ANALYZING AND SECURING BINARIES THROUGH STATIC DISASSEMBLY

VRIJE UNIVERSITEIT

ANALYZING AND SECURING BINARIES THROUGH STATIC DISASSEMBLY

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op vrijdag 23 juni 2017 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Dennis Adriaan Andriesse

geboren te Heiloo, Nederland

promotor:	prof.dr.ir. H.J. Bos
copromotor:	dr. J.M. Słowińska

For Noortje

If you try and take a cat apart to see how it works,
the first thing you have on your hands is a non-working cat.

Douglas Adams

ISBN 978-90-5383-269-1

Acknowledgements

Let me begin by expressing my deep gratitude to Herbert Bos. I know everybody says this, but I truly could not have wished for a better advisor. Not only did Herbert give me immense freedom in pursuing my own ideas, but he supported me even when I was torn between completing my Ph.D. and pursuing a flying career with the Air Force. Although leaving the Air Force was one of the toughest decisions of my life, I don't regret for a second that I finally chose to finish my Ph.D. with Herbert.

I am equally grateful to my copromotor, Asia Slowinska. Even after she left the university, she still took the time to meet with me and provide detailed feedback on my papers and ideas. Asia is a great researcher, and never failed to help me improve my work considerably.

Next, I want to thank the members of my thesis committee: Andy Tanenbaum, Thorsten Holz, Frank Piessens, Christian Rossow, and Davide Balzarotti. Thank you all for dedicating your time to my thesis.

During my Ph.D. I was fortunate enough to intern with Cisco, where I worked on CFI techniques for embedded systems. Thanks to Jim Warren, Steve Rich and Sape Mullender for battling through countless bureaucratic obstacles to make it happen.

I would also like to (again) thank Christian Rossow for advising me during my M.Sc. thesis, which led to the opportunity to start my Ph.D. I enjoyed all our subsequent projects, especially our work on Gameover Zeus.

All other colleagues with whom I collaborated on projects also deserve special thanks: Victor van der Veen, Enes Göktaş, Xi Chen, Ben Gras, Lionel Sambuc, Cristiano Giuffrida, Tillmann Werner, Brett Stone-Gross, Daniel Plohmann, Christian J. Dietrich, and Manolis Stamatogiannakis. While I haven't collaborated directly with all my co-workers (they are so numerous these days that I won't even attempt to enumerate them), they were all great colleagues.

Thanks also goes to my parents and the rest of my family, and I apologize for all the things I disassembled when I was young. Finally, I am of course incredibly grateful to my wife Noortje and my son Sietse, who remind me that other things, besides research, are important too.

Dennis Andriesse
March 2017

Contents

List of Figures

List of Tables

List of Listings

Publications

Parts of this thesis have been published before. The corresponding publications are listed below.

Dennis Andriesse, Asia Slowinska, and Herbert Bos. "Compiler-Agnostic Function Detection in Binaries". In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, EuroS&P'17, 2017.

Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries". In *Proceedings of the 25th USENIX Security Symposium*, USENIX Sec'16, 2016.

Dennis Andriesse, Victor van der Veen (*joint first author*), Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. "Practical Context-Sensitive CFI". In *Proceedings of the 22nd Conference on Computer and Communications Security*, CCS'15, 2015.

Dennis Andriesse, Herbert Bos, and Asia Slowinska. "Parallax: Implicit Code Integrity Verification Using Return-Oriented Programming". In *Proceedings of the 45th Conference on Dependable Systems and Networks*, DSN'15, 2015.

Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. "StackArmor: Comprehensive Protection from Stack-Based Memory Error Vulnerabilities for Binaries". In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'15, 2015.

The following publications were produced during the course of the research that led to this thesis, but are not included in the thesis itself.

Dennis Andriesse, Christian Rossow, and Herbert Bos. "Reliable Recon in Adversarial Peer-to-Peer Botnets". In *Proceedings of the 15th Internet Measurement Conference*, IMC'15, 2015.

Dennis Andriesse and Herbert Bos. "Instruction-Level Steganography for Covert Trigger-Based Malware". In *Proceedings of the 11th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, DIMVA'14, 2014.

Dennis Andriesse, Christian Rossow, Brett Stone-Gross, Daniel Plohmann, and Herbert Bos. "Highly Resilient Peer-to-Peer Botnets Are Here: An Analysis of Gameover Zeus". In *Proceedings of the 8th International Conference on Malicious and Unwanted Software*, MALWARE'13, 2013.

Christian Rossow, Dennis Andriesse, Tillmann Werner, Brett Stone-Gross, Daniel Plohmann, Christian Dietrich, and Herbert Bos. "P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets". In *Proceedings of the 34th IEEE Symposium on Security and Privacy*, S&P'13, 2013.

Chapter 1

Introduction

Disassembly is the backbone of binary analysis, fulfilling the basic need to identify code in binaries, and translate it into a form fit for human analysis or further processing. Without it, we would have no way to efficiently analyze malicious programs, leaving us without means to study the details of their inner workings and devise defenses. Nor could we hope to statically rewrite legacy or proprietary binaries to improve their security characteristics, or to scan for and fix vulnerabilities in binaries. Binaries for which we have no source could never be analyzed or modified, and would forever remain vulnerable to attacks discovered after the original time of compilation. Clearly, effective and precise disassembly is a highly desirable tool for any low-level systems security work.

1.1 Motivation and Problem Statement

Despite the fact that every Instruction Set Architecture (ISA) has a well-defined set of instruction opcodes, parsing these opcodes and corresponding mnemonics from the bytes in a binary is not a straightforward process. This is illustrated in Listing 1.1, which shows some of the challenges involved in disassembly.

Listing 1.1 shows how a simple function from `opensshd` v7.1p2 is compiled by `gcc` 5.1.1 from C to x86-64 code (x64 for short). Note that the function does nothing special. It uses a `for` loop to iterate over an array, applying a `switch` statement in each iteration to determine what to do with the current array element: skip uninteresting elements, return the index of an element which meets the criteria, or print an error and exit if something unexpected happens. Despite the simplicity of the C code, the compiled x64 version of this function (shown on the right side of the listing) is far from trivial to disassemble correctly.

One of the first things to notice is that the x64 implementation of the switch statement is based on a *jump table*, a construct commonly emitted by modern compilers. The jump table implementation avoids the need for a complicated tangle of conditional jumps. Instead, the instruction at address `0x4438f9` uses the switch

```
int
channel_find_open(void) {
  u_int i;
  Channel *c;

  for(i = 0; i < n_channels; i++) {
    c = channels[i];
    if(!c || c->remote_id < 0)
      continue;
    switch(c->type) {
    case SSH_CHANNEL_CLOSED:
    case SSH_CHANNEL_DYNAMIC:
    case SSH_CHANNEL_X11_LISTENER:
    case SSH_CHANNEL_PORT_LISTENER:
    case SSH_CHANNEL_RPORT_LISTENER:
    case SSH_CHANNEL_MUX_LISTENER:
    case SSH_CHANNEL_MUX_CLIENT:
    case SSH_CHANNEL_OPENING:
    case SSH_CHANNEL_CONNECTING:
    case SSH_CHANNEL_ZOMBIE:
    case SSH_CHANNEL_ABANDONED:
    case SSH_CHANNEL_UNIX_LISTENER:
    case SSH_CHANNEL_RUNIX_LISTENER:
      continue;
    case SSH_CHANNEL_LARVAL:
    case SSH_CHANNEL_AUTH_SOCKET:
    case SSH_CHANNEL_OPEN:
    case SSH_CHANNEL_X11_OPEN:
      return i;
    case SSH_CHANNEL_INPUT_DRAINING:
    case SSH_CHANNEL_OUTPUT_DRAINING:
      if(!compat13)
        fatal(/* ... */);
      return i;
    default:
      fatal(/* ... */);
    }
  }
  return -1;
}
```

```
<channel_find_open>:
4438ae: push rbp
4438af: mov  rbp,rsp
4438b2: sub  rsp,0x10
4438b6: mov  DWORD PTR [rbp-0xc],0x0
4438bd: jmp  443945
4438c2: mov  rax,[rip+0x2913a7]
4438c9: mov  edx,[rbp-0xc]
4438cc: shl  rdx,0x3
4438d0: add  rax,rdx
4438d3: mov  rax,[rax]
4438d6: mov  [rbp-0x8],rax
4438da: cmp  QWORD PTR [rbp-0x8],0x0
4438df: je   44393d
4438e1: mov  rax,[rbp-0x8]
4438e5: mov  eax,[rax+0x8]
4438e8: test eax,eax
4438ea: js   44393d
4438ec: mov  rax,[rbp-0x8]
4438f0: mov  eax,[rax]
4438f2: cmp  eax,0x13
4438f5: ja   443926
4438f7: mov  eax,eax
4438f9: mov  rax,[rax*8+0x49e840]
443901: jmp  rax
443903: mov  eax,[rbp-0xc]
443906: leave
443907: ret
443908: mov  eax,[rip+0x2913c6]
44390e: test eax,eax
443910: jne  443921
443912: mov  edi,0x49e732
443917: mov  eax,0x0
44391c: call [fatal]
443921: mov  eax,[rbp-0xc]
443924: leave
443925: ret
443926: mov  rax,[rbp-0x8]
44392a: mov  eax,[rax]
44392c: mov  esi,eax
44392e: mov  edi,0x49e818
443933: mov  eax,0x0
443938: call [fatal]
44393d: nop
44393e: jmp  443941
443940: nop
443941: add  DWORD PTR [rbp-0xc],0x1
443945: mov  eax,[rip+0x29132d]
44394b: cmp  [rbp-0xc],eax
44394e: jb   4438c2
443954: mov  eax,0xffffffff
443959: leave
44395a: ret
```

Listing 1.1: Example of disassembled switch statement (from `opensshd` v7.1p2 compiled with `gcc` 5.1.1 for x64, source edited for brevity). Interesting lines are shaded.

input value to compute (in `rax`) an index into a table, which stores at that index the address of the appropriate case block. This way, we only require a single indirect jump (at address `0x443901` in the listing) to transfer control to any case address the jump table defines.

While efficient, jump tables make disassembly more difficult because they use *indirect control flow*. The lack of an explicit target address in an indirect jump makes

it difficult for disassemblers to track the flow of instructions past this point. As a result, any instructions targeted by an indirect jump remain undiscovered unless specific (compiler-dependent) heuristics are implemented in the disassembler to discover and parse jump tables. For our example, this means that the instructions at addresses `0x443903`–`0x443925` are not discovered without such heuristics. (As discussed in Chapter 2.4, alternative disassembly methods which are control-flow independent come with their own set of drawbacks.)

Matters are complicated even more because there are multiple `ret` instructions in the switch, as well as calls to the `fatal` function, which throws an error and never returns. In general, it is not safe to assume that there are instructions following a `ret` instruction or non-returning `call`; instead, these may be followed by data or padding bytes not intended to be parsed as code. However, the converse assumption, that these instructions are *not* followed by more code, may lead the disassembler to miss instructions, leading to an incomplete disassembly.

This example has demonstrated some of the challenges involved in disassembly. As we discuss in Chapter 2.7, many more challenges exist. This is especially true if, as we do in this thesis, we consider not only the problem of disassembling raw instructions, but also related problems such as function detection, which enable more powerful binary analysis applications based on raw disassembly. In the remainder of this thesis, we use the term "disassembly" in the broadest sense, including these related problems, as described in more detail in Chapter 2.

Disassembly is the basis of virtually all areas of binary analysis. A prominent example is malware analysis, where disassembly is used to study the behavior of malicious software such as botnets, in order to devise mitigation methods. Moreover, disassembly is crucial for binary-level anti-exploitation and vulnerability analysis systems [54; 140]. For instance, recent years have seen a vast amount of research on binary-level Control-Flow Integrity methods, which rely on disassembly to determine legal control flows for binaries, and to statically instrument binaries with enhanced security features [23; 144; 197]. Such binary-level systems are crucial for securing untrusted or proprietary binaries for which source is not available. Disassembly also finds a myriad of applications outside of security research; for instance, in binary reoptimization systems [108; 123; 179].

Summarizing, we have established that disassembly is both challenging, and an important tool in low-level systems security. This leads us to ask: How can we minimize the inaccuracies and challenges involved in disassembly, so that we can safely use it to build secure systems? This is the overarching subject of this thesis.

1.2 Research Questions

The problem of how to apply potentially inaccurate disassembly to safely build secure systems is the theme of Part I. Phrasing the issue more specifically gives rise to our first research question:

Question (1): *Given all the potential disassembly inaccuracies, how can we effectively and safely apply binary analysis to build systems for analyzing and securing legacy and proprietary binaries?*

We study solutions to this question by building several practical systems for securing binaries, based on disassembly and binary rewriting. In these systems (listed in Section 1.3), we investigate multiple techniques for maximizing security and reliability guarantees while tolerating potential inaccuracies in the disassembly and binary rewriting process.

During the course of investigating Question (1), we developed a rigid understanding of the sort of output and precision we require from disassembly to achieve our goals. However, during the publication process of our papers, we also frequently encountered reviewers who were uncertain about what exactly is and is not feasible with binary analysis, and what problems one should expect with a given approach. This frequently led to questions such as: "Can you guarantee that all functions will be instrumented?", or: "How likely is it that your approach might break protected binaries, or cause them to crash?".

Given the error-prone nature of binary analysis, these questions are justified. One should naturally take precautions to ensure maximum accuracy and graceful failure in the event of inaccuracies. However, we also found a distinct lack of agreement in the research community about just how reliable disassembly is, and which precautions are needed in a particular situation. This lack of agreement frequently leads to disagreements among both reviewers and researchers about the merits of a given binary analysis-based system. This led us to pursue our next research questions, which we explore in Part II of this thesis:

Question (2): *How precise is disassembly (in the broadest sense) in practice, and how frequently should we expect inaccuracies of each possible kind?*

Question (3): *To what extent is there a consensus on disassembly precision in the binary analysis community, and where is that consensus mismatched with our findings from Question (2)?*

Our work shows that, in many situations, disassembly performs better than is commonly assumed in the literature (see Chapter 6). However, another major conclusion from our work on Questions (2) and (3) is that function detection is highly problematic as a disassembly primitive (more so than any other primitive), frequently yielding false positive and false negative rates in excess of 20%. Our final research question, also addressed in Part II, is therefore:

Question (4): *How can we achieve more precise function detection results?*

In our pursuit of this research question, we develop a novel function detection method which provides more accurate results than existing systems, while making fewer assumptions on the properties of the analyzed binary.

1.3 Contributions and Outline

The remainder of this thesis begins by providing the necessary background in Chapter 2. Subsequently, we discuss our contributions in the following two topics.

(1) Binary protection solutions for stripped legacy and proprietary binaries (Part I of this thesis).

- We develop *StackArmor*, a comprehensive system to protect binaries against stack-based memory error vulnerabilities, including both inter- and intra-frame defenses. We describe *StackArmor* in Chapter 3.
- In Chapter 4, we introduce *PathArmor*, the first practical context-sensitive Control-Flow Integrity implementation. *PathArmor* is open-source, and provides strong protection for binaries against a range of control-flow hijacking attacks.
- We implement *Parallax*, a novel stand-alone tamperproofing system for binaries. *Parallax* is the first system which can protect arbitrary code while withstanding instruction cache modification attacks that affect prior work. It is discussed in Chapter 5.

(2) Quantitative measurements and improvements of disassembly precision (Part II of this thesis).

- We perform a large-scale study of disassembly on real-world x86 and x64 binaries. This study aims to fill the void in the binary analysis community caused by a lack of consensus on the true precision and problems in disassembly. We measure all relevant aspects of disassembly, and publicly release our results and ground truth to facilitate future studies. Moreover, we compare our findings to three years worth of binary analysis work published in top venues, and pinpoint where the expectations of this research are mismatched with our findings. We discuss our results in Chapter 6.
- Given that function detection is currently the most inaccurate disassembly primitive, we develop a novel function detection system called *Nucleus*. *Nucleus* takes a Control-Flow Graph-centric approach, which is both compiler-agnostic, and more accurate than existing signature-based work. *Nucleus* is open-source, and is discussed in Chapter 7.

We begin each part with a brief outline of what will be discussed, and conclude each part with a discussion of lessons learned. Chapter 8 concludes this thesis, recapitulating our main results and discussing perspectives for future work.

Chapter 2

Background

Disassembly is a broad subject, which encompasses more than just raw instruction recovery. Moreover, the difficulty and accuracy of disassembly depends on multiple factors, such as the availability of symbols and the presence of complex binary-level constructs. This chapter provides the necessary background to understand the tradeoffs involved, and provides our definitions of disassembly and other terms used throughout this thesis.

2.1 Binary Analysis Primitives

This thesis considers disassembly in its broadest sense: in addition to finding and parsing instructions, we also want to recover and analyze more complex primitives useful for binary analysis. This section defines and describes the primitives that we are interested in throughout this thesis.

Instruction The basic mnemonic representation of a machine-level instruction, as shown on the right side of Listing 1.1.

Basic Block (BB) A sequence of instructions, where the first instruction is the only entry point (the only instruction targeted by any jump in the binary), and the last instruction is the only exit point (the only instruction in the sequence which may jump to another basic block) [16]. Some disassemblers consider call instructions to be exit points, while others do not. Except where otherwise noted, we do not consider calls to be exit points in this thesis.

Function A set of basic blocks, such that together these implement the functionality defined in a corresponding source-level function (as in Listing 1.1), possibly merged with a set of basic blocks from inlined functions. Instead of recovering all basic blocks associated with a function, disassemblers may report only the first address (*function start* detection), or report both the first and last addresses (*function boundary* detection). These terms are defined more rigorously in Chapter 7.

Function signature The list of parameters and return type for a particular function.

Control Flow Graph (CFG) A digraph $G_{cfg} = (V_{bb}, E_{cf})$, which describes how control flow edges $E_{cf} \subseteq V_{bb} \times V_{bb}$ connect the basic blocks V_{bb} in a function [16]. In practice, disassemblers often omit indirect edges, or define a global CFG rather than per-function CFGs. Therefore, we also define the *Interprocedural CFG (ICFG)*, allowing us to abstract from disassemblers' varying CFG definitions.

Interprocedural CFG (ICFG) The union of all function CFGs, connected through interprocedural call and jump edges. We will sometimes define the ICFG to include indirect edges, other times these will be excluded. This will be made clear on a per-scenario basis (usually in each chapter where the term is relevant).

Callgraph A digraph $G = (V_{cs} \cup V_f, E_{call})$ linking the set V_{cs} of call sites to the function starts V_f through may-call edges $E_{call} \subseteq V_{cs} \times V_f$ [16]. Similarly to the CFG, disassemblers deviate from the traditional callgraph definition by including only direct call edges. Therefore, we use the term "callgraph" to denote only the *direct callgraph*, except where noted otherwise.

2.2 The x86/x64 Instruction Set Architecture

Our work in this thesis focuses on the x86 instruction set, and its 64-bit version called x86-64 (or x64). Except where stated otherwise, this thesis assumes stripped compiler-generated x86/x64 binaries based on C/C++ source.

The x86/x64 ISA is interesting because it is incredibly common in both the consumer market (especially in desktop computers) and in binary analysis research (in part due to its popularity in end-user machines). This alone makes x86/x64 an obvious platform for our research. However, several other properties of the ISA make it even more suitable for study, in large part because of its complexity and the special challenges it poses to disassemblers.

