at label `.L6`, only one argument is passed. Note that the other argument is passed at line 17.

Another challenge is to differentiate stack arguments from local variables and register spills. In the same example, line 16 assigns 0 to local variable `x` (corresponds to line 4 in source code). However, it can be incorrectly recognized as passing an argument.

the stack pointer value on function entry.

```
extern void (*fp)(int, int*);
int a, b;
int test() {
  int x = 0;
  int *p = &x;
  if (a > 0)
    (*fp)(a, p);
  else
    (*fp)(a + 2, p);
}

test:
  subl   $12, %esp
  movl   a, %eax
  leal   8(%esp), %edx
  movl   $0, 8(%esp)
  movl   %edx, 4(%esp)
  testl  %eax, %eax
  jg     .L6
  addl   $2, %eax
.L6:
  movl   %eax, (%esp)
  call   *fp
  addl   $12, %esp
  ret
```

Figure 6.3: Challenges for identifying callsite arguments

6.2.2 Our Analysis

To overcome these problems, our insight is that the exact argument information is not necessary (although more accurate) for enforcing a finer-grained CFI, and we can derive an *overapproximation* of prepared arguments. Similar to callee analysis, for a callsite, we focus on the *stack region* that is used for passing arguments. In other words, our callsite analysis aims to infer an overapproximation of the argument region. Recall that our callee analysis produces an *underapproximation* of argument region, therefore, our policy enforces that an indirect callsite can only target callees which accept *smaller or equal* argument region, but larger.

To identify callsite argument region, we developed a reaching definition analysis for stack locations. Specifically, all stack locations (below return address) are involved, and a stack location is *defined* (i.e., considered as potential argument) if it is written to. Since compiler knows the exact location for callsite stack arguments, they should be written using concrete offsets (relative to $BaseSP$, the stack pointer value on function entry). Therefore, our analysis ignores indirect memory writes (the address value set is not a singleton), which are not used for argument pushing. Note that this applies to callsites to variadic functions as well, as there is no difference with calls to normal functions (the number of arguments is known in both cases).

In our analysis, all stack locations are *killed* in case of an indirect call instruction (i.e., it is definitely not an argument for another indirect call that comes later). This is because, for indirect call instructions, since the unknown callee might overwrite its own arguments during the invocation, compiler has to make conservative assumptions that all current stack arguments could be clobbered, and cannot be reused for future indirect calls.

Note that special care needs to be taken for functions that call `alloca`s. In assembly code, `alloca` subtracts an unknown delta to the stack pointer, and therefore its new offset to $BaseSP$ cannot be determined. However, all previously defined locations should not be arguments to future indirect callsites, as their distances to the stack top are not statically known by the compiler. Therefore, our analysis kills all defined locations, and assigns a new symbolic value $NewSP$ to stack pointer.

For each callsite, after performing the reaching definition analysis, the *defined* stack slots are the potential stack arguments. However, they are likely to contain *spurious* arguments that are actually local variables or register spills. We therefore develop further techniques to refine the initial results.

First, we observe that stack arguments for a callee must all be pushed and laid out *contiguously* to each other, to meet callees' expectations. Therefore, an undefined slot indicates that the following defined ones are all spurious arguments. For example, if the defined stack slot offsets[13] are {4, 8, 16, 20}, we exclude offset 16 and 20 as stack arguments, as offset 12 is not defined (i.e., definitely not prepared as stack argument).

Our second technique for removing spurious arguments is to eliminate "callee-save" slots from the initial results. Since a function needs to preserve certain register values, they are first saved to callee-save slots on the stack, and then restored before function return. Obviously, these stack locations should not be clobbered or used to pass arguments. Although we could use a platform-agnostic yet complicated analysis [37], we opt to extract callee-save information from `.eh_frame` section of binaries. Note that this information is widely available from stripped binaries for UNIX-like operating systems [45].

After spurious argument removal, the argument region for each callsite could be determined. As discussed, it can be used together with the callee argument region for a fine-grained CFI enforcement.

6.3 CFI Policies

A CFI policy is concerned with restricting indirect control flow transfers to pre-defined sets of targets. As discussed, we focus on forward-edges, or more specifically, indirect calls in this chapter.

Indirect calls can be used for callbacks, or dynamic function dispatches. Specially, in C++ programs, all virtual function dispatches are based on indirect calls. Since virtual functions are abundant in C++ programs, it is common that in these binaries, the majority of indirect calls are results of virtual dispatches.

In the following sections, we present different fine-grained CFI policies that are based on function types. We first present a general policy that is widely applicable to all C/C++ programs, followed by a more precise policy for securing virtual dispatches of C++ programs.