The x86 ISA is a Complex Instruction Set Computing (CISC) architecture with a long history of backwards compatibility (dating back to 1978), leading to a very dense instruction set, in the sense that the vast majority of possible byte values represent a valid opcode [100; 142]. This exacerbates the code versus data problem, making it less obvious to disassemblers that they have mistakenly interpreted data as code. Moreover, the instruction set is variable-length, and allows unaligned memory accesses for all valid word sizes. Thus, x86/x64 allows unique complex binary constructs, such as (partially) overlapping and misaligned instructions.

2.3 Static versus Dynamic Analysis

Broadly speaking, binary analysis can be divided into two main philosophies: static analysis, and dynamic analysis. This thesis focuses mostly on static analysis (using

dynamic analysis only when the disadvantages described below can be mitigated). Both static and dynamic analysis come with their own unique tradeoffs.

Static disassembly takes a binary as input, and attempts to discover and parse all code within that binary without running it. There also exists the contrasting notion of *dynamic disassembly*. Dynamic disassembly *does* run a binary, and discovers instructions by tracing them as they are executed.

Dynamic disassembly has multiple advantages. For instance, consider the problem of resolving indirect control flow, as illustrated earlier in Listing 1.1. When tracing a program dynamically, the target of any indirect jump is immediately clear, because we can observe which instruction is executed next. For much the same reason, we never have to worry about accidentally interpreting data as code.

However, these advantages come with a price: we only see instructions that are actually executed, so that each dynamic disassembly run reveals only part of the code in a binary. Finding the rest of the code (and indeed, even knowing when we have seen all of it) is an extremely challenging problem, known as the *code coverage* problem [127]. Solutions to this problem attempt to drive execution towards unexplored paths, but ensuring that all paths are covered in a reasonable amount of time is still an unsolved problem [57].

Another disadvantage of dynamic analysis is that the runtime monitoring involved often imposes a large performance overhead on the analyzed program. In some scenarios, this may be unacceptable; for instance, if the aim is to instrument a binary with additional security guarantees in a production setting, we want to minimize overhead. Moreover, the need for an external analysis framework makes distribution of instrumented binaries less practical.

Avoiding the code coverage problem and minimizing overhead are important benefits of static analysis. Static methods promise highly complete analysis, and enable comprehensive, stand-alone, low-overhead instrumentation. Since the applications are so appealing, it is worth the effort to study how we can improve the guarantees of static analysis, or use it effectively despite potential inaccuracies.

2.4 Disassembly Methods

There are two main approaches to static disassembly: *linear* and *recursive* disassembly. These are illustrated in Figure 2.1.

Conceptually, linear disassembly is the simplest approach. It iterates through all code segments in a binary, decoding all bytes consecutively and parsing them into a list of instructions. The risk of this approach is that not all bytes may be instructions. Some compilers, such as Visual Studio, intersperse data such as jump tables with the code (see also Chapter 6). When parsing this data as code, a disassembler may encounter invalid opcodes. Worse: the data bytes may coincidentally correspond to valid opcodes, leading the disassembler to output bogus instructions (quite likely on dense ISAs such as x86/x64). On ISAs with variable-length op-

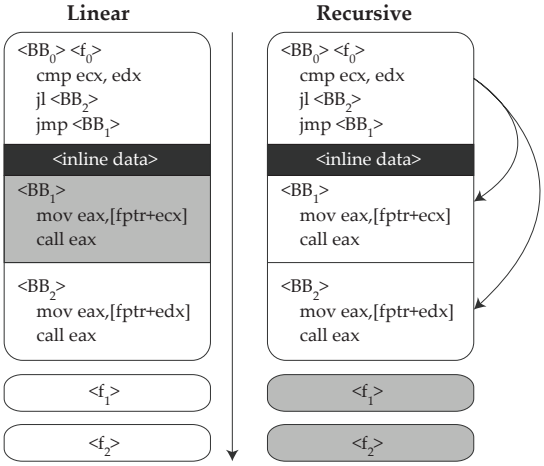


Figure 2.1: Disassembly methods. Arrows show disassembly flow. Gray blocks show missed or corrupted code.

codes, such as x86/x64, the disassembler may even become desynchronized with respect to the true instruction stream. Though the disassembler will eventually self-resynchronize [118], desynchronization can cause the first few real instructions following the data to be missed [118; 161].

Recursive disassembly avoids these problems by being control-flow sensitive. It starts from known entry points into the binary (such as the main entry point, and exported function symbols), and recursively follows control flow from there to discover code. This allows recursive disassembly to work around data bytes in all but a handful of corner cases.[1] As we have seen in Listing 1.1, the downside of this is that basic blocks (or even entire functions, such as f_1 and f_2 in Figure 2.1) targeted by indirect jumps or calls are likely to be missed, unless special (compiler-specific, and error-prone) heuristics are used to find them. Recursive disassembly is the *de facto* standard used in well-known disassemblers like IDA Pro [83] and Dyninst [31].

2.5 Binary Instrumentation

Sometimes, we not only want to analyze a binary, but also modify it. This situation is common in binary security research. For instance, we may want to extend a binary with additional checks upon each (indirect) `call` and `ret`, verifying that the destination of the control transfer has not been tampered with. Modifying a binary in this kind of way is called *binary instrumentation*. We make heavy use of it in Part I of this thesis, to implement novel binary security techniques.

[1] To maximize code coverage, recursive disassemblers typically assume that the bytes directly after a `call` instruction must also be disassembled, since they are the most likely target of an eventual `ret`. Additionally, disassemblers assume that both edges of a conditional jump target valid instructions. Both of these assumptions may be violated in rare cases, such as in deliberately obfuscated binaries.

Like binary analysis in general, binary instrumentation can follow either a static or dynamic approach, with much the same tradeoffs as discussed in Section 2.3. Dynamic instrumentation has the added advantage that it can easily inject or modify instructions as they are executed (typically in a dedicated code cache). However, the overhead involved is relatively high (slowdowns of $4\times$ or more are not uncommon, even for highly optimized platforms [121]), and distribution of instrumented binaries is not as straightforward as with static instrumentation. Well-known dynamic instrumentation platforms include Pin [101; 121], Dyninst [31] and DynamoRIO [41].

Static binary instrumentation, as the name implies, disassembles and then modifies a binary in-place, producing a new stand-alone binary that incorporates all modifications. Although this provides the benefits of low overhead and stand-alone binaries, static instrumentation is more error-prone than its dynamic counterpart.

It is typically not possible to alter code in-place, as this causes code to shift around, invalidating offsets and pointers used by calls and jumps throughout the binary. Without extensive symbolic information (see Section 2.6), it is infeasible to find and fix all affected pointers. Since we cannot rely on the availability of such information, a common alternative is to duplicate the code section, and insert *trampolines* at the start of each function in the original copy [178]. Function calls target the original copy, which (through the trampoline) immediately jumps to the modified version, in the duplicated code section. This allows us to modify the duplicated code without invalidating code pointers, as these point only to the original code. While this approach increases binary size, it induces relatively low overheads (achieving slowdowns in the order of 10%–20% for our purposes).

Clearly, any errors in the disassembly used by a static instrumentation framework lead to undefined results unless care is taken to ensure graceful failure. Part I of this thesis pursues the question of how to build reliable binary protection systems which tolerate a degree of disassembly/instrumentation errors. The static instrumentation platforms we use are PEBIL [114] and Dyninst [31] (which supports static as well as dynamic instrumentation).

2.6 Symbols, DWARF, PDB and Stripped Binaries

High-level source code (such as C code) is centered around functions and variables with meaningful, human readable names. When compiling a program, compilers emit *symbols* which keep track of such symbolic names, and describe how they are mapped into the binary. For instance, function symbols provide a mapping from symbolic, high-level function names to the first address and the size of each function. This information is normally used by the linker when combining object files, and also aids debugging.

Symbolic information can be emitted as part of the binary, or in the form of a separate symbol file, and it comes in various flavors. Basic symbols are needed by the linker, but far more extensive information can be emitted for debugging pur-

poses. Debugging symbols go as far as providing a full mapping between source lines and binary-level instructions, describing function parameters, stack frame information, and more. For ELF binaries, debugging symbols are typically generated in the DWARF format [82], while PE binaries usually use the proprietary Microsoft PDB format.

Clearly, symbolic information is extremely useful for binary analysis. To name just one example, having a set of well-defined function symbols at our disposal eliminates the challenging function detection problem, even if only basic linker symbols are available. Unfortunately, extensive debugging information is typically not included in production-ready binaries, and even basic symbolic information is often stripped to reduce file sizes and prevent reverse engineering (especially in the case of malware or proprietary software). This means that disassemblers, and by extension any work that relies on disassembly of arbitrary binaries, must be able to deal with the far more challenging scenario of stripped binaries without any form of symbolic information. Throughout this thesis, we therefore assume as little symbolic information as feasible, and we focus on stripped binaries except where noted otherwise.

2.7 Complex Constructs: Challenges for Static Analysis

In Listing 1.1, we have already introduced several challenging cases for static disassembly. We have also mentioned the existence of many more complex cases which binary analysis may encounter. This section introduces a set of complex corner cases which are often cited as being particularly harmful to the accuracy of disassembly [31; 125; 161]. These are relevant throughout this thesis. Chapter 6 in particular studies the prevalence of these constructs in real-world binaries, and quantifies their impact on disassembly precision.

Overlapping/shared basic blocks Basic blocks may be shared between functions, hindering disassemblers in distinguishing these functions from each other.

Overlapping instructions Since x86/x64 uses variable-length instructions without any enforced memory alignment, jumps can target any offset within a multi-byte instruction. This allows the same code bytes to be interpreted as multiple overlapping instructions, some of which may be missed by disassemblers.

Padding code and inline data Especially at high optimization levels, modern compilers add padding code (often `nop` instructions or `NULL` bytes) between functions, and even between basic blocks within a function. This code is not intended to be executed, but to align functions and basic blocks in memory so they can be accessed with optimal efficiency. In addition, compilers like Visual Studio intersperse data, such as switch jump tables, with code. If a disassembler mistakenly interprets data as code, this will lead to false positive instructions, and could lead to false negatives if the instruction parser becomes desynchronized from the real instruction stream.

Unreachable/indirectly reached code Recursive disassemblers like IDA Pro and Dyninst follow control flow to discover code. While this approach works well for separating code from data, it cannot discover functions or basic blocks that are never called, or which are only called or jumped to indirectly. As we have seen in Listing 1.1, switches are also a common source of indirect control flow.

Non-contiguous functions Many disassemblers assume that each function is laid out in a single contiguous memory range [69; 83]. This assumption is convenient for signature-based function detection, which works by scanning for a function prologue and epilogue. Depending on the compiler and optimization level, functions may instead consist of multiple disjoint memory ranges, which require deeper analysis to be associated with the correct function.

Multi-entry functions Instead of a single entry point, a function may have multiple alternative entry points. For instance, `glibc` defines the `splice` function, which has an alternative entry called `__splice_nocancel` that may be called depending on whether thread safety is required. Function detectors which do not consider this may misclassify each alternative entry block as a separate function.

Tail calls In this common optimization, a function ends not with a return, but with a jump to another function. This makes it more difficult to detect where the optimized function ends.

Alternative prologues/epilogues Disassemblers often scan for well-known signatures of function prologues and epilogues to detect functions. At high optimization levels, these recognizable code sequences are often missing, causing misidentification of the affected functions.

Obfuscated code In addition to complex constructs "naturally" emitted by compilers, binaries may also contain deliberately obfuscated code. There exist a myriad of obfuscation techniques, ranging from methods to confuse disassemblers by breaking assumptions (such as where calls will return), to more invasive approaches like self-modifying code. An excellent overview of obfuscation is given by Collberg and Nagra [60]. Obfuscated code occurs mostly in malware, though it has also been used in proprietary software to resist reverse engineering (Skype being a famous example [34]). Because our focus is on securing binaries which do not actively resist our efforts, most research in this thesis is not concerned with obfuscated code.

I Retrofitting Security Into Stripped Binaries

Outline

We now focus our attention on exploring methods for adding security to binaries, based on static analysis and binary rewriting. We emphasize two important properties in each of the discussed systems.

P(1): *Maximum performance and reliability, even with imperfect disassembly primitives underlying the analysis.* For instance, it is acceptable for an imprecise analysis to cause a *graceful reduction* of security guarantees, but it should never lead to a broken or crashing binary.

P(2): *A minimal need for symbolic or source-level information.* In cases where a particular binary analysis is highly unreliable, we sometimes apply a *policy-driven* approach to symbolic information: We design our systems such that they function normally in binary-only mode (and can thus be used for legacy or proprietary binaries), but can take advantage of any available symbols to improve precision.

We implement these properties in three binary protection systems, thereby studying how to effectively apply these properties in practice.

(1) Chapter 3 discusses *StackArmor*, a high-performance comprehensive stack protection approach for binaries.

(2) In Chapter 4, we introduce *PathArmor*. *PathArmor* is a Control-Flow Integrity (CFI) system, which implements defenses against control-flow hijacking that are agnostic of the underlying exploit, on both the backward and forward edge.

(3) Finally, in Chapter 5, we discuss *Parallax*, a stand-alone tamperproofing system for binaries that resists tampering attempts by a hostile user on the same system.

At the end of this part of the thesis, we provide an overview of the various binary analysis strategies explored in our systems, and discuss how they satisfy P(1) and P(2). We also discuss the tradeoffs and applicability of each approach for use in future binary analysis-based work. Part II of this thesis ties into this part by quantifying and improving the precision of the disassembly primitives we rely on. This improves (our understanding of) the precision (and thus security) guarantees we can expect from analyses based on each primitive.

Chapter 3

StackArmor: Comprehensive Protection from Stack-Based Memory Error Vulnerabilities for Binaries

StackArmor is a comprehensive protection technique for stack-based memory error vulnerabilities in binaries. It relies on binary analysis and rewriting strategies to drastically reduce the uniquely high spatial and temporal memory predictability of traditional call stack organizations. Unlike prior solutions, *StackArmor* can protect against arbitrary stack-based attacks, requires no access to the source code, and offers a policy-driven protection strategy that allows end users to tune the security-performance tradeoff according to their needs. We present an implementation of *StackArmor* for x86-64 Linux and provide a detailed experimental analysis of our prototype on popular server programs and standard benchmarks (SPEC CPU2006). Our results demonstrate that *StackArmor* offers better security than prior binary- and source-level approaches, at the cost of only modest performance and memory overhead even with full protection.

3.1 Introduction

While common defenses like W$\oplus$X, canaries, and traditional ASLR prevent naive return address overflows and code injection attacks, they have done little to eliminate stack-based attacks altogether. Mainly, the complexity of the attacks has increased as attackers resort to advanced techniques like Return-Oriented Programming (ROP) [163]. Likewise, they exploit the stack's predictable layout to disclose useful information, stored in current, previous, or reused stack frames [53]. We conclude that, despite all efforts, the stack remains a hugely attractive target for at-

tackers, mainly because it is an exploit-friendly contiguous mapping with spatial and temporal allocation locality that is *entirely* predictable—obviating even the need for "feng shui" strategies used on the heap [173].

In this chapter, we address the problem at its root by completely abandoning the idea of a linearly growing stack. We statically rewrite binaries to isolate and fully randomize the locations of stack frames and individual stack buffers, countering both spatial attacks like overflows and temporal attacks like stack-based use-after-frees.

While we provide more comprehensive protection than prior solutions, better stack defenses have been proposed before. Existing approaches include compiler extensions [18; 19], shadow stacks [33; 58; 61; 85; 145; 152; 166; 194], Control-Flow Integrity (CFI) [15; 67; 197], and binary rewriting to add buffer protection [169], but they either rely on source code and leave binaries at the mercy of attackers, or offer only very limited protection. Specifically, there is currently no stack protection technique for binaries that mitigates all of the following attack vectors: (1) buffer overwrites and overreads within a stack frame, (2) buffer overwrites and overreads across stack frames, (3) stack-based use-after-frees, (4) uninitialized reads (in reused stack frames). As a result, stack attacks are still rampant. Attackers use them both to divert control flow (e.g., in ROP attacks) and for memory disclosures [53].

Information leakage and buffer overflow attacks, in particular, are greatly helped by the predictability of the stack layout. Although the start is typically randomized, the stack itself grows in an entirely predictable fashion, making the disclosure of canaries, return addresses, or data pointers of previous stack frames as simple as leaking uninitialized data or exploiting buffer overreads. The same applies to exploits modifying data in another stack frame. For example, randomization between stack frames would have stopped recent high-profile attacks on Asterisk [72], Xen [73], Kerberos [70], and MS Office [71].

Our binary-level approach makes *StackArmor* ideally suited for many practical scenarios: the adoption of advanced security measures in popular compilers is slow. Compiler maintainers are conservative and tend to reject options that incur significant overhead. The `-fstack-protector-strong` option in `gcc` is a case in point: it was tailored to a very narrow threat model for performance reasons. As most vendors simply use common compilers like `gcc`, any measure not added to it for performance reasons will not make it into production binaries. Without binary-level defenses, users cannot decide to sacrifice some performance for better security.

Contributions We introduce *StackArmor*, a novel stack protection technique that shields binaries from all of the above attacks. To provide comprehensive protection, *StackArmor* relies on static analysis enabled by state-of-the-art binary analysis tools—which provide the necessary program abstractions, such as functions and their control-flow graphs. Our static analysis is also supported by information on the location and size of stack objects, for example provided by debug symbols (similar to prior binary-level protection techniques [85]) or dynamic reverse engineering techniques [116; 168]. *StackArmor* can also operate in complete absence of these, by

gracefully reducing its (intra-frame) protection guarantees. Using binary rewriting to instrument call and return instructions, *StackArmor* provides tailored protection based on application-specific performance and security requirements. In full protection mode, *StackArmor* relies on a combination of randomization, isolation, and secure allocation techniques to create the illusion that *all* the stack frames and the individual stack buffers are drawn from a fully randomized space with no spatial or temporal predictability guarantees. Unlike all the existing solutions, this strategy can comprehensively protect against *arbitrary* stack-based attacks.

To summarize, our contributions are as follows.

- We present *StackArmor*, a novel stack protection technique which combines inter- and intra-frame defenses to stop arbitrary spatial and temporal attacks.
- We present an implementation of *StackArmor* for x86-64 Linux. Ours is the first system that provides such comprehensive stack protection for binaries.
- We provide a detailed experimental evaluation of our prototype, and show that it achieves a modest performance and physical memory overhead of 5% and +3MB, respectively, on average, on single-threaded server programs, while scaling well even to heavily threaded server programs (28% and +112MB with 100 worker threads, on average) with full protection.

3.2 Threat Model

StackArmor prevents memory error attacks exploiting *spatial* and *temporal* locality of reference on the stack. This section briefly elaborates on both classes of attacks and discusses the limitations of existing stack protection techniques.

3.2.1 Spatial Attacks

Spatial attacks exploit memory errors to access data outside the prescribed buffer bounds. Well-known memory error examples include stack-based buffer overflows and underflows. Attackers exploit them to corrupt memory objects with malicious writes, or leak secrets through unintended reads. Attacks can target both control data, such as return addresses or function pointers, and non-control data, such as variables storing user privilege levels.

To access a target object, an attack first estimates its address and next obtains a pointer to the target location via a vulnerable buffer. It either exploits a vulnerable buffer and a target object located in the same stack frame (*intra-frame* attack) or in different ones (*inter-frame* attack). In a traditional stack organization, both stack frames and per-frame objects are contiguously allocated in memory, so the attack can safely rely on the predictability of the relative distance between the buffer and the target object.

3.2.2 Temporal Attacks

Temporal attacks exploit memory errors to access data outside the prescribed object lifetime. Such attacks rely on predictable memory reuse to read/write data from a newly allocated object via a reference to a deallocated object or, conversely, read data from a deallocated object via a reference to a newly allocated object. Memory errors which give rise to these attacks are commonly referred to as *use-after-free* and *uninitialized read* errors, respectively. They can be successfully exploited to corrupt or leak both control and non-control data.

On the stack, temporal attacks exploit erroneous memory accesses into deallocated stack frames (via dangling pointers), or into uninitialized stack variables containing old data. In a traditional stack organization, stack frames are allocated and deallocated in a predetermined order, so an attack can determine which two objects overlap across stack frame allocations and corrupt/leak the intended data. In this scenario, the attack relies on the predictability of stack frame reuse induced by stack memory allocation.

3.2.3 Defenses

Figure 3.1 compares the stack organization adopted by different stack protection techniques, including *StackArmor*. Figure 3.1a shows the original stack layout.

Modern ASLR techniques introduce random gaps between stack frames [33; 66] and between buffer and non-buffer stack objects [66], separating and permuting them in two adjacent per-frame regions (Figure 3.1b). This strategy alone does not change the order of the stack frames, nor does it isolate vulnerable stack buffers, leaving the stack exposed to (spatial and temporal) guessing or spraying attacks. The problem is exacerbated by the gaps being limited in size for practical reasons and often statically determined for efficiency reasons [66]. These practical limitations result in even poorer randomization entropy, stack frame reuse unpredictability, and resilience to information leakage attacks.

Existing shadow stack techniques [33; 40; 85; 194] take a different approach, isolating the (potentially) vulnerable stack buffers on a separate, but contiguous, shadow stack (Figure 3.1c). This strategy alone does not prevent buffers from attacking *each other* in a predictable way in intra- and inter-frame spatial attacks, nor does it attempt to protect against temporal attacks.

In contrast, *StackArmor* completely disrupts the traditional stack organization, creating the illusion that stack frames and vulnerable buffers are neither temporally nor spatially adjacent in memory, but randomly drawn and isolated from one another (Figure 3.1d). This strategy prevents *all* the spatial and temporal attacks considered.

Additionally, in contrast to most of the above mechanisms, *StackArmor* operates entirely at the binary level, and can do so even in the absence of debug symbols. Thus, *StackArmor* does not rely on access to source code or recompilation.

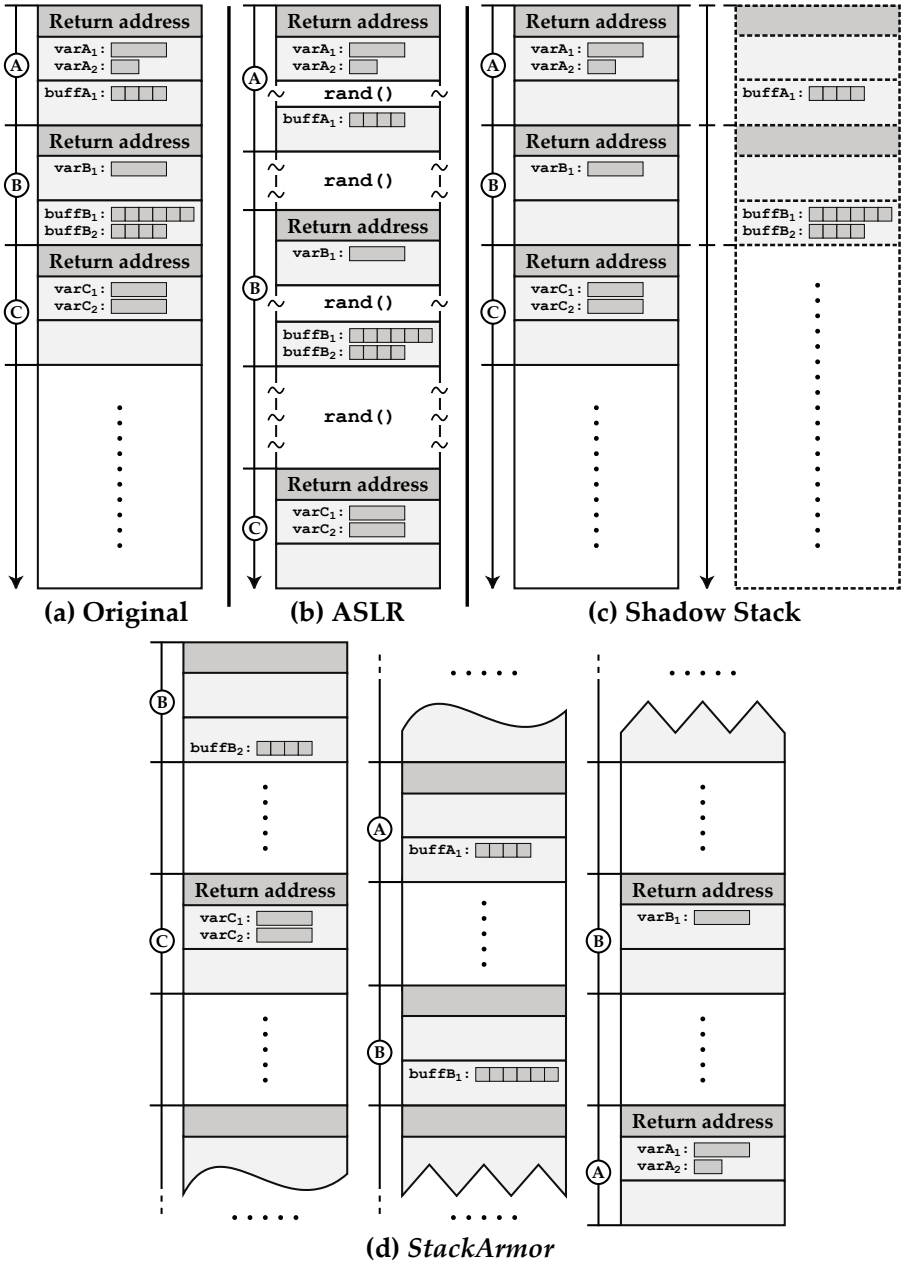


Figure 3.1: Comparison of different stack protection techniques.

3.3 StackArmor Overview

Figure 3.2 illustrates the overall *StackArmor* architecture. It consists of three *analysis modules*, a *binary rewriter*, and a *secure allocator*. The analysis modules pro-

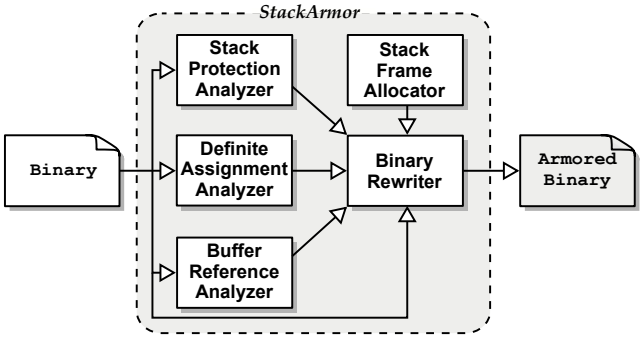


Figure 3.2: High-level overview of *StackArmor*.

vide support for the binary rewriting, while the allocator is employed to ensure an unpredictable allocation of stack frames. In this section, we describe the design of *StackArmor*. We defer the implementation details until Section 3.4.

The three analysis modules statically analyze each function found in the binary and determine what protection measures it requires. The *Stack Protection* (SP) analyzer conservatively decides which functions are not provably safe from spatial or use-after-free attacks, so need randomized (and isolated) stack frames. For this purpose, the analysis pinpoints functions that compute pointers to local variables.

Functions that are not assigned randomized stack frames still require protection from uninitialized reads. To this end, the *Definite Assignment* (DA) lists functions which are not provably safe from uninitialized variables. When such a function is called, *StackArmor* zero-initializes its relevant stack objects, effectively creating the illusion that the stack frame has been allocated from a random pool of zero-initialized frames and preventing potential errors from being exploited.

The *Buffer Reference* (BR) analyzer identifies stack buffers (and their references) that are provably safe to relocate. This strategy is used to isolate potentially vulnerable stack buffers from the original stack and prevent intra-frame spatial attacks. To this end, the BR analyzer relies on information about location and size of all the per-function stack objects, which is provided by debug symbols (if available) or dynamic reverse engineering techniques [116; 168].

Next, *StackArmor* combines the results of the analyses and the *Binary Rewriter* instruments all functions that cannot be conservatively proven safe. It creates a new stack frame for each function call and for each stack buffer, while the *Stack Frame Allocator* ensures at runtime that the frames are allocated in an unpredictable manner. After statically rewriting the binary, the resulting (*armored*) binary can run natively.

3.3.1 Stack Protection Analyzer

The *SP analyzer* employs static analysis to conservatively identify functions that cannot be proven safe from spatial and use-after-free attacks, so require stack protec-

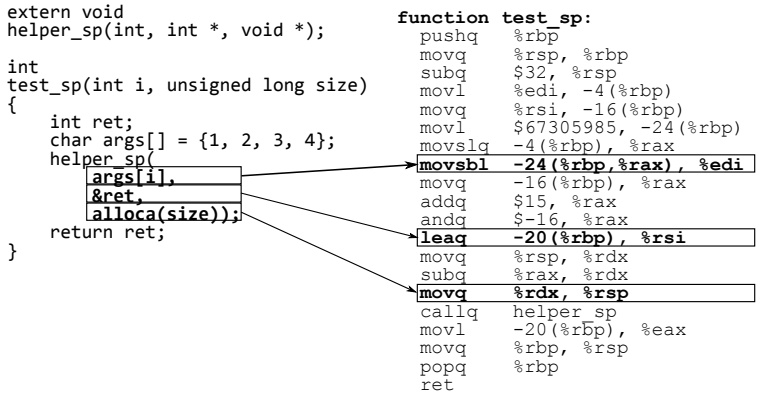


Figure 3.3: A sample *SP-unsafe* function that violates all three of the *SP-safety* rules imposed by the SP analyzer.

tion. It classifies as *SP-unsafe* all functions that compute pointers to local variables. These are functions which: (1) have stack-allocated buffers, (2) call `alloca`, or (3) contain stack variables that have their address taken. Our algorithm is inspired by the `-fstack-protector-strong` option in `gcc` [12], which uses similar analyses (at the source level) to identify functions prone to buffer overflows. One key difference is that our strategy is more generally tailored to locating *any* uses (and possibly leaks) of pointers into stack objects, allowing our analyzer to also identify functions prone to use-after-free attacks. Another difference is that operating at the binary level raises more challenges since stack accesses are mediated by the stack (or frame) pointer, generally subject to aliasing.

To address this challenge, the SP analyzer overapproximates the conditions above using a data-flow analysis over the Control-Flow Graph (CFG) of every function. In *SP-safe* functions, *StackArmor* allows references to stack objects only via the stack (or frame) pointer and a constant offset. More specifically, for every function, the SP analyzer performs a forward analysis of its CFG and marks the function as *SP-unsafe* if any of the following *SP-safety* rules hold:

(1) The stack is accessed through the stack (or frame) pointer and an offset stored in another register.

(2) The stack (or frame) pointer or derived pointers are stored into registers or memory outside the function's prologue and epilogue.

(3) The stack (or frame) pointer is manipulated outside the function's prologue and epilogue.

The purpose of rule (1) is to detect when a stack buffer is accessed in its local function, while rule (2) prohibits implicit accesses to stack variables, by confirming

that their pointers are never stored or passed to callees. Finally, rule (3) detects `alloca` invocations (and possibly other unsafe idioms).

The example function in Figure 3.3 violates all three *SP-safety* rules and is thus classified as *SP-unsafe*.[1] To read `args[i]`, the function accesses the stack through `%rbp` and `%rax`, violating rule (1). The second rule is violated when the function computes the address of `ret` and stores the resulting `%rbp`-derived pointer to `%rsi`. Finally, the invocation of `alloca` causes a manipulation of the `%rsp` register, which violates rule (3).

Because modern compilers typically generate very simple (and efficient) stack-accessing instructions for functions that maintain no pointers into the stack, our analysis can safely exclude these functions from instrumentation. As shown in Section 3.5.3, our analysis classifies 80% of functions across the SPEC CPU2006 benchmarks as *SP-safe* (geometric mean).

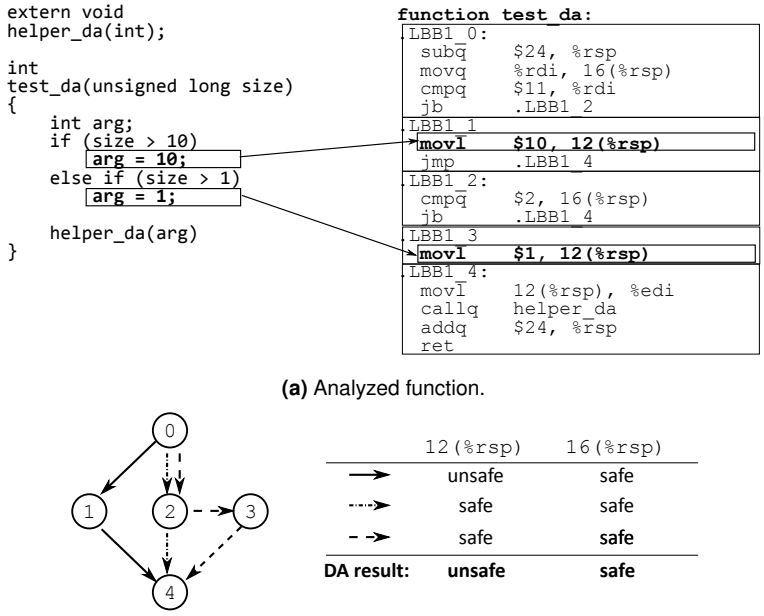
3.3.2 Definite Assignment Analyzer

To determine the functions (and objects) that require protection from uninitialized reads, the *DA analyzer* uses static analysis to conservatively identify all the *DA-unsafe* objects. These are defined as stack objects that cannot be proven to be initialized before they are first read. Our strategy is inspired by similar source-level analyses employed in safe languages to implement zero initialization semantics [91]. An important challenge when operating at the binary level is that object boundaries are no longer exposed in the code in any obvious way. Another challenge is that aliasing problems may generally prevent the analyzer from unambiguously mapping all the accesses to stack objects.

To address these challenges, the DA analyzer relies on two key observations. First, the functions that require uninitialized read protection are only those that have been marked as *SP-safe*, since the others are already protected using randomization and isolation. The *SP-safe* functions, by definition, have no buffers, pointers into the stack, or stack-accessing instructions that our data-flow analysis cannot map into a constant stack frame offset. In other words, we know that variables in *SP-safe* functions are not initialized in other functions, which drastically simplifies our definite assignment analysis by reducing it to a basic intra-procedural data-flow analysis [81]. Second, once our analysis has determined the constant stack frame offsets for all stack load and store instructions, the rest of the analysis can simply operate at the byte rather than the object level.

To determine functions and stack variables that require protection (and thus zero initialization), the DA analyzer proceeds as follows. For every function, it traverses its CFG in depth-first fashion and maintains a per-path tag map to keep track of the bytes in the stack frame that are read or written to in the current path. For every path, a first write-before-read event causes the DA analyzer to mark the target bytes as *path-safe* and a first read-before-write event causes the DA analyzer to mark the

[1] All assembly listings presented in this chapter were generated with `clang` 3.3.



(a) Analyzed function.

(b) Control-Flow Graph and analysis results.

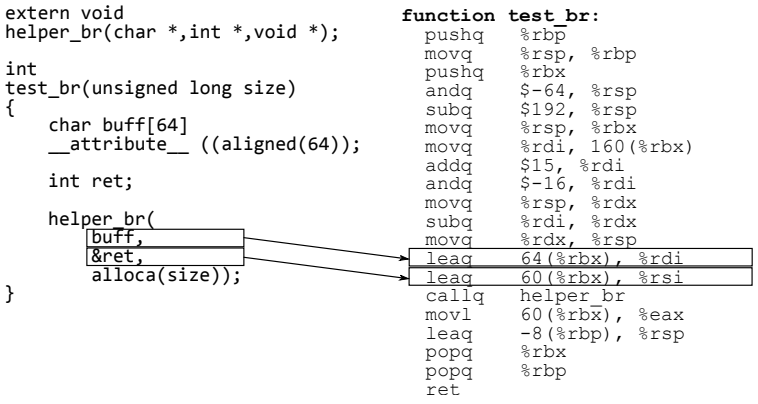
Figure 3.4: A sample *DA-unsafe* function. On the CFG path marked with solid arrows, the analyzer cannot prove that `12(%rsp)` (containing the `arg` variable) is initialized.

target bytes as *path-unsafe*. If the traversal reaches an unresolved control transfer or a function call, it marks all the bytes that are not marked at all yet as *path-unsafe*. At the end, all the bytes in the stack frame (and the function itself) that have been marked as *path-unsafe* at least once are marked as *DA-unsafe*, thus requiring uninitialized read protection. The example function in Figure 3.4 is *DA-unsafe*, since the analyzer cannot prove that on each CFG path, the stack location `12(%rsp)` (containing the `arg` variable) is written before it is read.

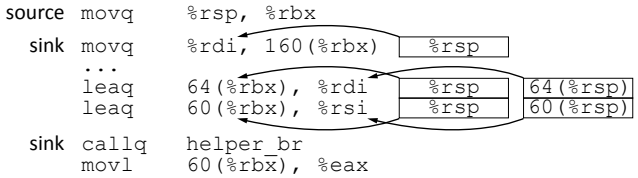
3.3.3 Buffer Reference Analyzer

For each function, the *BR analyzer* determines which stack buffers are safe to isolate in separate frames, meaning that all references to these buffers must be detected and relocated as well. The isolation serves as a protection against intra-frame spatial memory corruption attacks. To this end, the BR analyzer performs an intra-procedural static analysis to unambiguously map all the instructions taking stack addresses. *StackArmor* can safely isolate a buffer only if it proves that none of its references are ever used to access other memory regions.

Even given information on the location and size of all the stack objects (as provided by debug symbols or dynamic reverse engineering techniques [116; 168]), the mapping poses significant challenges. First, the stack (or frame) pointer is subject to aliasing. Another difficulty is that unlike source-level solutions [18], we cannot



(a) Analyzed function.



(b) Propagation by the BR analyzer of an explicit stack reference.

Figure 3.5: A sample function with an ambiguous stack reference. The `%rbx` base pointer, derived from `%rsp`, is used to access two separate objects on the stack: the `ret` variable located at `64(%rbx)` and the `buff` object at `60(%rbx)`.

assume that the relative layout of independent objects in memory is undefined. At the binary level, references to stack objects are inherently ambiguous—due to intra-procedural compiler optimizations, a reference to one object could later be used to access a completely different object within the function.

Figure 3.5 illustrates the problem. In this example, `clang` reserves a dedicated *base pointer* (`%rbx`) to access stack objects. The function prologue sets up the pointer to point to the bottom of a fixed-size portion of the stack, excluding stack space dedicated to *Variable-Length Arrays* (VLAs). In our example, this behavior is induced by the presence of stack alignment for the `buff` object and the VLA allocated on the stack using `alloca` [150]. In `gcc`, this situation is handled in a similar way, mediating the necessary accesses to stack objects with a dedicated *Dynamic Realigned Argument Pointer* (DRAP) register (typically `%r10`) [105]. Using an additional register to access the stack causes a stack reference (`movq %rsp, %rbx`) to be used to access distinct stack objects (both `buff` and `ret`) before calling `helper_br`. As detailed in the figure, the BR analyzer detects the ambiguity and refuses to remap the stack references for the given function.

The algorithm implemented in the BR analyzer operates in two steps. First, for each function, it identifies and propagates all the explicit stack references down the

CFG. The mechanism is inspired by the way prior techniques use constant propagation to discover targets of indirect calls [77; 170]. Next, the analysis verifies that each reference targets a single stack object.