6.3.1 (Normal) Indirect Calls

For normal (non-virtual) indirect calls, our CFI policy is based on argument arity. Specifically, an indirect callsite is allowed to reach a function if the following constraints are satisfied:

- The target is a start of an indirectly reachable function.

[13] The slot size is 4 bytes. The offsets are relative to the stack pointer value on entry of callee, and therefore slot with offset 0 holds the return address.

- The callsite prepares more arguments than the target function takes.

Note that this policy is conservative. Since our analysis identifies more arguments for callsites but not less, and identifies less arguments for callees but not more, the policy would not break any legitimate control flow transfers. On the other hand, it reduces the number of *invalid* targets allowed by each callsite. We quantify this in Section 6.4.

6.3.2 Virtual Calls

Section 2.5.2 presented existing techniques for restricting C++ virtual calls. In this section, we develop a more precise policy that is based on accurate analysis.

With recovered function types, a natural choice is to combine it with the vfGuard policy (Section 2.5.2). Specifically, a target is allowed for a virtual call if the following two conditions are satisfied:

1. The target is an entry of a valid VTable, and its offset into the VTable is the same as that determined by the virtual callsite (the vfGuard policy).

2. The target function has compatible signature with the virtual callsite (the same policy for normal indirect calls).

Our CFI policy restricts this policy further. The key observation is that polymorphic objects usually have multiple virtual functions, and one such object tends to call different virtual functions along its life time. If different virtual calls are found to be invoked on the same object, the base CFI policy on each virtual call can be propagated. Specifically, our policy enforces that the virtual calls on the same object can only invoke virtual functions of some VTable, if *each* callsite-callee pair is compatible.

Figure 6.4 presents such an example. In this code snippet, the same `this` pointer is used for invoking different virtual functions. Specifically, in line 3 the `this` pointer is loaded to `ebx` from a memory location. The VTable pointer is retrieved from its first field and loaded to `eax` in line 4. The `this` pointer is pushed as the first argument in line 5, and a virtual call is invoked in line 6, and its target comes from offset 4 of the VTable pointed by `eax`. Lines 7-9 invoke another virtual function on the same `this` pointer. And the offset into the VTable is 12 (0xc).

In this example, the two virtual calls are invoked on the same object pointer (`ebx`), and they are said to be in the same cluster. All recovered VTables are matched against this cluster, instead of each virtual call as in vfGuard. A

```
push   %ebx
sub    $0x18,%esp
mov    0x20(%esp),%ebx  ; 'this' ptr is loaded to %ebx
mov    (%ebx),%eax      ; %eax points to the VTable
mov    %ebx,(%esp)      ; 'this' ptr is pushed as argument for the
    1st vcall
call   *0x4(%eax)       ; first virtual call
mov    (%ebx),%eax      ; %eax points to the VTable
mov    %ebx,(%esp)      ; 'this' ptr is pushed as argument for the
    2nd vcall
call   *0xc(%eax)       ; second virtual call
...
```

Figure 6.4: Multiple virtual calls with the same `this` pointer

VTable is only matched if each virtual call of the cluster satisfies the base policy. Note that if a VTable is matched, the corresponding virtual function address at the right offset for a virtual call is collected into its allowed target set.

Figure 6.5 illustrates this procedure. The left part of the figure is the same code as shown in Figure 6.4, but with non-call instructions removed for clarity. The right part presents the identified VTables. Note that only virtual function array of the VTable are shown, and corresponding offsets for virtual function entries are marked. The numbers in parentheses indicate the number of arguments for virtual callsites and virtual functions, as detected by our analysis.

In this figure, vfGuard policy allows control flow transfers referred by all arrows. The second callsite is not allowed to target VTable B, as there is no corresponding entry at offset 0xc. However, our policy is stronger in that all control flow transfers represented with dashed arrows are disallowed. For example, edge (c) originated from the first virtual call is prevented because the second virtual call indicates that VTable B cannot be a match for the object. Moreover, edges (d) and (e) are not allowed: the second virtual call has incompatible signature with the corresponding virtual function in VTable C, indicating a mismatch between the object and VTable C.