In the first step, the BR analyzer runs an intra-procedural flow-sensitive static data-flow tracking analysis [63] to determine where stack addresses are dereferenced, stored to memory, or escape the current function. It takes two types of taint seeds present in the instructions of the CFG: constants and explicit stack references (i.e., operands of the form `rsp+offset` or `rbp-offset`, where `offset` is an immediate value). The analyzer considers each reference in turn and treats this reference and all the constants as tainted. With this setup, for each control-flow path, it propagates the tainted values down the CFG, in a depth-first fashion and builds up expressions representing the computed values. When the propagation reaches a *sink* instruction, it labels the propagated reference as follows.

(1) If the sink instruction accesses memory at an address tainted by the stack reference currently under consideration, we distinguish two cases. (a) The address evaluates to a single stack location. In this case, the analysis locates the target stack object, labels the reference accordingly, and then continues the propagation. (b) The address contains an unresolved index. This case causes the analysis to label the reference as `unknown` and stop.

(2) If the sink stores a value tainted by the reference to memory, the analysis conservatively assumes that the value is a valid pointer and proceeds as in (1).

(3) If the sink is a `call` instruction and a register holds a value tainted by the reference, the analysis conservatively assumes that this register may contain a valid pointer and proceeds as in (1).

Once the propagation completes, the analysis verifies that across all paths, each reference is labeled with exactly one known object (i.e., a buffer). If this check fails, the analysis conservatively reports no buffers for the current function. However, if each reference *can* be successfully mapped into a single stack buffer, the analysis reports on all the buffers and all the stack-referencing instructions that reference those buffers. Note that, since our analysis is fully conservative, it may occasionally fail to isolate some buffers (see Section 3.5). This conservative behavior is needed to ensure that we produce no false positives, which would otherwise result in instrumentation-induced undefined behavior.

3.3.4 Stack Frame Allocator

Figure 3.6 depicts the allocation strategy adopted in our *stack frame allocator*. For each thread call stack, the allocator maintains a pool of F contiguous *physical frames* (PFs), all preallocated in a random region of the virtual address space. Each physical frame consists of D data pages surrounded by 1 guard (non-mapped) page to isolate frames from one another. To map logical frames into one or more physical

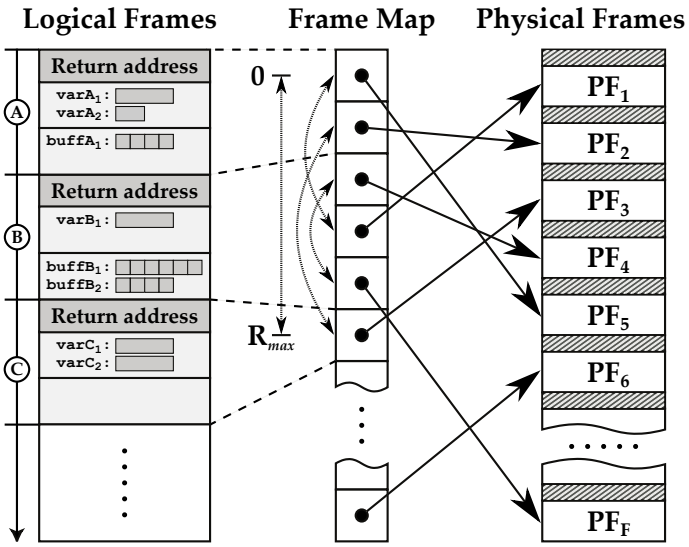


Figure 3.6: *StackArmor*'s stack frame allocation strategy.

stack frames, our allocator uses a *frame map* of F preallocated entries, each initialized with a pointer to a physical frame. When allocating a frame at runtime, the allocator fetches the next entry in the frame map and decrements the frame map index, incrementing it again upon deallocation.

To ensure that the relative distance between physical frames is unpredictable, the entries in the frame map are initialized with a random permutation of the physical stack frames. This static randomization strategy efficiently protects against spatial attacks, but in itself is insufficient for temporal attacks, since the entries in the frame map can still be predictably reused across consecutive calls (e.g., in a loop). To protect against temporal attacks, our allocator performs in-place frame map randomization, swapping the next entry with the entry located at a random offset $R \in [1; R_{max}]$ before allocating a new frame. R is computed using a global counter and a random number provided by the `rdrand` x86 instruction. This in-place randomization strategy eliminates the need for free lists and efficiently satisfies all our requirements.

Because instrumented stack frames are allocated at page granularity with low reuse, we take additional measures to minimize physical memory consumption. Moreover, we must comply to the restrictions on the maximum number of (guarded) virtual memory areas (VMAs) imposed by the operating system (65,535 on stock Linux). For these reasons, our allocator retains fine-grained control over the memory pages preallocated in the stack frame pool using predetermined *soft limits*. In particular, while the parameter F establishes the maximum number of active stack frames, a soft limit F_{SL} determines how many frames are immediately made available to each application thread. The other $F-F_{SL}$ frames are initially all mapped as consecutive inaccessible pages, allowing the operating system to consider them as a

single VMA. When a thread exhausts its physical frames, our allocator doubles the value of the soft limit F_{SL} and remaps the new physical frames.

We use a similar adaptive strategy to manage the individual physical stack frames. The soft limit D_{SL} determines how many data pages are immediately made available to each stack frame. The other D–D_{SL} pages are initially all mapped as inaccessible. In the rare cases when a stack frame requires more space, a user-level page fault handler (a signal handler intercepting `SIGSEGV` signals) increases D_{SL} on demand using the same exponential growth strategy described earlier. The soft limit D_{SL} is restored to its initial value upon stack frame deallocation. The extra memory pages are then also returned to the operating system (using POSIX `MADV_DONTNEED`, similar to prior work [17]), and remapped as inaccessible.

The allocator configuration parameters are fully user-configurable, with carefully chosen defaults that proved effective in our tests. The parameter R_{max} controls the entropy of our in-place randomization strategy. By default, *StackArmor* opts for maximum entropy, using $R_{max} = F_{SL}$ in non-threaded programs. In multi-threaded programs, we gradually decrease the per-thread value of R_{max} (and F_{SL}) with the number of active threads, by 5% for each newly created thread, with a minimum of $R_{max} = F_{SL} = 128$. This strategy strictly bounds physical memory consumption in heavily threaded programs. To further reduce memory overhead (due to more resident physical frames) and performance overhead (due to poorer data cache locality), users may also opt to configure lower R_{max} values.

The maximum number of active stack frames F and the soft limit F_{SL} default to $16,384$ and $1,024$, respectively. These values do not excessively reduce the number of VMAs available to the program, resulting in a reduction of only 39% on stock Linux for a program with 100 active threads within the soft limit. Moreover, the values can elastically adapt to programs with deep instrumented call stacks. The number of per-frame data pages D and the soft limit D_{SL} default to the OS-specified maximum stack size (2,048 pages on stock Linux) and $0.1D$, respectively. This yields a conservative stack allocation strategy while providing strong physical memory consumption guarantees.

3.3.5 Binary Rewriter

The *binary rewriter* uses the information provided by the analysis modules to guide the instrumentation process. First, it instruments all the call sites invoking *SP-unsafe* functions, so that each such function runs in a new randomized and isolated stack frame, protected against inter-frame spatial and temporal attacks.

Figure 3.7 shows the functionality we add to instrumented call sites. Before each instrumented `call` instruction, our allocator reserves an armored stack frame. We then copy all the already pushed call arguments onto the new armored stack, and redirect the stack pointer (`%rsp`) to the new frame. We save the return address (pushed by `call`), the old `%rsp`, and the new `%rsp` in a dedicated *context* on a separate stack. The callee now runs protected with an armored stack.

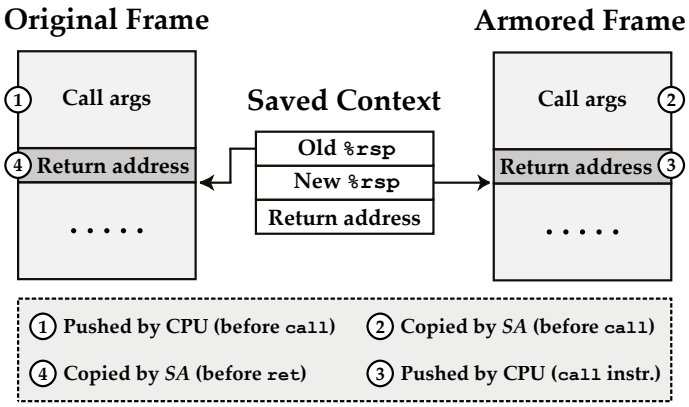


Figure 3.7: Call site instrumentation in *StackArmor*.

We also instrument return (`ret`) sites, so that upon return of an armored function, the armored stack frame is deallocated and the original stack resumed. Resuming the original stack is done by restoring the saved `%rsp`, and pushing the saved (and trusted) return address onto the stack just before the `ret` instruction.

Our call site instrumentation strategy is necessary to copy caller-specified arguments, whose number may change across call sites due to variadic calls. However, it may also complicate stack management when caller and callee cannot be statically paired with one another. For instance, consider the case of indirect calls. Since we can typically not determine statically whether their target is *SP-unsafe* or not, we must conservatively instrument such call sites. However, if the callee turns out to be *SP-safe*, the corresponding `ret` would not be instrumented, and execution would return with `%rsp` still pointing into the armored stack frame.

To address this problem, the rewriter instruments the instructions following call sites which contain indirect calls, library calls, and other special idioms such as `setjmp`. This instrumentation code then restores the original stack and allows the caller to resume execution consistently. Our `setjmp` instrumentation also checks if control returned from a `longjmp` invocation. If so, it garbage collects all the deeper (no longer needed) physical stack frames.

The complementary situation is also possible: an uninstrumented call site with an instrumented *SP-unsafe* callee. For example, this case may occur when dealing with uninstrumented libraries which call user-specified callbacks. To detect (and ignore) this situation, the return site instrumentation checks if the current `%rsp` is lower than the new `%rsp` in the most recent saved context. Our instrumentation strategy can efficiently handle all possible combinations of (instrumented or uninstrumented) caller-callee combinations in a conservative way. Thus, *StackArmor* supports unrestricted use of shared libraries and arbitrary optimizations driven by our static analyzers.

To protect against intra-frame spatial attacks, the rewriter instruments all the *SP-unsafe* functions with buffers reported by the BR analyzer. In particular, it first instru-

ments the entry site (i.e., after the function prologue) to relocate each reported buffer into a new stack frame provided by the allocator. Second, it remaps all the stack-referencing instructions reported by the BR analyzer to reference the corresponding buffers in their own independent frames. These per-buffer frames are garbage collected when the main armored frame is deallocated.

To protect the remaining *SP-safe* functions against uninitialized reads, the rewriter instruments the entry sites of all stack regions reported by the DA analyzer, adding zero-initialization code. We implement efficient zero-initialization semantics by coalescing multiple `bzero` writes into the same memory word (8 bytes).

3.4 Implementation

We implemented *StackArmor* for the Linux x86-64 platform, using PEBIL [114] to statically instrument 64-bit ELF binaries, turning them into stand-alone armored binaries. *StackArmor* is easily portable to other UNIX systems. In this section, we discuss the disassembly requirements of our prototype, and also present implementation details of our instrumentation approach. We also discuss limitations of our current *StackArmor* implementation.

3.4.1 Binary Disassembly and Analysis

StackArmor requires information about instructions, basic blocks, CFGs, and functions. As discussed in Chapter 2, these primitives may suffer from inaccuracies or incompleteness. It is often possible to make a tradeoff between these two properties, for instance by sacrificing completeness for higher correctness (a *conservative* analysis). We show that *StackArmor* relies only on the correctness of disassembly and not its completeness. It is designed to cope with incomplete information, gracefully reducing security guarantees without breaking the binary. This section discusses the disassembly requirements and tradeoffs applied in *StackArmor*.

3.4.1.1 Incomplete disassembly

As discussed in Chapter 6, we cannot rely on fully correct and complete static disassembly of stripped x86/x64 binaries [22; 170; 188]. Challenging cases like indirect control flows mean that many recovered primitives (especially CFGs and functions, but potentially even instructions) may be incomplete. Thus, we must ensure that *StackArmor* can be used safely (without affecting the semantics of the instrumented binary) even if 100% accurate disassembly is not available. We discuss how *StackArmor* deals with the following challenges.

(1) *Unresolved jumps*: To deal with unresolved `jmp` instructions (indirect jumps whose targets remain unknown), our static analyses behave in a conservative way. The SP analyzer classifies functions with an unresolved jump as *SP-unsafe*. Similarly, the BR analyzer labels all buffers in such functions as `unknown`. The rewriter

then isolates such functions by setting up a new armored stack frame. Should the rewriter miss and fail to instrument a return instruction, *StackArmor* handles this case without trouble, as discussed in Section 3.3.5. Since the DA analyzer never analyzes functions already classified as *SP-unsafe*, it need not consider issues with unresolved jumps.

(2) *Unresolved calls*: Similarly to unresolved jumps, the analyzers also deal with unresolved indirect `call` instructions in a conservative way. However, an unresolved `call` does not influence the instrumentation of the calling function.

(3) *Missing functions*: Since *StackArmor* is not aware of functions which are missed by the disassembly process, it simply does not analyze or instrument them. As discussed in Section 3.3.5, the binary rewriter ensures that the binary works well even if an instrumented function calls an uninstrumented one, or vice versa. Therefore, missing functions only result in a graceful reduction of security due to their not being instrumented.

3.4.1.2 Stack pointers and function prologue

Our static analyses (Sections 3.3.1–3.3.3) consider explicit stack references: operands of the form `rsp+offset` or `rbp-offset`. While this assumes the special role of `rsp` and `rbp`, nothing bad will happen if (due to optimizations) `rbp` is not used as the base pointer—the analyses are limited to references derived from the `rsp` register. To detect function prologues, *StackArmor* follows the ABI for x86-64/Linux [9]. In this, `rsp` is a "sacred" register, inherently used by the important `push`, `pop`, `call`, and `ret` instructions. Thus, in practice (for non-obfuscated binaries), we can safely assume that `rsp` is used as the stack pointer.

3.4.1.3 Function arguments

As mentioned in Section 3.3.5, before a `call`, the binary rewriter copies all call arguments already pushed onto the stack. To do this, it examines the basic block containing the `call` instruction and checks how many bytes to transfer. Our approach places only the following restrictions on the calling convention: (1) stack-based argument passing is done in a single basic block, and (2) callers make no assumptions on how arguments are handled by callees. Though *StackArmor* is compatible with more general solutions (as proposed in prior work [20; 21]), we are not aware of any calling convention violating (1) or (2), and encountered no problems in our tests.[2]

3.4.2 Instrumentation

Our binary rewriter is based on PEBIL [114], an efficient static binary instrumentation platform for Linux. PEBIL can install hooks at arbitrary locations in a binary to

[2] Although tail calls violate (2), they do not require copying back callee-owned arguments, given that the callee returns directly to the caller of the caller.

call a predetermined handler enclosed in a shared library, with the instrumentation automatically saving and restoring registers to create a consistent execution context.

For our purposes, we extended PEBIL in three ways. (1) We only save and restore registers actually used in *StackArmor*'s handler, so that we minimize context switching costs for our instrumentation code (this is safe because *StackArmor* does not use external library calls on the instrumentation path). (2) We implemented support for handlers enclosed in a static library (injected into the binary by our custom rewriting tool `elfinject`), thereby eliminating the costs associated with indirect PLT calls on the instrumentation path. This allows our stack frame allocator to be implemented as a static library with efficient position-dependent code. (3) We allow PEBIL to access Thread-Local Storage (TLS), where our instrumentation stores references to per-thread metadata and stack frames managed by our allocator.

3.4.3 Limitations

By default, PEBIL relies on symbols for function detection. However, this is not a limitation of *StackArmor* itself, which is designed to handle incomplete disassembly without breaking the binary, and while gracefully reducing its security guarantees (as discussed in Section 3.4.1). As we discuss in Chapter 7, our novel function detector *Nucleus* is capable of finding over 95% of the functions even in highly optimized stripped binaries. When used in *StackArmor*, this would ensure that the vast majority of stack frames is analyzed and (if needed) instrumented, even in stripped binaries.

Our current *StackArmor* implementation inherits PEBIL's lack of support for C++-style exceptions. Though this limitation can be addressed with additional implementation effort, this proved unnecessary for our tests with several real-world server applications and benchmarks (Section 3.5).

3.5 Evaluation

We evaluated *StackArmor* on a workstation equipped with an Intel i7-4770K CPU clocked at 3.90 GHz, a 256KB per-core cache, an 8MB shared cache, and 8GB of DDR3-1600 RAM. We ran all our tests on an Ubuntu 12.10 installation running Linux kernel 3.12 (x86-64).

For our evaluation, we selected several popular servers: `lighttpd` v1.4.28, `vsftpd` v1.1.0, `opensshd` v3.5, and `exim` v4.69. To benchmark `lighttpd`, we used the Apache benchmark [1] configured to issue 25,000 requests with 10 concurrent connections and 10 requests/connection. To benchmark `vsftpd`, we used `pyftpbench` [5] configured to open 100 connections and request 100 files of size 1 KB per connection. To benchmark `opensshd` and `exim`, we used the OpenSSH test suite and a homegrown script which repeatedly launches `sendemail` [7], respectively. Moreover, to test *StackArmor* in memory-intensive scenarios and better investigate the performance-security tradeoffs, we also evaluated with all the C

benchmarks in SPEC CPU2006. We ran all our experiments 11 times, and report the median. We ensured that the CPUs were fully loaded throughout our tests.

In our evaluation, the BR analyzer was given information on the location and size of stack objects, generated from debug symbols. This means we evaluate *StackArmor*'s best-case security guarantees. Under these circumstances, our BR analyzer was able to identify (and isolate) 90.7% of the buffers on average across all our programs. As discussed in Sections 3.3.3 and 3.4.1, security guarantees reduce gracefully when debug information is not available, and (for instance) data structure reverse engineering techniques are used instead [168].

Our evaluation answers four key questions. (1) *Security*: Is *StackArmor* effective in protecting against both spatial and temporal stack-based attacks? (2) *Performance*: Does *StackArmor* yield acceptable runtime overhead? (3) *Memory usage*: How much memory does *StackArmor* require? (4) *Multithreading*: Does *StackArmor* perform and scale well in multithreaded programs?

3.5.1 Security Against Spatial Attacks

To evaluate the security guarantees offered by *StackArmor* against spatial attacks, we measured the *attack surface* reduction. This quantifies both the number of vulnerable targets (stack-allocated objects) and the number of offenders (such as stack-allocated buffers) in intra-frame and inter-frame attack scenarios.

3.5.1.1 Intra-frame attack surface reduction

The intra-frame attack surface $S_{\text{intra}}(f)$ of a given function f quantifies the extent to which the B_f stack-allocated buffers in f's stack frame threaten the N_f stack objects in the same frame through potential buffer overflow/underflow attacks. Here, N_f encompasses both non-control and control data, including the return address.

$$S_{\text{intra}}(f) = \sum_{i=1}^{B_f} \sum_{j=1}^{N_f} \text{canAttack}(i, j) \; ? \; 1 : 0$$

Ideally, *StackArmor* reduces the attack surface induced by a traditional stack organization ($B_f \times N_f$) to 0 (no buffer can predictably attack other intra-frame objects). In general, however, the reduction is subject to the precision of our BR analysis. Table 3.1 shows the mean intra-frame attack surface reduction across all functions with stack-allocated buffers. We also compare against the protection offered by traditional shadow stack techniques in the ideal case (source-level, with all buffers remapped).