Static Analysis To enable above policy, we develop static analysis techniques to cluster virtual calls based on whether they operate on the same object. To capture the usage of object (i.e., `this`) pointers, precise data flow information is required. We therefore use an approach that is similar to vf-

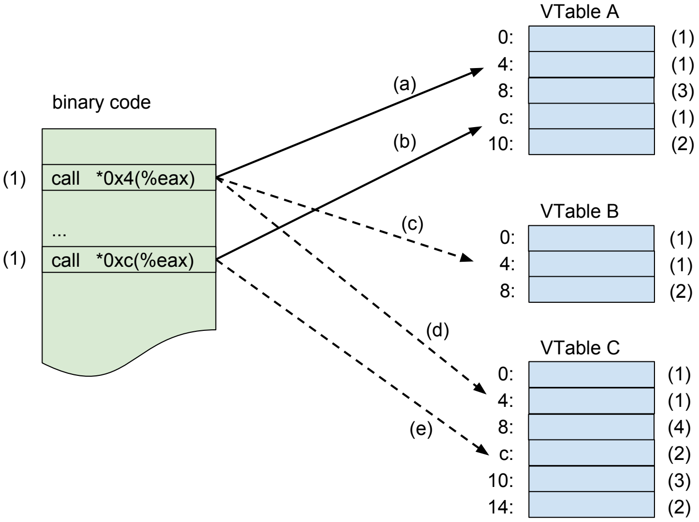


Figure 6.5: CFI policies for virtual calls

Guard.

We choose to transform each function into static single assignment (SSA) form [30, 95], since the def-use and use-def chains are explicit. Our analysis is intra-procedural and is performed on functions that contain multiple indirect callsites. The function is first lifted into an intermediate representation (IR), and then its control-flow graph (CFG) is built. After the code is transformed into SSA, a phase of definition propagatition is performed. This procedure is recursive and stops when registers and memory locations are defined with expressions containing only the initial values of registers, function arguments or global variables.

Since multiple paths may be involved, the target of a virtual call may be represented as a collection of expressions, each corresponds to a specific path. If the object pointer is the same for all paths of different virtual calls, these virtual calls are added into the same cluster.

As an example of our analysis, for the function shown in Figure 6.4, our analysis generates the target expressions for the two different callsites:

$$call\ *(*(*(esp_0 + 4)) + 4) \tag{4}$$

$$call\ *(*(*(esp_0 + 4)) + 12) \tag{5}$$

Since $*(esp_0 + 4)$ is used as the `this` pointer for both virtual callsites, they are added into the same cluster.

	Callees			Callsites		
file	perfect	total	%	perfect	total	%
astar	81	86	94.19%	1	1	100.00%
bzip2	48	53	90.57%	11	11	100.00%
gcc	3138	3489	89.94%	57	70	81.43%
gobmk	723	2332	31.00%	26	27	96.30%
h264ref	296	353	83.85%	5	8	62.50%
hmmer	434	451	96.23%	2	2	100.00%
libquantum	77	84	91.67%	0	0	-
mcf	21	23	91.30%	0	0	-
omnetpp	1355	1704	79.52%	103	175	58.86%
perlbench	892	1146	77.84%	19	23	82.61%
sjeng	83	88	94.32%	0	0	-
geomean	**199**	**244**	**81.48%**	**12**	**14**	**83.59%**

Figure 6.6: Statistics for function argument analysis accuracy

6.4 Evaluation

6.4.1 Analysis Precision

In order to evaluate the precision of our function type analysis, we extracted ground truth from the DWARF debug information of the ELF binaries. Note that our analysis operates on the stripped version of these binaries, and is not making use of this debug information.

For callee arguments evaluation, we first extract functions which are identified with the `DW_TAG_subprogram` tag, which includes the function start address as one of its properties. For each function, we extract its argument information from entities with `DW_TAG_formal_parameter` tag, which includes properties about the location, type and size of the argument. Based on this information, we derive the argument region as the ground truth, and compare it with our analysis results.

The left part of Figure 6.6 shows the precision evaluation for callee (function) argument analysis. The column `total` means the total number of functions for each binary, while the `perfect` column indicates the the number of functions that our analysis produces perfectly accurate result (same argument region as in the ground truth). As presented in the figure, in 82% cases (geomean) our analysis is perfectly accurate and does not underestimate arguments.

To evalute callsite analysis, we rely on two approaches. First, similar to callee analysis evaluation, we utilize debug information as ground truth. Although DWARF standard has no official support yet [1], it is fortunate that GCC has an extension that encodes arguments prepared for callsites [5]. However, since the expected use for the extension is for other purpose (accurate backtrace for debugging), one limitation is that the information is not complete in binaries: some callsites do not contain argument information, or only contain information for some arguments. Therefore, the only verification we can do is to check the argument region produced by our analysis actually *includes* the region derived from the debug information. Or in other words, our analysis does not over-approximate arguments. The evaluations on SPEC 2006 programs indicates 0 error for our callsite argument analysis, as compared to the incomplete debug information.

For a more comprehensive evaluation, we used a second method: we developed a Pintool [65] for dynamic analysis. Specifically, the tool records all indirect edges that are taken (i.e, the source and target instructions) at runtime. We also assume that if there is a control transfer from a callsite to a callee, they have matching signatures, i.e., the same number and types of arguments. Since from debug information we have the ground truth for each *function*, we can use it for the corresponding *callsite*.