As shown in the table, *StackArmor* consistently yields a high attack surface reduction. We achieve 100% stack buffer isolation for `lighttpd` and `vsftpd`, and a worst-case reduction of 94.4% for `opensshd`. We offer stronger security than traditional (source-level) shadow stacks, which (unlike *StackArmor*) also fail to prevent in-frame attacks between buffers.

	Intra-frame		Inter-frame	
	Shadowing (Source)	*StackArmor*	Shadowing (Source)	*StackArmor*
`lighttpd`	100.0%	100.0%	99.0%	99.9%
`exim`	96.1%	96.8%	97.2%	99.9%
`opensshd`	93.2%	94.4%	94.0%	99.9%
`vsftpd`	100.0%	100.0%	99.6%	99.9%
SPEC$_{gm}$	91.5%	95.95%	94.6%	99.9%

Table 3.1: Mean attack surface reduction for all functions with stack-allocated buffers.

3.5.1.2 Inter-frame attack surface reduction

The inter-frame attack surface $S_{\text{inter}}(f)$ of a function f is subject to the probability p_k of the stack frame of the caller k (a function in the set of C_f callers of f) being active on the call stack before that of f. Note that while an attacker could potentially overflow into any active stack frame (even of functions that do not call f directly), the spatial predictability guarantees reduce as we move higher up the call stack.

$$S_{\text{inter}}(f) = \sum_{i=k}^{C_f} \left[p_k \cdot \sum_{i=1}^{B_f} \sum_{j=1}^{N_k} \text{canAttack}(i, j) \ ? \ 1 : 0 \right]$$

To concretely compute $S_{\text{inter}}(f)$ for our test cases, we assume p_k to be a uniform distribution. That is, $p_k = 1/C_f$ for a traditional stack organization and $p_k = 1/R_{max}$ for *StackArmor*, with the swap size R_{max} set to $1,024$ in our experiments (see Section 3.3.4). To find the set of C_f callers for every given f, we applied static callgraph analysis of our test programs using LLVM [112]. Our implementation relies on data structure analysis [113] (an efficient context-sensitive and field-sensitive points-to analysis) to conservatively analyze function pointers used in indirect calls.

Table 3.1 shows the mean inter-frame attack surface reduction across all functions with stack-allocated buffers, comparing *StackArmor* against traditional (source-level) shadow stack techniques. As the table shows, *StackArmor* yields a very high inter-frame attack surface reduction of 99.9% across all our tests, consistently higher than traditional source-level shadow stack techniques. The higher reduction compared to our intra-frame analysis highlights the effectiveness of *StackArmor* in increasing the randomization entropy in probabilistic attack models. Even when compared to prior source-level stack randomization strategies that introduce random gaps between objects [33; 94], *StackArmor* yields much stronger randomization guarantees, ensuring that logically contiguous objects (and frames) are never physically adjacent in memory.

	Basic	+Intra-frame	+UZero
`lighttpd`	1.06x	1.07x	1.10x
`exim`	1.01x	1.04x	1.05x
`opensshd`	1.00x	1.01x	1.01x
`vsftpd`	1.00x	1.01x	1.04x
SPEC_{gm}	1.16x	1.22x	1.28x

Table 3.2: Benchmark runtimes normalized against the baseline.

3.5.2 Security Against Temporal Attacks

To evaluate the security guarantees offered by *StackArmor* against temporal attacks, we analyzed the unpredictability of stack frame reuse. To this end, we measured the randomness of physical stack frame addresses generated by *StackArmor*. For each of the benchmarked programs, we evaluated the randomness of stack frame addresses under four configurations: (1) Baseline, (2) *StackArmor* with $R_{max} = 0$, (3) ASLR, and (4) *StackArmor* with $R_{max} = F_{SL} = 1,024$. Our ASLR implementation dynamically generates random inter-frame gaps $g \in [0; 40\text{KB}]$ using the `rdrand` instruction. This actually yields higher entropy than modern ASLR techniques used in practice [33; 66], which allow deterministic [33] or periodic [66] stack frame reuse in loops, or use statically generated random inter-frame gaps [66].

To measure randomness, we conducted a non-parametric hypothesis test for randomness (Bartels' rank test [29]), with the null hypothesis that the sequence is generated randomly. For the first two configurations, p-values are consistently below $1.9\text{e}{-7}$, and for ASLR they are below $9.3\text{e}{-3}$. Thus, for these configurations, we can reject the null hypothesis at significance level $\alpha = 0.01$. In contrast, *StackArmor* shows high p-values $\in [0.37; 0.90]$, indicating that *StackArmor* yields truly unpredictable stack frame reuse and strong protection against temporal attacks.

3.5.3 Performance

Table 3.2 and Figure 3.8 show the overhead induced by *StackArmor*, split into individual contributing factors. Table 3.2 shows runtime overheads, while Figure 3.8 depicts overheads in terms of number of cycles and executed instructions. The *Basic* overhead shows the overhead due to mapping logical stack frames into randomized stack frames (as maintained by our allocator) only. *+Intra-frame* and *+UZero* show the resulting overhead when adding intra-frame protection and zero initialization semantics, respectively. Finally, *Rewriter only* and *Rewriter+Allocator* (shown only in Figure 3.8) isolate the costs of static rewriting (saving/restoring registers, and jumping into *StackArmor*'s handler and back), and overhead due to stack frame allocation/deallocation.

The *Basic* runtime overhead amounts to 16% on SPEC (geometric mean), and only 6% in the worst case for the server binaries. Isolating individual stack buffers adds an additional 6% overhead in the worst case. Finally, enabling zero initializa-

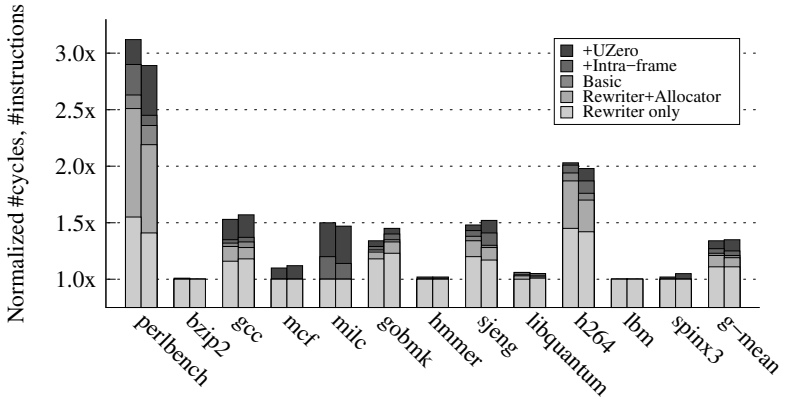


Figure 3.8: SPEC CPU2006 performance overhead (#cycles and #instructions).

tion yields an overall overhead of 28% for SPEC (geometric mean), and 10% (worst case) for the servers.

Figure 3.8 provides an explanation for the higher runtime overhead we observe for SPEC CPU2006 compared to the server programs. This is due to heavily recursive benchmarks such as `perlbench`; eliminating these reduces the geometric mean overhead by 10%. Moreover, Figure 3.8 shows that cycle and instruction overheads are very similar to the runtime overheads, demonstrating that the reduced cache locality induced by *StackArmor* is a relatively insignificant overhead factor. We can also see that in most cases, binary rewriting costs dominate the overhead, while allocation costs are a close second.

Table 3.3 shows the function characteristics of the SPEC CPU2006 benchmarks, and details the instrumentation decisions made by our SP analyzer and DA analyzer. The stack statistics (max depth and frame size) were measured during execution. For comparison, we also show statistics for the `-fstack-protector-strong` source-level analysis (SP-unsafe, source).

Our results show that our analyzers are effective at limiting instrumentation overhead. In particular, our SP analyzer reported only 20% of the functions as unsafe on average (geometric mean), coming close to the 16% rate achieved by `-fstack-protector-strong` (which, unlike *StackArmor*, needs source). Over all the *SP-unsafe* functions, our BR analyzer correctly identified 92.8% of the buffers (geometric mean). Our DA analyzer classified 52% of the functions as unsafe, while marking only 42% of stack objects in those functions for zero initialization.

Due to the limited number of unsafe (and thus instrumented) stack frames, none of the tests required our allocator to dynamically increase the soft limit F_{SL}, even for the worst-case maximum stack depth of 394 frames (`perlbench`). As can be seen in the relatively large maximum stack frame sizes observed (10.4KB, geometric mean), many of the functions ran uninstrumented, efficiently reusing the physical stack frame inherited from their caller. Even so, stack frames stayed small enough

	Functions (#)				Stack	
	Total	SP-unsafe (Source)	SP-unsafe (*StackArmor*)	DA-unsafe	Max depth	Max frame (KB)
perlbench	1,885	397	409	1,227	394	8.7
bzip2	112	18	22	48	4	10.2
gcc	5,630	846	972	3,499	48	54.2
mcf	33	1	4	17	80	82.8
milc	244	73	77	91	9	80.5
gobmk	2,690	335	337	1,351	5	5.5
hmmer	548	118	121	324	1	2.8
sjeng	153	41	42	73	2	1.1
libquantum	127	27	31	56	1	2.0
h264ref	599	101	105	391	4	2.7
lbm	29	5	8	14	30	26.3
sphinx3	380	57	59	218	4	21.4
SPEC$_{gm}$	332	54	66	172	9	10.4

Table 3.3: Instrumentation decisions and call stack statistics for SPEC CPU2006.

to avoid the need for an increase in the soft limit D_{SL} by our user-level page fault handler (as shown in Section 3.5.4, memory overhead is also limited).

Overall, *StackArmor* provides a stronger and broader protection model than traditional shadow stack approaches, with comparable runtime overhead [33; 40; 58; 61; 85; 145; 166; 182; 194]. While source-level stack randomization approaches achieve lower overhead, all existing work does so at the cost of poorer entropy and isolation guarantees [33; 66]. These results show that *StackArmor* provides strong security, while allowing an efficient and flexible security-performance tradeoff.

3.5.4 Memory Usage

StackArmor's stack frame allocation strategy translates to higher virtual and physical memory usage. Since virtual memory is plentiful in modern (x86-64) systems, we focus our analysis on physical memory usage. Figure 3.9 depicts the resident set size (RSS) increase for varying values of the maximum swap size R_{max}. To isolate the effects of varying R_{max}, we configured *StackArmor* with the default number of maximum stack frames and no soft limit ($F = F_{SL} = 16,384$).

With $R_{max} = 0$, the RSS increase is only caused by internal fragmentation, since stack frames are allocated at page granularity (0.2–0.5MB increase across our server programs). Setting $R_{max} > 0$ progressively reduces stack frame reuse, increasing RSS linearly because a new (previously non-resident) stack frame is drawn for (nearly) each call. For very large R_{max} (e.g., $R_{max} = 10,000$), we observe a very high RSS increase (0.7–118.5MB across our server tests). The cause for this behavior becomes evident in some long-running SPEC benchmarks which continuously call functions with many in-frame buffers. This quickly makes all *StackArmor*'s physical stack frames resident in memory, resulting in a worst-case RSS increase of 195.1MB for SPEC. Clearly, there is a tradeoff to be made between randomization entropy and RSS, and R_{max} should typically be bounded. In our default configura-

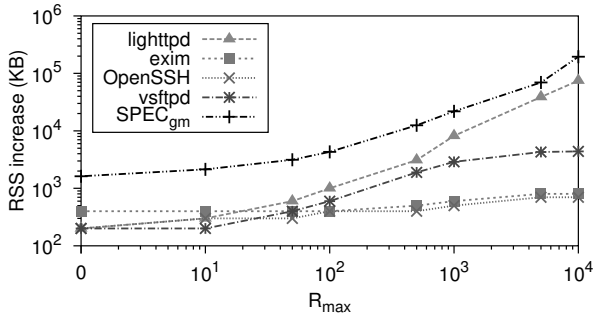


Figure 3.9: RSS increase due to *StackArmor*.

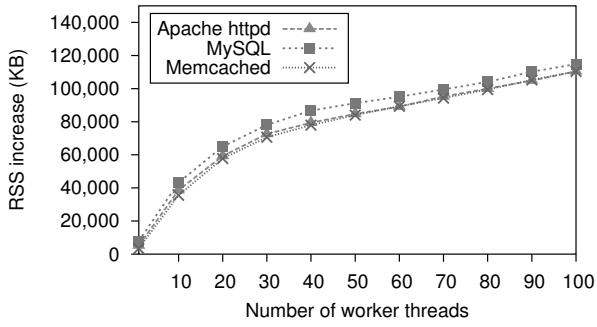


Figure 3.10: RSS increase due to *StackArmor* in multithreaded programs.

tion with $R_{max} = 1,024$ (which our previous experiments show to provide sufficient entropy), the RSS increase for SPEC is limited to only 22MB (geometric mean).

3.5.5 Multithreading Support

We evaluated scalability in threaded programs on MySQL (v5.1.65), `memcached` (v1.4.20), and Apache `httpd` (v2.2.23, `mpm_worker_module`). For each server, we varied the number of worker threads $T = [1; 100]$, matching the number of concurrent connections/operations to T in our benchmarks. We evaluated Apache using the Apache benchmark [1], configured to issue 25,000 requests and 10 requests/connection. MySQL was evaluated using Sysbench OLTP [8], with 10,000 transactions using a read-write workload. For `memcached` we used the memslap benchmark [3] with 1,000,000 operations. We configured *StackArmor* with the defaults described in Section 3.3.4 and full protection.

Runtime overhead for increasing T is nearly constant (29–33% for Apache, 35–37% for MySQL, and 13–15% for `memcached`). This owes to the servers' thread pooling strategies, minimizing the need for *StackArmor* to allocate per-thread metadata on the fast path. The relatively high overhead compared to non-threaded programs is due to extra TLS-accessing costs incurred by *StackArmor* to implement

thread-safe allocation and instrumentation. Since these costs are bounded even for $T = 100$, they do not impair *StackArmor*'s ability to scale efficiently.

The RSS increase grows linearly with the number of threads, due to the per-thread stack frame pools maintained by our allocator. As can be seen in Figure 3.10, *StackArmor*'s ability to adapt R_{max} to the number of active threads results in a worst-case RSS increase of only 115MB (MySQL, with $T = 100$). Thus, *StackArmor* provides reasonable scalability even for heavily threaded servers. At the same time, it preserves strong randomization entropy guarantees, with $R_{max} = \{299, 474, 878\}$ for the default thread configurations of Apache `httpd`, MySQL, and `memcached`, respectively.

3.6 Related Work

Protection from stack-based vulnerabilities Early stack protection work often relies on canaries [12; 64; 87]. These only affect spatial (smashing) attacks and are susceptible to information leakage. Shadow stacks offer stronger isolation, but are ineffective against temporal attacks and typically limited to buffer-to-non-buffer attacks [33; 40; 58; 61; 85; 145; 166; 182; 194]. Traditional Address Space Layout Randomization (ASLR) randomizes the base of the stack [10; 32; 137]. However, this still allows attacks that rely only on the relative distance or reuse between stack objects. Fine-grained ASLR also introduces random gaps between stack frames and buffers/non-buffers [33; 66], but remains vulnerable to spraying attacks and information leakage. Moreover, it offers no isolation of stack objects.

Protection from temporal attacks Prior work explores protection against heap-based use-after-free vulnerabilities, through garbage collection [36; 148], secure allocation [17; 30; 80; 122; 134], and dynamic memory checking [43; 47; 129; 162]. *StackArmor* applies some ideas from this work to the stack, such as fully randomized allocation [30; 122; 134] with a sparse page layout [134] and a single object per page(s) [122]. Some dynamic tools can detect uninitialized reads on the stack [43; 47; 129; 162], but they incur very high overhead. More lightweight approaches sacrifice precision for performance [30; 94]. *StackArmor* implements both strong and lightweight protection against uninitialized reads. *StackArmor*'s zero initialization is comparable to secure deallocation (which is, however, quite expensive on the stack) [59]. Definite assignment analysis, similar to our binary-level approach, has been implemented before using source [68; 79; 80; 91; 104].

Protection from generic memory errors A myriad of generic memory error protection approaches have been explored in the last decade. Many popular techniques, including data flow integrity [50], write integrity testing [18], bounds checkers [19; 78; 193], and memory safe environments [68; 79; 80; 104; 128], require source code or recompilation. Binary-level approaches, including Control-Flow Integrity

(CFI) [15; 197], Instruction Set Randomization (ISR) [143], ROP protection systems [56; 136], and Dynamic Taint Analysis (DTA) [62; 130] can stop various control-flow diversions, but (unlike *StackArmor*) do not protect non-control data. Moreover, techniques like DTA [62; 130], multi-variant execution [65; 155; 156], and tools like Valgrind [129] and Dr. Memory [43] often incur performance overheads of an order of magnitude or more.

3.7 Conclusion

Nearly two decades after the first stack smashing attack, performance concerns still lead modern compilers to ship with weak stack protection mechanisms, ultimately resulting in binaries left at the mercy of attackers. We have introduced *StackArmor*, a novel comprehensive stack protection system which offers a practical solution to this problem. Unlike prior systems, *StackArmor* can efficiently protect against arbitrary spatial and temporal stack-based attacks, operates entirely at the binary level, and supports policy-driven defenses to allow end users to tune the performance-security tradeoff. To achieve this, *StackArmor* abandons the traditional stack organization and relies on a combination of randomization, isolation, and zero initialization to create the illusion that stack objects are drawn from a fully randomized space. We use a novel static binary analysis approach to limit protection to stack frames that truly need it, greatly reducing performance overhead. Our experimental results show that *StackArmor* is practical, efficient, and provides more comprehensive protection than all prior binary- and source-level solutions.

Chapter 4

Practical Context-Sensitive Control-Flow Integrity

Control-Flow Integrity (CFI) is a well-established family of techniques to prevent control-flow diversion attacks by restricting control edges to a set of well-defined legal flows. Current CFI implementations track control edges individually, insensitive to the context of preceding edges. Recent work demonstrates that this leaves sufficient leeway for powerful ROP attacks. Context-sensitive CFI, which can provide enhanced security, is widely considered impractical for real-world adoption. Our work shows that Context-sensitive CFI (CCFI) for both the backward and forward edge can be implemented efficiently on commodity hardware. We present *PathArmor*, a binary-level CCFI implementation which tracks paths to sensitive program states, and defines the set of valid control edges *within the state context* to yield higher precision than existing CFI implementations. Even with simple context-sensitive policies, *PathArmor* yields significantly stronger CFI invariants than context-insensitive CFI, with similar performance.

4.1 Introduction

Control-Flow Integrity (CFI) [15] has developed into one of the most promising techniques to stop code reuse attacks against C and C++ programs. Typically, such attacks circumvent common defenses such as DEP/W$\oplus$X by diverting a program's control flow to a set of Return-Oriented Programming (ROP) gadgets [52; 163]. Similarly, they defeat widely deployed ASLR by either targeting gadgets at fixed (non-randomized) addresses [39], or by dynamically disclosing the addresses of randomized gadgets [171]. CFI promises to prevent all such attacks by ensuring that all control transfers conform to the program's original Control Flow Graph (CFG). In theory, CFI is very powerful and, in its purest and ideal form, provably secure against most integrity violations of the control flow [13].

Ten years after the original CFI proposal [14], however, researchers are still working to find practical CFI implementations [56; 90; 111; 136; 177; 197; 199], able to approximate the security of the purest form of CFI with acceptable performance. Common CFI solutions, including state-of-the-art binary-level implementations such as bin-CFI [199] and CCFIR [197], attempt to substantially relax constraints on the set of legal targets for both the backward edge (e.g., `ret` instructions) and forward edge (e.g., indirect `call` instructions). While doing so reduces the performance overhead to only a few percent, it also provides more degrees of freedom for attackers. Other even more lightweight CFI solutions, such as ROPecker [56] and kBouncer [136], build on heuristics and hardware support to detect anomalous control flows (which resemble ROP gadget chains) and stop many current exploitation attempts at low performance overheads. Unfortunately, a string of recent publications comprehensively shows that it is possible to circumvent all these lightweight CFI solutions with relatively little effort [49; 76; 96; 97; 158].