The right part of Figure 6.6 shows the results for this evaluation: in 84% (geomean) of the cases, our callsite analysis was able to produce perfect argument region information.

6.4.2 Policy Correctness

Although a stricter CFI policy is desired for stronger protection, it is even more important that the deployed policy is correct: i.e., it is compatible with benign executions of the program. Because otherwise the policy would not be adopted due to false alarms.

To evaluate the correctness of our policy, we used the same Pintool as discussed in the last section. For the records generated by dynamic analysis, we check whether each edge violates the CFI policies that we have generated. Note that policies for both normal indirect calls and virtual calls are respectively checked.

We evaluted SPEC 2006 benchmark, and found no such violations. This indicates that our policy is compatible with benign program executions.

Note that although our policies can be enforced using different platforms [105, 65, 50], the actual instrumentation is not the focus of this work. Indeed, a CFI policy is nothing but a mapping between a control-flow transfer instruction to a set of allowed targets, and existing systems can be readily used for realizing

File	BinCFI	IFunc	Reduction	Type check	Reduction	Overall reduction
astar	10	10	0.00%	4	60.00%	60.00%
bzip2	58	10	82.76%	2	80.00%	96.55%
gcc	7751	1222	84.23%	885	27.58%	88.58%
gobmk	2580	1798	30.31%	1542	14.24%	40.23%
h264ref	169	48	71.60%	34	29.17%	79.88%
hmmer	260	31	88.08%	23	25.81%	91.15%
libquantum	9	9	0.00%	7	22.22%	22.22%
mcf	8	8	0.00%	6	25.00%	25.00%
omnetpp	1369	1248	8.84%	760	39.10%	44.49%
perlbench	2595	731	71.83%	341	53.35%	86.86%
sjeng	152	16	89.47%	11	31.25%	92.76%
geomean	**454**	**153**	**66.34%**	**83**	**45.83%**	**81.77%**

Figure 6.7: Indirect call targets reduction of our language-agnostic CFI policy

our policy efficiently. This has been demonstrated repeatedly by many CFI solutions [106, 104, 75, 94].

6.4.3 CFI Strength

Several metrics were proposed to evaluate the protection strength of a CFI scheme [106, 18]. However, their limitations have been recognized and discussed [19, 18]. While we acknowledge that a more systematic evaluation approach should be developed in future research, to enable direct comparison, we utilize existing metrics.

Generally, the more targets an indirect call is allowed, the more freedom an attacker is granted, and the more likely an exploit could be launched. Therefore, we evaluate how our technique helps reduce the possible targets. Specifically, we first compare our language-agnostic policy with that of BinCFI [106], an existing coarse-grained CFI technique that works on COTS binaries[14]. Then our CFI policy for virtual calls are evaluated against that of vfGuard [77], a state-of-the-art precise virtual call protection system.

Figure 6.7 presents our evaluation results for SPEC 2006 programs. Column `BinCFI` shows the allowed targets for *each* indirect call of the executable.

[14] We did not compare with another coarse-grained CFI scheme CCFIR [104] (which has similar CFI policy hence similar protection strength) because it is specific to Windows, and relies on relocation information that is not available for Linux binaries. In addition, we leave out comparison with the other fine-grained CFI [94], because it only works on x86-64, and is orthogonal to our approach.

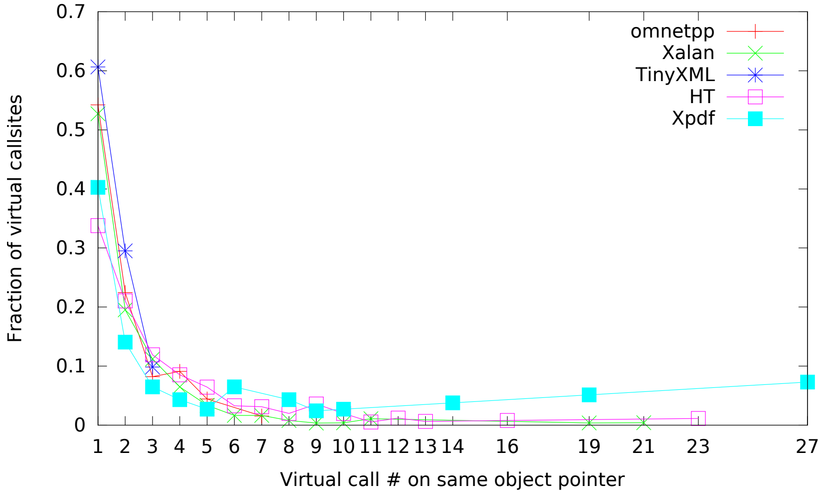


Figure 6.8: Distribution of virtual callsites based on number of identified virtual calls on the same object pointer

Column `IFunc` is the number of indirectly reachable functions we have identified. By enforcing indirect calls to only reach these targets, we can achieve a reduction of 66% (geomean). The next column `Ours` shows the median[15] number of targets that an indirect call can reach, based on our argument region checking policy. As the next column shows, this further reduces the targets by 46% (geomean), and brings an overall reduction of 82%, as compared to BinCFI.

Since the CFI precision for C++ virtual calls depends on the number of virtual callsites identified to be invoked on the same object pointer, we first show this distribution for a set of programs in Figure 6.8. As presented in the figure, a significant fraction of virtual callsites share object pointer with one or more others. And on the same object pointer, up to 27 different virtual calls are invoked.

For the same set of programs, Figure 6.9 shows the reduction of virtual call targets for our C++-specific policy as compared with vfGuard. The `vfGuard`, `Conjugate` and `Conjugate&Type` columns show the median number of allowed targets for each policy. “Conjugate” is the policy that propagates the offset constraint (as in vfGuard) to vcalls that share the same object pointer. As presented, about 40% of targets can be removed (geomean). The reduction stems from the combination of the “Conjugate” policy and “Type” policy,

[15] Each call site can reach different number of targets, hence we statistically representative number is needed.

File	**vfGuard**	**+SameObj**	**+Type+SameObj**	Reduction
omnetpp	20	20	13	35.0%
Xalan	309	220	155	49.8%
TinyXML	6	6	3	50.0%
HT Editor	34	32	24	29.4%
Xpdf	38	38	24	36.8%
geomean	**34**	**32**	**20**	**41.2%**

Figure 6.9: Virtual call targets reduction of our CFI policy

and “Conjugate” alone does not lead to much reduction. Note the last column “Conjugate vcalls” indicate the percentage of virtual calls that are found to be invoked on the same objects as some other virtual calls. For each number, this correponds to the sum of all y-axis values except x = 1 for each program in Figure 6.8.

6.5 Security Analysis

In this section, we evaluate the effectiveness of our approach against advanced code reuse attacks.

6.5.1 Coarse-Grained CFI Bypass Attacks

It has been shown that coarse-grained CFI is bypassible [47, 32]. In this section, we analyze how our approach can block such attacks, with the proof-of-concept example from the original paper [47].

The original exploit has several phases. Because the vulnerability could be used to corrupt a function pointer, the first phase of the attack is to transfer control to a return instruction. The second phase pivots the stack and leverages an advanced ROP chain that only uses coarse-grained CFI conformant gadgets, and finally a system call is invoked to change memory permission – leading to easy code injection attacks. Next we focus on the first phase of the attack, and show that the exploit could be defeated even without backward edge protection.

Note that the first phase of the attack only uses gadgets that begin from valid function entry points, in order to bypass coarse-grained CFI protection. However, one key step is to make the indirect callsite (incorrectly) target a function that takes less arguments. Figure 6.10 shows the gadget that performs this task. Note that our analysis detects line 5 actually is a callee save instruction and hence the indirect call at line 6 takes 0 arguments. Moreover,

```
mov %edi, %edi
push %esi
mov %ecx, %esi
mov (%esi), %eax
push %edi ; actually callee save; but accessed as an argument by the function called next line
call 0xc4(%eax)
```

Figure 6.10: A gadget in the PoC exploit

because the callee takes at least one argument (and more than the callsite), this control flow is blocked by our CFI policy, and the attack is defeated.

6.5.2 COOP Attacks

An end-to-end COOP attack can be divided into two stages. In the first stage, an attacker exploits a vulnerability of a program, and uses the obtained memory write capability to prepare a set of counterfeit objects, and also redirects control flow to the main loop [83]. In the second stage, the main loop dispatches vfgadgets based on the counterfeit objects. Note that although a COOP variant can replace the main loop with recursion [29], since conceptually the same goal is achieved, we universally use “main loop” to refer to the sequential dispatches of prepared vfgadgets.

Next, we discuss how COOP is mitigated in each of the two stages. We first focus on the effects of VTable offset constraints and function type constraints for vfgadget dispatches *inside* the main loop. And then explain how our CFI policy prevents the main loop from being reached in the first place, with an even higher precision.

Inside the main loop. The original COOP attack [83] introduced two variants for abusing vfgadgets (i.e., virtual functions gadgets). The first variant can utilize a fake VTable inside the read-only section, while the second variant is restricted with valid VTables. Although the first variant can bypass coarse-grained VTable protection schemes [46, 103], it is stopped by a stricter policy, such as the one from vfGuard [77]. The second variant is stealthier, however, since only valid VTables can be used, the available vfgadgets are more limited.