A key problem with traditional CFI solutions (even recent source-level fine-grained ones [177]) is that they enforce only context-insensitive CFI policies, which examine control edges in isolation and attempt to statically derive the resulting superset of all the possible targets according to the CFG. The lack of context inevitably results in weak CFI invariants, allowing attackers to freely chain edges together and form paths that are even trivially infeasible in the original CFG (e.g., returning to a function never on the active call stack [97]).

Context-sensitive CFI techniques are a promising way to address this problem, since they rely on context-sensitive static analysis to associate CFI invariants to control-flow *paths* (i.e., multiple consecutive edges) in the CFG and enforce such invariants on execution paths at runtime. The stronger security guarantees provided by context-sensitive CFI have been acknowledged as early as in the original CFI proposal, but their real-world adoption has been rapidly dismissed as impractical [14].

In this chapter, we demonstrate that Context-sensitive CFI (CCFI) can indeed be implemented in an efficient, reliable, and practical way for real-world applications. We present *PathArmor*, the first binary-level CCFI solution which enforces context-sensitive CFI policies on both the backward and forward edges. *PathArmor* relies on commodity hardware support to efficiently and reliably monitor execution paths to sensitive functions which can be used to mount control-flow diversion attacks [136], and uses a carefully optimized binary instrumentation design to enforce CCFI invariants on the monitored paths. *PathArmor*'s path invariants are derived by a scalable context-sensitive static analysis performed over the CFG on-demand, which uses caching of path verification steps to achieve high efficiency. Verification itself is also very efficient, since all the CFI checks are batched at sensitive program points. We release *PathArmor* open source.[1]

To show the practicality of our design, we prototype two context-sensitive and binary-level CFI policies (for the backward and forward edges, respectively) on top

[1] *PathArmor* is available at https://github.com/dennisaa/patharmor.

of *PathArmor*. Moreover, our framework can also serve as a general foundation for even stronger CCFI implementations, for instance using context-sensitive data-flow analysis at the source level. Even in the current setup, *PathArmor* provides a comprehensive CCFI protection system with much stronger security guarantees than traditional CFI solutions, while matching or even improving their performance. Moreover, due to its optimized design, *PathArmor* can also serve as an efficient basis for fine-grained context-insensitive CFI ($\overline{\text{C}}$CFI) policies.

Contributions Our contribution in this chapter is threefold.

- We identify the key challenges towards practical CCFI implementations and investigate opportunities to address these challenges in real-world applications and commodity platforms.
- We present *PathArmor*, a framework to efficiently support arbitrary context-sensitive and context-insensitive CFI policies on commodity platforms. To fulfill its goals, *PathArmor* relies on hardware support, binary instrumentation, and on-demand static analysis to batch even sophisticated CFI checks at the relevant sensitive points in a binary. We complement *PathArmor* with fine-grained $\overline{\text{C}}$CFI policies and simple but comprehensive (backward and forward edge) CCFI policies, making it the first practical CCFI implementation.
- We evaluate *PathArmor* on popular server applications and the SPEC CPU2006 benchmarks. Our results show that *PathArmor* can significantly restrict the number of legal control flows compared to traditional CFI solutions (-70% across all our applications, geometric mean), while yielding bounded memory usage (+18–74MB overall) and low runtime performance overhead (3% on SPEC and 8.5% on the servers, geometric mean).

4.2 Context-sensitive CFI

The general goal of every CFI solution is to allow all the control flows which occur in the interprocedural control-flow graph (CFG) defined by the programmer, and reject the largest possible fraction of the other flows as illegal [15]. This section formalizes the definition of a legal flow adopted in existing practical CFI solutions, contrasts it with the stricter definition adopted in Context-sensitive CFI (CCFI), and details the key challenges towards practical CCFI.

4.2.1 Legal flows

We model a CFG as a digraph $G = (V, E)$ where V is the set of basic blocks, and E the set of control edges in the CFG defined by the program.

Traditional CFI [15] enforces that each individual (indirect) control transfer taken by the program during the execution must match an edge in the CFG:

Context-insensitive CFI ($\overline{\text{C}}$CFI). *For each control transfer $e_i = (v_x, v_y)$ between basic blocks v_x and v_y, $\overline{\text{C}}$CFI enforces that $e_i \in E$.*

That is, $\overline{\text{C}}$CFI checks conformance to the current position in the CFG and does not distinguish between different paths in the CFG that lead to a given control transfer. For instance, consider the following two paths that both lead to function *E()*:

```
A(){ indirect call to B(); } C(){ indirect call to D(); }
B(){ indirect call to E(); } D(){ indirect call to E(); }
```

Disregarding the context would allow E to return to either B or D. However, we should only allow a return (backward edge) to B, when coming from A (via B). Likewise, we should only allow a return to D if the program got there via C.

As an example for the forward edge, suppose B and D both call E with callback argument cb_B and cb_D, respectively. When E invokes the callback, $\overline{\text{C}}$CFI would allow either target, while taking the context into consideration would allow us to (rightly) conclude that cb_B is only legal if we reached E via B.

To mimic context-sensitive behavior on the backward edge, a number of existing CFI solutions use a shadow stack [33; 54; 58; 61; 85; 145; 152; 166; 194]. However, shadow stacks are often expensive at the binary level [54; 74; 166]. Moreover, unlike CFI techniques, their security relies on the integrity of in-process runtime information, typically protected using system-enforced ASLR which is prone to probabilistic attacks against its memory protection guarantees.

All existing CFI solutions implement fully context-insensitive ($\overline{\text{C}}$CFI) policies as described above. In addition, some binary-level solutions, like CCFIR [197] or binCFI [199], further relax their $\overline{\text{C}}$CFI policies for performance. These implementations group control transfer sources and destinations based on a general definition of type, and enforce that the source and destination type match:

Practical $\overline{\text{C}}$CFI. *For each control transfer $e_i = (v_x, v_y)$ between basic blocks v_x and v_y, practical $\overline{\text{C}}$CFI ensures $x \in sources(type(e_i)) \wedge y \in sinks(type(e_i))$, where $sources(\tau)$ and $sinks(\tau)$ denote the sets of program locations having outbound or inbound edges of type τ, respectively.*

While practical $\overline{\text{C}}$CFI precludes malicious control transfers like jumps into the middle of a function, or returns to non-call sites, attackers can still successfully mount powerful attacks using gadgets which adhere to the imposed type restrictions [48; 49; 76; 96; 158].

CCFI provides stronger CFI invariants than both practical and ideal $\overline{\text{C}}$CFI. Rather than considering control transfers individually, CCFI examines each transfer in the context of recently executed transfers:

CCFI. *Given a path $p = (e_1, e_2, \ldots, e_n)$ of control transfers leading to a given program point P, CCFI verifies the validity of P by checking that $\forall i \in \{1, 2, \ldots, n\}$, edge e_i is consecutively valid in the context of all preceding CFG edges $e_1, \ldots, e_{i-1}$.*

Since CFI checks are enforced *per path* (not *per edge*), CCFI can enable arbitrarily powerful context-sensitive policies on both the backward and forward edges.

4.2.2 Challenges

In this section, we discuss the three fundamental challenges towards practical CCFI. Subsequently, the remainder of this chapter presents *PathArmor*, the first practical binary-level solution to these problems.

C1: Efficient path monitoring A major challenge in implementing a practical CCFI solution is identifying an efficient mechanism to constantly monitor paths of executed control flow transfers at runtime. Other than imposing minimal performance overhead, the path monitoring mechanism should also be reliable. That is, neither the program nor the attacker should be able to tamper with the recorded data. All these requirements were considered the key obstacle to the real-world adoption of context-sensitive CFI in the original CFI proposal [14]. To address this challenge, *PathArmor* relies on branch recording features available in modern x86-64 processors to implement efficient and reliable path monitoring.

C2: Efficient path analysis To verify the validity of a path to a given program point P, CCFI needs to statically analyze the CFG and identify the legal paths to P in a context-sensitive fashion, validating all the edges in the path. The naive solution, statically enumerating *all* legal paths to *all* the relevant program points, cannot scale efficiently, with the number of paths growing exponentially with $|V|$ and $|E|$.

This *path explosion* problem is well known in several domains (symbolic execution, among others [109]). Even focusing our static analysis on a particular program point and sequence of indirect control transfers derived by runtime information only partially eliminates this problem. Path explosion can still occur between any two indirect control edges, especially in the presence of loops and long sequences of direct jumps and calls.

To address this challenge, *PathArmor* relies on an on-demand, constraint-driven context-sensitive static analysis over a normalized CFG representation. The constraints, derived by runtime information recorded by our path monitor, allow our context-sensitive path analysis to efficiently scale to arbitrarily large and complex CFGs.

C3: Efficient path verification To detect control-flow diversion attacks, CCFI needs to carefully select program points to verify the current execution path for validity. To provide strong security guarantees, path verification needs to be performed in all execution states that are potentially harmful. The naive solution, performing path verification after every executed control transfer, is clearly inefficient and scales poorly with path length.

To address this challenge, *PathArmor* relies on a kernel module to efficiently verify only the paths to well-defined sensitive functions in the program. While the verification still needs to run for each path to these functions encountered during execution, *PathArmor* aggressively caches verification results to minimize the resulting

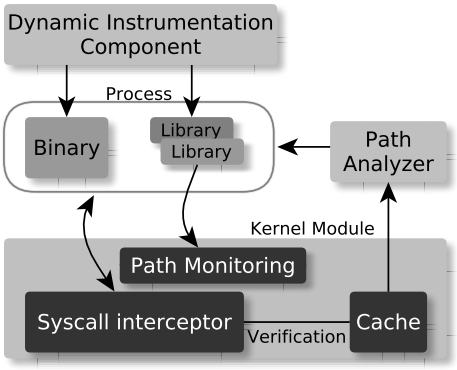


Figure 4.1: Overview of *PathArmor*.

impact on runtime performance. Since the number of paths to sensitive functions is limited in practice (as shown in Section 4.5.3 for popular server programs), caching is effective in amortizing path verification costs throughout the execution.

4.3 PathArmor Overview

Figure 4.1 presents a high-level overview of *PathArmor* and details its three main components: (1) a kernel module, (2) an on-demand static analyzer, and (3) an instrumentation component.

PathArmor relies on a *kernel module* which provides a Branch Record core to support per-thread control transfer monitoring in multi-process and multi-threaded programs. For this purpose, our module uses the 16 *Last Branch Record* (*LBR*) registers available in modern Intel processors and only accessible from ring 0. This strategy allows our module to monitor paths of (16) recently exercised control transfers in an efficient and reliable way (addressing **C1**).

In addition to path monitoring, our kernel module triggers path verification steps upon security-sensitive system calls issued by the program (and other special sensitive operations, as detailed later). To further improve the performance of path verification, our module also maintains a path cache, which stores hashes of previously verified paths and eliminates the need to enforce more expensive CCFI checks at each verification (addressing **C3**). We discuss our kernel module in more detail in Section 4.3.1.

Once the kernel module is loaded, protected program binaries run with our *dynamic instrumentation component*, which instruments the binary at load-time. This component first starts our *path analyzer*, an external trusted component which runs in the background and waits for path verification requests from the kernel module via a dedicated upcall interface.

To satisfy path verification requests, our analyzer receives all the necessary LBR-based path information, as well as constraints on indirect and interprocedural direct control transfers, from our kernel module and performs static analysis on-demand to enforce our CCFI policies. For this purpose, the analyzer reconstructs the CFG of the target binary and preprocesses it with a preliminary *CFG reduction* step that prunes all the irrelevant intraprocedural edges from the control-flow graph. This step and our constraint-based strategy eliminate all the intraprocedural and interprocedural path explosion threats, ensuring a scalable on-demand path analysis (addressing **C2**).

After determining whether a path is valid, our analyzer reports its findings back to the kernel module, which, in response, stops the program (if verification fails) or populates the path cache (otherwise). We elaborate more on our path analyzer in Section 4.3.2.

After initializing the path analyzer, our instrumentation component sets up an in-program communication channel with the kernel module to enable (and later manage) path monitoring for the target binary. Finally, we instrument the binary according to a predetermined sensitive path termination policy.

PathArmor can be configured to verify either full paths to sensitive system calls or truncate paths at the library call interface. Our current implementation uses the latter mode of operation by default, given that the LBR in its current incarnation on commodity hardware can only record the 16 most recently executed control transfers, and allowing branch tracing inside the libraries can potentially "pollute" paths and thus erase program context. This observation was also made in prior work [136].

The tradeoff (which can be reconsidered with future hardware extensions) is that *PathArmor*'s default configuration can defend against control-flow diversion attacks only in the program, excluding attacks originating from vulnerabilities in libraries. We evaluate the feasibility of future in-library path tracking in Section 4.5.5. We discuss our instrumentation component in more detail in Section 4.3.3.

4.3.1 Kernel Module

As illustrated in Figure 4.1, the kernel module consists of two main components: (1) a system call interceptor that sends validation requests (via a cache) to the on-demand static analyzer, and (2) a Branch Record core (LBR API) that monitors and records branches occurring within the protected binary's main address space.

4.3.1.1 System Call Interception

As mentioned in Section 4.2.2, *PathArmor* limits verification to a small number of security sensitive path endpoints in order to maintain minimal runtime overhead. In particular, these endpoints consist of a set of dangerous system calls an attacker requires to deploy a meaningful exploit, like `exec` and `mprotect` (and other sensitive operations, see Section 4.3.3.3). We refer to them as sensitive calls. Like other work in this area [56], we propose to monitor only these dangerous endpoints, rather than every possible library and system call.

To intercept system calls at runtime, the kernel module installs an alternative syscall handler. When our target requests execution of a dangerous system call, we pause execution, collect LBR data, and forward it to the on-demand static analyzer in user space. If the analyzer returns `true` (meaning that the path was found in the CFG and thus is valid), the kernel module stores a hash of the path in a cache data structure before permitting the system call. We use cryptographically secure second-preimage resistant[2] hash algorithms (MD4 in our evaluation) to prevent path crafting attacks, where attackers craft an invalid path with a hash that collides with that of a valid path.

If the exact same path is executed a second time, *PathArmor* finds its hash in the cache, obviating the need for a new validation. In general, *PathArmor* only sends a request to the on-demand static analyzer if no match was found in the cache. This limits the amount of overhead caused by traversing the CFG.

In the event that the on-demand static analysis returns `false` (no valid path was found in the CFG), the module stops the program and reports that an attack was detected. With the LBR data still in place, this can also help pinpoint the exact location of the attack.

4.3.1.2 Branch Recording

In addition to path verification, the kernel module provides a Branch Recording core that implements support for tracking branches on a per process-thread basis. In addition, it exposes an interface to the instrumented libraries that is used to disable branch recording during library execution. *PathArmor* can record branches either using the LBR (the current default) or Intel's Branch Trace Storage (BTS) feature. Although prior work has shown that BTS imposes a significant performance slowdown (typically in the order of 20–40x [172]), its "unlimited" nature provides a useful means to enable in-library tracking, and to measure how many LBR registers are required to approach optimal security (Section 4.5).

Ideally, we would configure the Branch Recording core to collect only indirect branches (indirect jumps, indirect calls and returns), as only these branches can be modified by an attacker. However, armed only with information about indirect branches exercised by the program, we cannot eliminate the path explosion problem. To solve this issue, we instruct the Branch Recording core to keep track of direct call instructions as well, which can be used by the on-demand static analysis to eliminate path explosion, rendering *PathArmor* efficient in practice. We elaborate more on this in Section 4.3.2.

To disable branch recording during library execution, we expose two `ioctl()` requests to libraries: `LIB_ENTER` and `LIB_EXIT`. The dynamic instrumentation component detailed in Section 4.3.3 inserts these requests for each used library function by instrumenting their entry and exit points. We discuss related implementation

[2]For a second-preimage resistant hash algorithm h and input x, it is computationally hard to find a second input $x' \neq x$ such that $h(x) = h(x')$.

challenges, such as how to enable branch recording again for callbacks, in depth in Section 4.4. Note that attackers cannot abuse `ioctl` requests to disable *PathArmor*, as discussed in Section 4.6.3.

4.3.2 Path Analyzer

The path analyzer verifies at runtime (but using the *static* CFG) if a particular path observed at an endpoint is valid. It consults the CFG of the binary and searches it for the path. We now discuss the three main steps of this analysis: *CFG generation*, *CFG reduction* to eliminate the path explosion problem, and *path validation*.

4.3.2.1 CFG Generation

To validate a path, *PathArmor* requires an accurate CFG of the protected binary. To obtain a CFG, we use existing binary analysis frameworks to disassemble and analyze binaries, as detailed in Section 4.4. Though previous work shows that the CFG can be obtained with reasonable accuracy [46; 198] (as confirmed by our findings in Chapter 6), *PathArmor* still chooses to err on the safe side, tolerating potential errors by *overestimating* the CFG when necessary. In the worst case, this may cause *PathArmor* to accept invalid paths, but it will never reject legitimate ones. Furthermore, *PathArmor* implements indirect edge resolution policies to augment a CFG walk with indirect edges in a context-sensitive manner. If these policies fail, we resort to a fine-grained context-insensitive policy instead [197; 199].

For *backward edges* (i.e., returns), our policies implement a fully context-sensitive resolution strategy, to which we refer as *call/return matching*. This strategy emulates a runtime call stack by tracking call and return edges as these are encountered.

For *forward edges* (i.e., indirect calls), our current prototype supports a simple context-sensitive strategy which resolves code pointers propagated across caller-callee pairs with no contrived data flow. This policy lets us unambiguously resolve indirect call sites, at which call targets are loaded as constants and passed as a callback argument. However, our path abstraction, in principle, enables much more complex context-sensitive extensions. We evaluate the additional security provided by forward-edge context-sensitivity in Section 4.5. In cases where our current policy is unable to trace a code pointer (e.g., in case of a long-lived code pointer stored on the heap), *PathArmor* resorts to a $\overline{\text{C}}$CFI policy which matches all indirect call sites with all the functions that have their address taken.

Indirect jumps, in turn, are conservatively resolved by the underlying binary analysis framework. We also implement a strategy to augment the precision of indirect jumps found in PLT entries. The CFG is updated with data received from the instrumentation component, enabling unambiguous target resolution. We discuss this resolution in more detail in Section 4.3.3.1.

4.3.2.2 CFG Reduction: Addressing Path Explosion

As discussed in Section 4.2, static analysis of large CFGs may lead to a path explosion problem, where the number of paths to explore increases exponentially with the exploration depth. *PathArmor* takes two measures to address the problem and perform efficient path verification.

First, as a preprocessing round, *PathArmor* performs a *CFG reduction* step that significantly prunes the CFG, and preserves reachability relations with respect to indirect edges and interprocedural direct edges. This step finds all possible paths of direct edges between entry and exit points of each function, and then collapses these paths down to a single edge between each entry point and the exit points reachable from it. This makes the subsequent search much faster, as needless (re-)explorations of direct edges can be avoided (e.g., loops).

Second, call/return matching (discussed in Section 4.3.2.1) allows us to recognize and discard impossible paths, such as paths that call a function from one call site, and subsequently return to another call site. Without call/return matching, the path search would have no way of identifying such mismatches.