As discussed in the original COOP paper, the vfGuard policy will be effective at least for small to medium sized C++ binaries, due to the limited number of vfgadgets available. Although for larger binaries, sufficient vfgadgets may be collected, our policy mitigate this by imposing further constraints.

Specifically, the function type of the virtual callsite and callee has also to be compatible.

To enable data flows between vfgadgets, COOP uses two techniques. The first is based on the field of a counterfeit object. Specifically, one vfgadget writes to an object field, while another reads from it. Note that since counterfeit objects can *overlap*, a single data field may belong to two or more objects at the same time. Therefore, it is possible that one vfgadget of class A accesses some object field of class B, even if they are completely not related.

The second technique is based on *implicit* data flows through argument registers. For example, vfgadget f is called before vfgadget g. If f modifies some scratch registers as a side effect of its invocation, and these registers happen to be the ones used for argument passing (such as `rcx` and `rdx`), then g can be called with attacker intended arguments.

TypeArmor [94] has shown that all existing COOP attacks can be stopped due to their restrictions on invalid data flows through register arguments. For architectures that mainly rely on stack for argument passing, our technique on function type enforcement provides similar protection. Therefore, for COOP to be evasive, the second technique for data flow cannot be used.

Base on our policy, to abuse a virtual call, attackers are left with vfgadgets whose offset in a VTable conforms with the callsite, and whose signature is compatible. These constraints make finding vfgadgets with desired semantics and passing information between each other much more difficult, if not impossible, therefore effectively mitigate COOP.

Before the main loop. As discussed, attacker needs to first hijack control flow to reach the main loop. In the context of a C++ program, a virtual call would be the common choice for abuse. Therefore, the VTable offset constraints and function type constraints play the same role in confining this virtual call. However, our CFI policy based on "same object pointer" analysis can place further restrictions.

In the same example as shown in Figure 6.5, suppose attacker uses the first virtual call to reach the main loop gadget, which is pointed by the entry at offset 4 of VTable C. If only VTable offset constraints and function type constraints were enforced, attacker would successfully redirect control to the main loop. However, our CFI policy actually detects the two virtual calls are based on the same object pointer, and constraints on the second virtual call is propagated to the first. Therefore, that edge is prevented. As shown in this example, our CFI policy protects the control transfer to the main loop with even higher precision.

7 Conclusions and Future Work

Function recovery for COTS binaries is a comprehensive and challenging problem. There are many tasks involved: from the very basic function boundary identification, to high level constructs and property recovery, and until full function decompilation. Accuracy of function recovery is the key for enabling many applications, and it requires deep analysis of function properties and constructs.

In this dissertation, we started from accurately recovering function calls and returns. Although seemingly straight-forward, call and return detection cannot only rely on instruction syntax, due to complications such as low level code and hand-written assembly. We therefore developed a static analysis to accurately infer function call and return intentions. With this information, backward control flow edges can be protected with improved compatibility as well as precision.

We then extended our analysis for function recognition. A more comprehensive scheme has been used to accurately reason about function entry and exit points. To that end, We develped effective analysis techniques to check control flow and data flow properties associated with a function interface. Our system not only significantly outperforms existing machine-learning and static analysis based approaches, but they are more amenable for demanding analysis and instrumentation applications.

Function type defines how functions interact with each other and its recovery can be used for many purposes. Our analysis focused on deriving stack arguments for both indirect callees and callsites. We utilized such information to enforce more fine-grained forward-edge CFI policies for both normal indirect calls and C++ virtual calls.

Different from general reverse engineering, our function recovery is designed to support a wide range of further analysis, and facilitate binary hardening with instrumentations. Using backward and forward edge CFI as examples, we demonstrated that robust and precise protection for COTS binaries can be enabled with deep static analysis of functions.

In future, we plan to incorporate other static analysis techniques for accurate function recovery. One direction is to leverage type inference for function argument analysis. The specific types of arguments, in addition to the count of arguments, will further improve CFI precision as well as enable other applications.

References

[1] DWARF debugging information format version 4. http://www.dwarfstd.org/doc/DWARF4.pdf.

[2] Hex rays. https://www.hex-rays.com/index.shtml.

[3] HT editor 2.0.20 - buffer overflow (ROP PoC). https://www.exploit-db.com/exploits/22683/.

[4] Itanium C++ ABI. https://mentorembedded.github.io/cxx-abi/abi.html.

[5] Representation of call sites in the debugging information. http://www.dwarfstd.org/ShowIssue.php?issue=100909.2&type=open.

[6] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *CCS*, 2005.

[7] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM TISSEC*, 2009.

[8] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools, 2nd Edition*. Addison Wesley, 2006.