4.3.2.3 Path Verification

The path analyzer is responsible for verifying the validity of a given path. The path is an LBR state containing direct and indirect calls, indirect jumps, and returns. To verify whether it is valid, the analyzer performs a Depth-First Search (DFS) on the CFG to find a path that contains the provided edges in the same order as they were recorded by the LBR. A recorded path is thus considered valid iff: (1) all edges in the LBR state exist within the CFG, and (2) every pair of two consecutive edges can be linked together via a valid path of direct edges within the CFG.

To ensure that the search terminates quickly if a path does *not* exist (e.g., the LBR state is malicious), the DFS does not follow indirect edges or direct call edges. Following such edges would not make sense, because by definition, such edges would be in the LBR state if they occurred on the path under analysis.

Note that due to our use of direct call recording in the LBR and our CFG reduction step, the DFS cannot get stuck on cycles within the CFG. Indeed, it first consults the LBR for the oldest recorded branch, from a basic block `A` to a basic block `B`, and then loops over all possible outgoing edges of `B` to see which one to follow. Due to the CFG reduction, direct jump edges are collapsed, so the outgoing edges of `B` are all either indirect edges or direct call edges. For each edge the DFS examines, it checks whether this edge is the next recorded branch. If this does not hold, it tries the next edge, until it finds one that matches the following LBR state. From here, it restarts analysis, starting from this new edge. This process continues until the last edge (the most recently recorded branch) is found.

4.3.3 Binary Instrumentation

The instrumentation component consists of a library instrumentation module (a special loader), and a dynamic binary instrumentation module. Its main objectives are (1) collecting address offsets (for both libraries and the target program) and passing these to the static analysis component, (2) instrumenting libraries such that they disable LBR tracking before their execution starts and re-enable it again once finished, and (3) starting the actual target process. In addition, the instrumentation component opens a communication channel with the kernel module that is used by the instrumentation code to communicate with the Branch Recording core.

Although our rewriter is dynamic in the sense that it inserts instrumentation at runtime (when the binary or a library is loaded), it is not a pure dynamic approach as described in Chapter 2.3. That is, rather than instrumenting on the fly, we completely rewrite each module as soon as it is loaded. Therefore, our approach does not suffer from the high runtime overheads mentioned in Chapter 2.3.

4.3.3.1 Loader

The loader sets up the *PathArmor* environment before starting the protected binary. It is implemented as a pre-loaded shared library using `LD_PRELOAD`, and instruments the target binary's `main()` function. This hook then opens an `ioctl()` interface with the LBR API of the kernel module, which is used by the inserted code snippets to notify the kernel module of specific events (e.g., `LIB_ENTER`).

In addition, the loader collects the program's PLT and GOT entries as well as the base addresses of the different libraries that are in place. It passes this information via the kernel module to the on-demand static analyzer so that it can distinguish calls to library functions from branches within the program's main address space. To this end, the target program is started with `LD_BIND_NOW=1`, causing the dynamic linker to resolve all symbols at program startup instead of using the default lazy function call resolution behavior.

4.3.3.2 Rewriter

By default, *PathArmor* truncates paths at library boundaries. For this purpose, our dynamic instrumentation module rewrites all library functions that are used by the program (i.e., those listed in the PLT, as well as those dynamically loaded using `dlsym()`). The inserted instrumentation ensures that library functions first send an LBR disable request to the LBR API in the kernel module before executing, and finish with an LBR enable request before returning to the program.

A library function may at some point invoke a callback handler which may or may not reside in the target's address space. If we do not re-enable the LBR again on callbacks, a bug in the callback handler could still be exploited by an attacker as we lose vital information on executed paths. To overcome this problem, we apply another round of dynamic instrumentation, this time to ensure that whenever a call-

back is invoked, LBR tracking is enabled again. We discuss this process in more detail below.

The dynamic instrumentation module of the initialization component performs necessary rewriting tasks at load time (when dynamically linked libraries are available) and at runtime (every time a new shared library is dynamically loaded into memory). Note that we only need to instrument shared libraries. No instrumentation is required in the protected application, leaving the original binary intact.

4.3.3.3 Callbacks

As mentioned above, a second dynamic instrumentation round is required in order to re-enable branch recording when a library function invokes a callback that lies in the program's binary (e.g., `qsort()`). Instrumenting callback sites is done by looping over all shared library functions and searching for indirect call instructions. For each indirect call, we insert a short instrumentation snippet that (1) tests if the target of the indirect call lies in the protected program's address space, and (2) if this is the case, wraps the call instruction between two `ioctl()` system calls that notify the kernel module that a callback function is entered or exited (`CALLBACK_ENTER` and `CALLBACK_EXIT`, respectively).

When the kernel module receives a `CALLBACK_ENTER` request, it pushes the current LBR state (i.e., the contents of the LBR registers as seen before the library function that performs the callback) to an internal stack of LBR contexts. When the callback exits (`CALLBACK_EXIT`), the kernel module pops the top of this LBR stack back into the actual registers. To support code that forks within a callback, the kernel module copies the stack of LBR contexts to the newly created process, so that parent and child both receive consistent branch records.

Observe that signals are essentially a specialized form of callbacks and can be processed in a similar manner. The only difference is that instead of instrumenting code, we install a hook on the kernel's signal delivery function. This hook executes before control returns to the signal handler, allowing us to save the current LBR context so that it can be restored upon the `sigreturn` system call.

Our approach of switching LBR contexts at the moment callback handlers are invoked could potentially allow an attacker to install a different handler than normally enforced by the CFG. Consider the case where an attacker exploits a memory corruption bug to install a callback handler that fits his needs. Without additional security measures, this operation may go unnoticed (control-flow diversion happens indirectly in the kernel or in the libraries).

To overcome this situation, *PathArmor* considers signal handler registration and LBR management operations (i.e., push context, pop context) to be sensitive operations themselves (thus triggering LBR validation). Moreover, *PathArmor* always copies the last branch entry during LBR context switching, storing it as the first branch entry for the new context. This enables our on-demand static analysis to apply our indirect edge resolution policies on the library-originated indirect call edge

before allowing the callback. A symmetric approach is used to avoid false positives for library-originated function pointers (e.g., returned by `dlsym()`) which are used for indirect call invocations by the program. Our static analyzer resolves the library target referred to by the function pointer in a dedicated way without needing more sophisticated modular CFI policies [132].

4.3.3.4 Special Constructs

Similarly to our callback support, *PathArmor* supports the `longjmp()` construct by implementing a special handler for this in the kernel module: for each `setjmp()`, the kernel stores the existing LBR contents along with the provided `env` argument. Upon a `longjmp()`, our module verifies the LBR contents, flushes them and restores the LBR with the appropriate state stored earlier (matching `env`). As with callbacks, we use our dynamic instrumentation component to insert dedicated `SETJMP` and `LONGJMP` `ioctl()` requests for each such construct.

4.4 Implementation

We implemented *PathArmor* on Linux v3.13 for x86-64 with support for multiprocess and multithreaded applications. Our kernel module is implemented as a standard loadable module for the Linux kernel in 1,752 SLOC. The on-demand static analysis component is implemented as a plugin for the Dyninst binary analysis and rewriting framework [31] in 6,741 SLOC overall. The library instrumentation is implemented as another Dyninst plugin in 1,625 SLOC.

To intercept sensitive system calls, we install an alternative syscall handler by overwriting the `MSR_LSTAR` register. *PathArmor* forwards most system calls directly to their vanilla implementation, imposing little to zero extra overhead. However, we consider a total of seven system call families to be dangerous, and start verification whenever these are encountered: `mprotect` and the `mmap` family (which can be used to disable DEP/W$\oplus$X), and the `exec` family (which can be used to start a malicious command) are obvious choices and have been considered in prior work in the area [56]. To address the challenges related to signal handling as detailed in Section 4.3.3.3, *PathArmor* also intercepts the `sigaction` and `sigreturn` system calls. *PathArmor* can also be configured to protect I/O system calls, to prevent attacks like data leaks and script injection in (for instance) web servers.

Since Linux v3.13 does not support per-task LBR context management, we implemented it to avoid pollution from other processes. We used the standard preempt notifier functionality (`preempt_notifier_register`) provided by the Linux kernel to install hooks on context-switches. During a context-switch-out (`sched_out`), *PathArmor* stores the LBR state of the current process into an LBR process table, to restore it later when the thread is scheduled in again (`sched_in`). This approach allows *PathArmor* to support binaries that use multithreading.

Our current *PathArmor* prototype is based on the Dyninst binary rewriting framework, and as a consequence does not support C++ exceptions. This limitation is not fundamental to *PathArmor*, and can be addressed in future work with additional engineering effort.

4.5 Evaluation

We evaluate *PathArmor* on a workstation equipped with a 3.10GHz Intel i5-2400 CPU and 8GB of RAM, running Ubuntu 14.04 with Linux kernel 3.13 (x86-64). To measure the worst case performance impact of *PathArmor*, we default to non-library mode, but we evaluate the effects of enabling in-library tracking in Section 4.5.5.

We evaluate *PathArmor* on several popular Linux server applications. These are widely used in the research community for evaluation purposes, and are also popular exploitation targets for both local and remote attacks. Moreover, they naturally contain a relevant number of security-sensitive functions and can greatly benefit from the protection guarantees provided by *PathArmor*. Our server test suite consists of three FTP servers (`vsftpd` v1.1.0, `proftpd` v1.3.3, and `pure-ftpd` v1.0.36), two web servers (`nginx` v0.8.54 and `lighttpd` v1.4.28), an SSH server (`opensshd` v3.5), and an email server (`exim` v4.69). We also evaluate *PathArmor*'s performance on SPEC CPU2006.

To benchmark our web servers, we used the Apache benchmark [1] configured to issue 25,000 requests with 10 concurrent connections and 10 requests per connection. To benchmark our FTP servers, we used `pyftpbench` [5] configured to open 100 connections and request 100 files of 1KB each per connection. Finally, to benchmark `opensshd` and `exim`, we used the OpenSSH test suite [4] and a homegrown script which repeatedly launches `sendemail` [7], respectively. We configured all our applications and benchmarks with their default settings. We ran all our experiments 11 times, checking that the CPUs were fully loaded throughout our tests, and report the median. We observed only marginal variations across runs.

Our evaluation answers four key questions: (1) *Security*: Does *PathArmor* significantly improve security against control-flow diversion attacks compared to existing CFI techniques? (2) *Memory usage*: How much memory overhead does *PathArmor* induce? (3) *Analysis time*: Does *PathArmor*'s static analysis complete in reasonable time? (4) *Runtime performance*: Does *PathArmor* yield low runtime overhead while protecting a relevant set of sensitive functions?

4.5.1 Security

To evaluate the security guarantees of *PathArmor* and, in particular, the improvements in CCFI over existing $\overline{\text{C}}$CFI techniques, we measured the strength of the CFI invariants extracted by our static analysis and enforced by *PathArmor* at runtime. For this purpose, we instructed our static analyzer to generate CFI statistics during the execution of our benchmarks and compare the results against fully context-insensitive

	Functions	CFG $\lvert V\rvert$	CFG $\lvert E\rvert$	LBR (Avg) $\frac{\lvert E_{\text{IB}}\rvert}{\lvert E\rvert}$	LBR (Avg) $\frac{\lvert E_{\text{IF}}\rvert}{\lvert E\rvert}$	LBR (Avg) $\frac{\lvert E_{\text{DF}}\rvert}{\lvert E\rvert}$
`vsftpd`	sa,mm,mp	4,052	9,269	0.33	0.23	0.44
`proftpd`	sa,sg,ki,mm	29,682	210,489	0.38	0.27	0.35
`pure-ftpd`	sa,	5,702	19,910	0.32	0.33	0.35
`lighttpd`	sa,sg,ki,m4,el	7,380	38,006	0.38	0.22	0.40
`nginx`	sa,ra,ki,m4,ee	26,029	432,829	0.45	0.20	0.35
`opensshd`	sa,sg,mm,el,ev,ee	14,749	63,644	0.38	0.26	0.36
`exim`	sa,sg,ki,ev,ee	37,906	167,867	0.34	0.28	0.38

Table 4.1: Control flow properties of the evaluated programs. The *Functions* column shows the sensitive functions which require protection: sa=`sigaction`, sg=`signal`, ra=`raise`, ki=`kill`, mm=`mmap`, m4=`mmap64`, mp=`mprotect`, el=`execl`, ev=`execv`, ee=`execve`. The *CFG* group shows the number of nodes and edges in the CFG. The *LBR* group shows the average number of indirect backward edges, indirect forward edges, and direct forward edges in the LBR during execution of our tests.

CFI policies. Note that these statistics are intended only to provide a clear picture of the strength of *PathArmor*'s invariants compared to other CFI solutions. As such, the following discussion focuses on a relative comparison across CFI implementations, rather than on absolute numbers.

Tables 4.1 and 4.2 present the resulting statistics. Table 4.1 quantifies the control flow properties of the evaluated binaries, showing their sensitive functions, inter-procedural CFG information, and prevalence of each edge type in the LBR during execution. The ICFG information is generated by our analyzer with fully context-insensitive indirect edge resolution policies. The LBR statistics show the fraction of indirect backward edges (*IB*), indirect forward edges (*IF*), and direct forward edges (*DF*) in the LBR averaged across all the sensitive function calls during execution of our benchmarks. Table 4.2 compares the control flows permitted by *PathArmor* (CCFI) to existing coarse-grained $\overline{\text{C}}$CFI and fine-grained $\overline{\text{C}}$CFI techniques.

As shown in Table 4.1, the number of sensitive functions as well as the number of nodes and edges in the CFG ($|V|$ and $|E|$, respectively) varies greatly across applications, reflecting their different internal structure. Moreover, the overall distribution of edge types in the LBR is relatively stable across applications, with backward edges dominating (indirect) forward edges (37% vs. 25% geometric mean). Encouragingly, direct forward edges (which, while necessary to scalably enforce our CCFI policies, also decrease the number of LBR entries subject to CFI enforcement) have a significant but non-dominant impact in practice (37% geometric mean).

In Table 4.2, the $|G|$ columns report the average number of targets (and thus gadgets) allowed by the given CFI policy for each indirect edge observed in the LBR. The $\min[G_{\text{Len}}]$ column provides more qualitative information on the resulting CFI-allowed gadgets, by averaging the minimum allowed gadget length for each edge observed in the LBR. As shown in the table, CCFI yields a significantly lower average number of gadgets compared to coarse-grained and fine-grained $\overline{\text{C}}$CFI (-99.7% and -61.6% geometric mean, respectively).

	$\overline{\text{C}}\text{CFI}_{\text{cg}}$ (Avg)		$\overline{\text{C}}\text{CFI}_{\text{fg}}$ (Avg)		CCFI (Avg)	
	$\lvert G\rvert$	$\min[G_{\text{Len}}]$	$\lvert G\rvert$	$\min[G_{\text{Len}}]$	$\lvert G\rvert$	$\min[G_{\text{Len}}]$
vsftpd	543.26	3.5	3.17	8.0	1.27	13.1
proftpd	3249.55	2.2	19.96	4.0	6.11	7.5
pure-ftpd	403.57	2.2	5.37	4.5	1.94	5.1
lighttpd	561.00	2.0	2.77	4.8	1.00	5.5
nginx	1482.08	2.8	23.40	9.3	14.90	9.9
opensshd	1725.20	2.1	16.02	3.9	4.37	7.2
exim	2588.53	2.2	25.10	4.4	11.05	11.1

Table 4.2: Comparison of permitted control flows in coarse-grained, fine-grained, and context-sensitive CFI (average number of legal targets and minimum gadget length).

Additionally, Figure 4.2 shows the CDF of the number of allowed targets for the two applications with the most sensitive calls (exim and proftpd). We observe similar trends for the other applications. The CDF confirms that CCFI allows very few targets for the vast majority of control flow transfers. For instance, on exim, 98% of control transfers have less than 13 legal targets, compared to around 86% for fine-grained $\overline{\text{C}}$CFI and 72% for coarse-grained $\overline{\text{C}}$CFI (the common policy for binary-level CFI solutions [197; 199]). This demonstrates the effectiveness of our context-sensitive CFI policies, which can drastically restrict the number of legal targets for most LBR entries.

Our improvements are also reflected in the overall complexity of the gadgets left to the attacker, with the average minimum allowed gadget length ($\min[G_{\text{Len}}]$) substantially increasing compared to the coarse-grained and fine-grained versions of $\overline{\text{C}}$CFI ($+245\%$ and $+53\%$ geometric mean, respectively). In general, shorter gadgets are easier to fit together and are more preferred than longer gadgets for building a ROP chain. By reducing the possible indirect edge targets, the attacker's gadget arsenal is diminished and the bar for exploitation raised.

As an example, Table 4.2 shows that the reduction in the average number of indirect edge targets from 17 to 2.3 for exim results in an increase of the average number of instructions in the shortest allowed gadgets from 4.4 to 11. With CCFI, a deeper gadget analysis also reveals a significant increase in the average number of register accesses in the shortest allowed gadgets compared to the coarse-grained and fine-grained $\overline{\text{C}}$CFI policies. The geometric mean register access counts for the coarse-grained $\overline{\text{C}}$CFI, fine-grained $\overline{\text{C}}$CFI and CCFI policies are 1.3, 4.5 and 7.7, respectively. This confirms the increased gadget complexity under CCFI policies.

To evaluate the effectiveness of the particular CCFI techniques implemented in *PathArmor*, we also examine the impact of context sensitivity on both the forward and backward edge in more detail. For this purpose, we first compare our (static) backward-edge CCFI policy to that enforced by a (dynamic) shadow stack, the only known (runtime) solution which mimics context-sensitive control-flow policies (though only on the backward edge, and using tamper-prone and more heavyweight instrumentation at the binary level). For a fair comparison, we focus our measurements on the fraction of backward edges observed in the LBR which are restricted

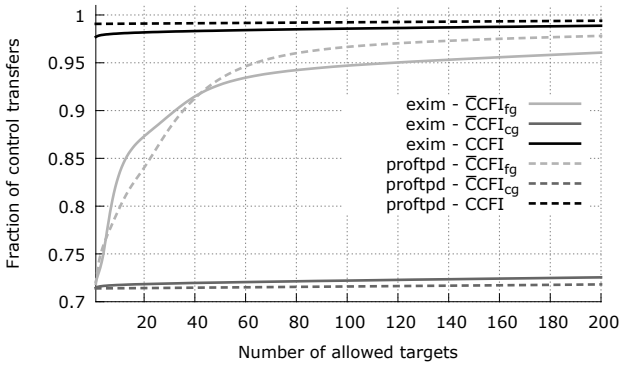


Figure 4.2: CDF of gadgets permitted by $\overline{\text{C}}$CFI and CCFI for `exim` and `proftpd`.

to only one target (in a fully context-sensitive fashion) by our CCFI techniques. We also use Intel's BTS feature to simulate an LBR of arbitrary size, overcoming the restrictions imposed by commodity hardware.

Figure 4.3 presents our results for increasing LBR sizes and the two applications with the most sensitive calls (`exim` and `proftpd`). We observe similar trends for the other applications. On commodity hardware (16 LBR entries), *PathArmor* can enforce a single target for nearly 75% of the backward edges observed in the LBR. In the remaining cases, the limited LBR size causes *PathArmor* to lose program context and resort to $\overline{\text{C}}$CFI policies. While the current LBR size limit prevents *PathArmor* from fully reaching the ideal shadow stack performance (100%), these results are still encouraging given the small default LBR size. In addition, Figure 4.3 shows that future hardware extensions can help fill the gap, e.g., enforcing a single target in 90% of cases with 70 LBR entries.