[9] P. Akritidis, M. Costa, M. Castro, and S. Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security*, 2009.

[10] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *ACM EuroSys*, 2013.

[11] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security*, 2016.

[12] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *Euro S&P*, 2017.

[13] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *EuroS&P*, 2017.

[14] G. Balakrishnan and T. Reps. WYSINWYX: What you see is not what you eXecute. *ACM TOPLAS*, 2010.

[15] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. Byteweight: Learning to recognize functions in binary code. In *USENIX Security*, 2014.

[16] D. Bounov and S. Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *NDSS*, 2016.

[17] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *Computer aided verification*, 2011.

[18] N. Burow, S. A. Carr, S. Brunthaler, M. Payer, J. Nash, P. Larsen, and M. Franz. Control-flow integrity: Precision, security, and performance. *arXiv preprint arXiv:1602.04056*, 2016.

[19] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security*, 2015.

[20] N. Carlini and D. Wagner. ROP is still dangerous: Breaking modern defenses. In *USENIX Security*, 2014.

[21] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan. Bingo: Cross-architecture cross-os binary search. In *FSE*, 2016.

[22] X. Chen, A. Slowinska, D. Andriesse, H. Bos, and C. Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[23] Y. Chen, D. Zhang, R. Wang, R. Qiao, A. Azab, L. Lu, H. Vijayakumar, and W. Shen. Norax: Enabling execute-only memory for COTS binaries on AArch64. In *S&P*, 2017.

[24] T. Chiueh and F. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *ICDCS*, 2001.

[25] T. Chiueh and M. Prasad. A binary rewriting defense against stack based overflows. In *USENIX ATC*, 2003.

[26] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *IEEE International Workshop on Program Comprehension*, 1999.

[27] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, M. Negro, M. Qunaibit, and A. Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *CCS*, 2015.

[28] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[29] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a trap: Table randomization and protection against function-reuse attacks. In *CCS*, 2015.

[30] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. 1991.

[31] T. H. Dang, P. Maniatis, and D. Wagner. The performance cost of shadow stacks and canaries. In *ASIACCS*, 2015.

[32] L. Davi, A. Sadeghi, D. Lehmann, and F. Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security*, 2014.

[33] L. Davi, A. Sadeghi, and M. Winandy. ROPdefender: A detection tool to defend against return-oriented programming attacks. In *ASIACCS*, 2011.

[34] Y. David, N. Partush, and E. Yahav. Statistical similarity of binaries. In *PLDI*, 2016.

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

[36] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security*, 2014.

[37] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *PLDI*, 2013.

[38] M. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Working Conference on Reverse Engineering*, 2004.

[39] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *CCS*, 2015.

[40] H. Flake. Structural comparison of executable objects. In *DIMVA*, 2004.

[41] A. Fog. Calling conventions for different c++ compilers and operating systems, 2015.

[42] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. Smartdec: Approaching c++ decompilation. In *Reverse Engineering (WCRE), Working Conference on*, 2011.

[43] A. Fokin, K. Troshina, and A. Chernov. Reconstruction of class hierarchies for decompilation of c++ programs. In *Software Maintenance and Reengineering (CSMR), Conference on*, 2010.

[44] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *USENIX Security*, 2001.

[45] Y. Fu, J. Rhee, Z. Lin, Z. Li, H. Zhang, and G. Jiang. Detecting stack layout corruptions with robust stack unwinding. In *RAID*, 2016.

[46] R. Gawlik and T. Holz. Towards automated integrity protection of C++ virtual function tables in binary programs. In *ACSAC*, 2014.

[47] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out of Control: Overcoming control-flow integrity. In *IEEE S&P*, 2014.

[48] E. Göktaş, E. Athanasopoulos, M. Polychronakis, H. Bos, and G. Portokalidis. Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In *USENIX Security*, 2014.

[49] I. Guilfanov. Decompilers and beyond. *Black Hat USA*, 2008.

[50] L. C. Harris and B. P. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 2005.

[51] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *CGO*, 2012.

[52] N. Hasabnis, R. Qiao, and R. Sekar. Checking correctness of code generator architecture specifications. In *CGO*, 2015.

[53] N. Hasabnis and R. Sekar. Automatic generation of assembly to IR translators using compilers. In *AMAS-BT*, 2015.

[54] N. Hasabnis and R. Sekar. Extracting instruction semantics via symbolic execution of code generators. In *International Symposium on Foundations of Software Engineering*, 2016.

[55] N. Hasabnis and R. Sekar. Lifting assembly to intermediate representation: A novel approach leveraging compilers. In *ASPLOS*, 2016.

[56] Intel. Intel 64 and IA-32 architectures software developers manual.

[57] D. Jang, Z. Tatlock, and S. Lerner. SafeDispatch: Securing C++ virtual calls from memory corruption attacks. In *NDSS*, 2014.

[58] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan. Recovering c++ objects from binaries using inter-procedural data-flow analysis. In *Program Protection and Reverse Engineering Workshop*, 2014.

[59] R. W. M. Jones, P. H. J. Kelly, M. C, and U. Errors. Backwards-compatible bounds checking for arrays and pointers in C programs. In *AADEBUG*, 1997.

[60] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification*, 2008.

[61] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*, 2014.

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

[63] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *NDSS*, 2010.

[64] longld. Payload already inside: Data reuse for ROP exploits. https://media.blackhat.com/bh-us-10/whitepapers/Le/BlackHat-USA-2010-Le-Paper-Payload-already-inside-data-reuse-for-ROP-exploits-wp.pdf.

[65] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[66] A. J. Mashtizadeh, A. Bittau, D. Mazieres, and D. Boneh. CCFI: Cryptographically enforced control flow integrity. In *CCS*, 2015.

[67] X. Meng and B. P. Miller. Binary code is not easy. In *ISSTA*, 2016.

[68] V. Mohan, P. Larsen, S. Brunthaler, K. Hamlen, and M. Franz. Opaque control-flow integrity. In *NDSS*, 2015.

[69] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *PLDI*, 2009.

[70] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[71] B. Niu and G. Tan. Modular control-flow integrity. In *PLDI*, 2014.

[72] B. Niu and G. Tan. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *CCS*, 2014.

[73] B. Niu and G. Tan. Per-input control-flow integrity. In *CCS*, 2015.

[74] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida. Marx: Uncovering class hierarchies in c++ programs. In *NDSS*, 2017.

[75] M. Payer, A. Barresi, and T. R. Gross. Fine-grained control-flow integrity through binary hardening. In *DIMVA*, 2015.

[76] M. Payer, T. Hartmann, and T. R. Gross. Safe loading - a foundation for secure execution of untrusted programs. In *IEEE S&P*, 2012.

[77] A. Prakash, X. Hu, and H. Yin. vfGuard: Strict protection for virtual function calls in COTS C++ binaries. In *NDSS*, 2015.

[78] R. Qiao and R. Sekar. Effective function recovery for COTS binaries using interface verification. Technical report, Secure Systems Lab, Stony Brook University, 2016.

[79] R. Qiao and R. Sekar. Function interface analysis: A principled approach for function recognition in COTS binaries. In *DSN*, 2017.

[80] R. Qiao, M. Zhang, and R. Sekar. A principled approach for ROP defense. In *ACSAC*, 2015.

[81] N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt. Learning to analyze binary computer code. In *AAAI*, 2008.

[82] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *CGO*, 2008.

[83] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A. Sadeghi, and T. Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *IEEE S&P*, 2015.

[84] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Usenix Security*, 2013.

[85] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Working Conference on Reverse Engineering*, 2002.

[86] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[87] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security*, 2015.

[88] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. (state of) the art of war: Offensive techniques in binary analysis. In *IEEE S&P*, 2016.

[89] 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 *WCRE*, 2013.

[90] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security. Keynote paper.*, 2008.

[91] L. Szekeres, M. Payer, T. Wei, and R. Sekar. Eternal war in memory. *S&P Magazine*, 2014.

[92] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *USENIX Security*, 2014.

[93] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida. Practical context-sensitive CFI. In *CCS*, 2015.

[94] V. van der Veen, E. Göktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *IEEE S&P*, 2016.

[95] M. J. Van Emmerik. *Static single assignment for decompilation*. PhD thesis, The University of Queensland, 2007.

[96] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *USENIX Security*, 2015.

[97] W. Xu, D. C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *FSE*, 2004.

[98] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*, 2015.

[99] K. Yoo and R. Barua. Recovery of object oriented features from c++ binaries. In *Asia-Pacific Software Engineering Conference (APSEC)*, 2014.

[100] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen. PAriCheck: an efficient pointer arithmetic checker for C programs. In *ASIACCS*, 2010.

[101] B. Zeng, G. Tan, and G. Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *CCS*, 2011.

[102] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song. Vtrust: Regaining trust on virtual calls. In *NDSS*, 2016.

[103] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. VTint: Defending virtual function tables' integrity. In *NDSS*, 2015.

[104] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou. Practical control flow integrity and randomization for binary executables. In *IEEE S&P*, 2013.

[105] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *VEE*, 2014.

[106] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security*, 2013.

[107] M. Zhang and R. Sekar. Control flow and code integrity for COTS binaries: An effective defense against real-world ROP attacks. In *ACSAC*, 2015.