To evaluate the effectiveness of our forward-edge CCFI policy, we examine the reduction in the number of allowed indirect call targets caused by context sensitivity. Due to the very limited number of indirect call entries in the LBR for our test programs (which rarely use indirect calls close to sensitive function points), we did not observe any significant reduction in our experiments. Therefore, to generalize our results and eliminate any application-specific bias, we applied our policy to all the code paths (rather than just those seen at runtime). This still results in a relatively small reduction overall (less than 5% in most cases). However, this is expected, given that our current binary-level forward-edge CCFI policy is very simple (only propagating function pointers passed in call arguments in a straightforward way), and only intended to demonstrate the practicality of implementing arbitrary forward-edge CCFI policies in *PathArmor*. To examine the potential for more sophisticated forward-edge CCFI policies, we approximated an *ideal* binary-level context-sensitive forward-edge analysis using higher-level language semantics, implemented on top of LLVM 2.9 Data Structure Analysis (DSA) [113].

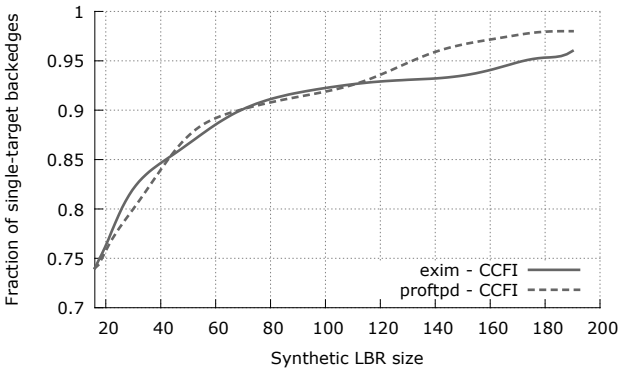


Figure 4.3: Fraction of single-target backward edges for CCFI when simulating an increasingly large LBR (`exim` and `proftpd`).

Table 4.3 shows the effect of the resulting forward-edge CCFI policy on our set of server programs. The policy causes a significant geometric mean reduction of 22% in the average number of indirect call targets. The reduction varies depending on the context-sensitive function pointer resolution accuracy. For `vsftpd`, we obtain a reduction of 62%, while numbers decrease for applications with more complex pointer resolutions. We believe these results are encouraging, simulating research on more sophisticated forward-edge CCFI policies for which *PathArmor* can serve as a basis. Moreover, DSA's flow-insensitive and unification-based design aggressively merges data-flow information, improving speed but also resulting in overly conservative results [113]. In addition, due to implementation limitations, DSA is known to produce even more conservative, and thus pessimistic, results on modern LLVM releases [2]. Thus, an updated version of DSA (or a more precise, but also less scalable analysis) would likely yield substantially improved forward-edge results.

Overall, our analysis shows that CCFI is effective in generating robust CFI invariants to defend against even sophisticated control-flow diversion attacks. While attacks are still theoretically possible (and they might be even for an ideal CCFI solution), the adoption of context sensitivity significantly limits the quantity and quality of gadgets available to the attacker. This is in stark contrast, for example, with unrestrictedly allowing simple *call-site gadgets*, which have been used to mount attacks against prior $\overline{\text{C}}$CFI techniques [97].

4.5.2 Memory Usage

PathArmor instrumentation must inherently keep track of analyzed paths and metadata for verifying the validity of paths. Thus, it increases memory usage at runtime. To evaluate this impact, we measured the physical memory used by instrumented applications compared to the baseline. Deploying our kernel module alone has a constant and marginal memory usage impact (+1MB). Our static analyzer, in turn,

	#icalls	$\frac{\text{targets}_{cs}}{\text{targets}_{ci}}$
`vsftpd`	6	0.38
`proftpd`	120	0.99
`pure-ftpd`	11	1.00
`lighttpd`	66	0.84
`nginx`	271	0.82
`opensshd`	131	0.82
`exim`	99	0.89
geomean	56	0.78

Table 4.3: Fraction of legal indirect targets for (ideal binary-level) context-sensitive versus context-insensitive forward-edge CFI.

yields a memory usage impact proportional to the size of the CFGs under active analysis, resulting in an increase of +18–74MB across all our applications.

More important is to assess the memory usage impact of our path caching strategy, given that caching static analysis results is important to minimize the performance impact on instrumented applications. Encouragingly, our measurements indicate a very small memory usage impact induced by our in-kernel path cache, resulting in a worst-case increase of only 2KB across all our applications during the execution of our benchmarks. This suggests that our path caching strategy is practical even for applications which periodically issue several different sensitive function calls, and even provides evidence that deploying a system-wide path cache that persists across application restarts (thus eliminating cache warmup-phase penalties for applications with strong real-time guarantees) may be a realistic option.

4.5.3 Analysis Time

PathArmor's on-demand path analysis translates to increased application runtime. To evaluate the resulting impact, we measured the time spent in our analyzer during the execution of our benchmarks. Table 4.4 presents our results.

The second group of columns details the total and average analysis time measured across all the paths analyzed. As shown in the table, the average time spent in our analyzer to inspect each path is relatively low (3ms geometric mean, with only marginal variations). This demonstrates that our optimizations (pre-normalizing the CFG and recording direct forward edges in the LBR) are effective in implementing a scalable context-sensitive path analysis even for large and complex CFGs.

In addition, the total time spent in our analyzer is marginal compared to the total benchmark runtime (49ms geometric mean versus several seconds). This shows the effectiveness of our path cache which, as reported in Table 4.4, was consulted thousands of times with only dozens of misses for most applications. We elaborate on the end-to-end impact of our on-demand path analysis strategy on runtime performance in the next section.

	Time (ms)		Cache Stats	
Server	*Total*	*Avg*	*#Misses*	*#Hits*
`vsftpd`	24	3	9	2,283
`proftpd`	140	4	39	2,495
`pure-ftpd`	56	2	27	1,915
`lighttpd`	28	2	13	2
`nginx`	24	5	5	10
`opensshd`	52	2	22	49
`exim`	100	3	40	1,871
geomean	49	3	18	213

Table 4.4: Path analysis time and runtime cache statistics.

4.5.4 Runtime Performance

To evaluate the impact of *PathArmor*'s instrumentation and path verification strategies on runtime performance, we measure the time to complete the execution of our benchmarks and compare against the baseline. Table 4.5 presents our results. The second group of columns details the normalized runtime across a number of *PathArmor* configurations. The *LBR only* configuration refers to *PathArmor* solely deploying its kernel module and saving/restoring the current LBR state at application thread context switch time. As shown in the table, this configuration introduces marginal performance impact (2.5% geometric mean). The overhead is somewhat more pronounced in the +*LInstr* and +*CBInstr* configurations (6.6% and 6.7% geometric mean), which additively account for our library entry point and callback instrumentation, respectively, but omit the path verification step in our kernel module. The +*PathVer* configuration, finally, refers to the default *PathArmor* setup, enabling full instrumentation and path verification using our on-demand static analyzer. As shown in the table, our cache-aware path analysis has relatively little impact on runtime performance (+1.7% geometric mean), resulting in the final average runtime overhead of 8.5% (geometric mean).

To shed some light on the key factors contributing to the performance overhead, we also instructed *PathArmor* to report statistics on the runtime events of interest, as shown in the third group of columns in Table 4.5. Our results confirm that library calls (#*LCalls*) are the most prevalent contributing factors in the mean case, also inducing the worst-case performance impact on `lighttpd` (27.3%). More aggressively instrumented operations like callback invocations (marginal, not reported in table), sensitive function calls (#*SCalls*) and signals (#*Signals*) have a less prominent impact and can thus be better amortized over the execution.

To obtain standard and comparable performance results across *PathArmor*'s configurations, we also measured the time to complete all the C programs in the SPEC CPU2006 benchmarks and compared against the baseline. Figure 4.4 presents our findings. Our results confirm the general behavior observed for our server applications, but the performance overhead is generally much lower (3% in *PathArmor*'s default configuration, geometric mean). This result stems from the lower number of

	Normalized Runtime				Event Stats		
Server	*LBR only*	*+LInstr*	*+CBInstr*	*+PathVer*	*#LCalls*	*#SCalls*	*#Signals*
`vsftpd`	1.000	1.000	1.000	1.000	35,883	42,446	208
`proftpd`	1.000	1.000	1.000	1.000	171,440	48,562	6
`pure-ftpd`	1.003	1.053	1.031	1.074	115,897	57,843	64
`lighttpd`	1.097	1.236	1.226	1.275	1,209,081	200,564	0
`nginx`	1.053	1.178	1.168	1.174	1,500,021	200,002	0
`opensshd`	1.003	1.003	1.031	1.020	24,313	720	8
`exim`	1.025	1.019	1.036	1.079	67,849	4,149	50
geomean	1.025	1.066	1.067	1.085	154,831	28,229	12

Table 4.5: Runtime normalized against the baseline and statistics gathered during the execution of our benchmarks.

library and system calls issued by SPEC programs, as expected for standard CPU-intensive (as opposed to syscall-intensive) benchmarks.

Overall, *PathArmor* imposes a relatively low runtime performance impact on all the test programs considered. This confirms that *PathArmor*'s lightweight instrumentation and cache-aware path analysis are successful in producing a runtime overhead comparable to the most efficient (source-level and forward-edge only) $\overline{\text{C}}$CFI techniques [177], while enforcing much more advanced context-sensitive CFI policies on both the forward and backward edge and operating entirely at the binary level.

4.5.5 LBR Pollution

As discussed in Section 4.3, *PathArmor* supports two modes of operation: (1) stop tracking branches at the library boundary, or (2) continue tracking within libraries. The current implementation of *PathArmor* uses the first mode by default, effectively increasing the control flow context of the protected binary during path verification. To also protect against control flow diversion triggered within library code, *PathArmor* can be configured with the second mode of operation. When running in this mode, branch tracking is never disabled at the cost of (partially) "polluting" the LBR from (self-contained) library code.

To evaluate the LBR pollution cost of running in full-library mode, we configure *PathArmor* to compare LBR contents right before and right after each library call and rerun the SPEC CPU2006 benchmarks. Table 4.6 shows the results. The average pollution rate of 25.68% overall (geometric mean) is likely acceptable in environments where untrusted, potentially vulnerable libraries are in place.

Tracking inside libraries leads to better performance, as this removes the jump to kernel during application-library transitions. Thus, as mentioned earlier, the results provided in our evaluation show worst-case performance. As discussed above, the tradeoff of in-library tracking is increased LBR pollution. However, this can be mitigated with complementary techniques, such as inlining library code or using hardware that provides a larger branch record.

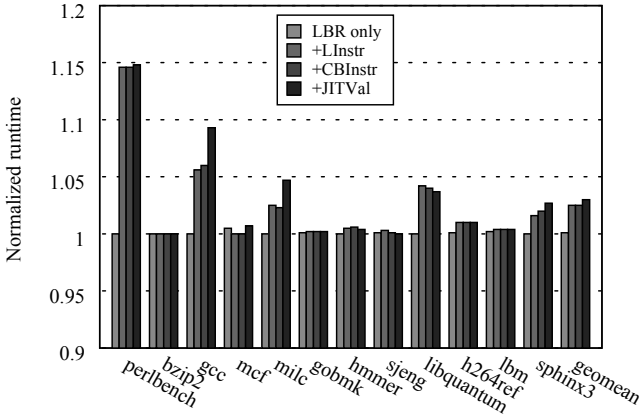


Figure 4.4: Runtime normalized against the baseline for SPEC CPU2006.

4.6 Discussion

This chapter has described context-sensitive CFI, and evaluated the design decisions made in *PathArmor*, our CCFI implementation. We now discuss evasion techniques an attacker may employ to bypass *PathArmor*, analyzing their impact, potential mitigations, and the limitations of our current solution.

4.6.1 History Flushing Attacks

An attacker may attempt to mount a *history flushing attack* to clear any traces of a ROP chain from the LBR. History-flushing attacks previously described in the literature first execute 16 innocuous *NOP-like gadgets* followed by a long *termination gadget* that restores argument registers and ultimately performs a security-sensitive system call [49]. The long termination gadget bypasses heuristics used in prior LBR-based solutions such as kBouncer [136] and ROPecker [56], which rely on weak security invariants based on gadget size (which they assume to be small) and gadget invocation frequency.

PathArmor is not vulnerable to this simple attack, as history flushing in *PathArmor* would require an attacker to craft a valid CCFI-permitted path of 16 NOP-like gadgets (using direct calls or indirect branches). This is much more difficult than chaining arbitrary and CFG-agnostic gadgets. In other words, the notion of a path in *PathArmor* is stronger than that of regular (context-insensitive) CFI and much stronger than that of kBouncer and ROPecker. Hence, while history flushing attacks generally remain of concern, *PathArmor*'s stronger invariants significantly raise the bar for the attacker.

For example, we have shown in Section 4.5.1 that it is generally much harder to maintain register states over the many branches required in history flushing attacks. This makes it very difficult for an attacker to setup arguments for calling a sensi-

	#libcalls	#polluted	%polluted
gcc	3,373,862	13,086,146	24.24
bzip2	449	1,284	17.87
perlbench	60,495,412	253,246,721	26.16
mcf	470,597	5,705,524	75.78
milc	28,807,387	65,657,612	14.24
gobmk	299,877	1,004,581	20.94
hmmer	4,098,071	18,395,790	28.06
sjeng	11,602	176,683	95.18
libquantum	52,609,059	105,222,996	12.50
h264ref	2,449,569	12,515,117	31.93
lbm	2,626,460	5,263,308	12.52
sphinx3	48,625,654	187,711,907	24.13
geomean	1,604,689	6,595,149	25.68

Table 4.6: LBR pollution caused by library calls for SPEC CPU2006. #libcalls: overall library calls, #polluted: overall polluted LBR entries, %polluted: percentage of LBR entries polluted (average).

tive function, and then maintain these arguments throughout the process of flushing history preceding the eventual sensitive call.

A related potential attack is to force context switches in order to clear the LBR and thereby indirectly mount a history flushing attack. Like other history flushing attacks, this attack strategy is also ineffective against *PathArmor*. This is because, as outlined in Section 4.4, *PathArmor* stores and restores LBR states during context switches on a per-thread basis.

4.6.2 Non-control Data Attacks

An attacker may attempt to mount a non-control data attack to indirectly influence the execution of existing security-sensitive functions in the program without directly diverting control flow. For example, an attacker can exploit an arbitrary memory write vulnerability to overwrite sensitive function arguments that are maintained in a data region.

Similarly to all the existing (and even ideal) CFI solutions, *PathArmor* cannot protect against these and other data-only attacks. Unlike existing whole-program CFI solutions, however, *PathArmor*'s history-based strategy would also allow an attacker to craft a ROP-based memory write primitive before jumping to the beginning of a valid execution path leading to a security-sensitive function. Nevertheless, since ROP is not necessary to perform an attacker-controlled memory write and arbitrary memory write vulnerabilities are actually very common, we do not believe this is a limiting factor within our threat model. We also note that binary-level defenses against non-control data attacks are explored in orthogonal work [168].

4.6.3 Endpoint-pruning Attacks

An attacker may attempt to evade detection by avoiding calls to sensitive endpoints recognized by *PathArmor*. This is because, similarly to prior endpoint-driven solu-

tions [56; 136], *PathArmor* enforces security invariants only at predetermined sensitive function calls. Assuming *PathArmor*'s default configuration, such *endpoint-pruning* attacks require the attacker to find alternative means to affect the system environment without relying on system calls such as `exec`, and `mprotect`.

While this is generally of concern depending on the goals of the attacker, *PathArmor* allows users to configure the list of sensitive endpoints according to their needs. For programs in which our default configuration is not sufficient to provide the required guarantees, users can custom-tune the list of endpoints and balance security and runtime performance.

Nevertheless, we believe that *PathArmor*'s default configuration alone drastically reduces the freedom of an attacker. Although ROP may still be used to perform arbitrary Turing-complete computations within an exploited application's own state space, without the ability to execute core security-sensitive system calls the impact on the system remains limited.

4.6.4 Instrumentation-tampering Attacks

An attacker may attempt to abuse the instrumentation employed in *PathArmor*'s default mode of operation (which disables branch tracking in library code) to alter the branch record. However, such attacks fail to circumvent *PathArmor*'s detection strategy. Consider the scenario wherein an attacker sets up a ROP chain that invokes the `ioctl` system call with a dedicated *PathArmor*-specific argument to tamper with the branch-tracking instrumentation. Depending on the request type, this attack will result in two possible outcomes.

In the case of a `CALLBACK_EXIT` request, *PathArmor*'s kernel module will immediately verify the current LBR state (see Section 4.3.3.3) and detect CCFI invariant violations caused by the originating ROP-based control flow.

In the case of a `LIB_ENTER` request, in turn, *PathArmor*'s kernel module will immediately return control to userland after disabling branch tracking, allowing the attack to resume in LBR-free execution. However, as soon as the attacker invokes a security-sensitive function, *PathArmor*'s kernel module will perform verification as normal. At that point, the LBR state will still reflect the branch record generated by the attacker's original ROP chain (leading to the previously issued `ioctl` system call), resulting, again, in *PathArmor* detecting the attack.

Note that an attacker can also attempt to later re-enable branch tracking via a `LIB_EXIT` operation, but a *PathArmor*-legal path of 16 indirect branches is then required to clear any traces of the original ROP attack. This is essentially equivalent to the history flushing attack discussed earlier.

4.7 Related Work

CFI was originally proposed by Abadi et al. [15]. The original (strict) CFI proposal incurs high overheads. This has lead to a myriad of proposals for practical CFI im-

plementations which realize better performance by strategically trading off security guarantees. There are two broad branches of CFI implementations: (1) Control-Flow Graph-based (CFG-based) CFI, and (2) Heuristic-based CFI.

CFG-based CFI focuses on enforcing properties of the CFG. Compiler-based approaches inherently require source to resolve (indirect) control transfers that are considered legitimate [15; 18; 35; 67; 85; 131; 187; 195]. Due to the availability of source information, these approaches are usually able to derive accurate CFGs. Binary-based approaches, while potentially less accurate (for instance, based on an overapproximated CFG), have the advantage of being applicable to legacy programs where the source code is not available [107; 190; 197; 198; 199]. Recently, modular CFI approaches have also been proposed. These are a variant of CFG-based approaches, which resolve part of the CFG at runtime, providing greater flexibility for dynamically computed targets (such as JIT-compiled code) [132; 133; 138].

In contrast to CFG-based CFI, heuristic-based CFI does not require a CFG to enforce integrity, and is typically agnostic of the specific protected binary. Such approaches include kBouncer [136] and ROPecker [56], which seek to detect anomalous control patterns at sensitive program points. Such approaches are easy to deploy, but are also relatively easy to circumvent, using attack patterns not captured in their heuristics (such as long NOP-like gadgets) [97].

Prior work explored devastating attacks against both CFG-based and heuristic-based CFI, using malicious combinations of individually legal control transfers [49; 76; 97]. *PathArmor* enables stronger defenses against such attacks by efficiently enabling context-sensitive CFI policies over paths to sensitive functions and disallowing many unnecessary forward and backward edges permitted by prior context-insensitive CFI policies (such as backward edges to arbitrary call-site gadgets [97]).

In prior fine-grained CFI techniques, context-sensitive policies have been explored only for backward edges and only using shadow stacks [33; 54; 58; 61; 74; 85; 145; 152; 166; 194]. As we have seen in StackArmor, discussed in Chapter 3, such approaches rely on strong memory randomization for their security guarantees. More importantly, they are specifically designed to prevent backward edge attacks, and cannot be extended to protect the forward edge.

In contrast to the runtime shadow stack approach, *PathArmor* resolves backward edges using a hardware-supported context-sensitive static analysis over the interprocedural CFG and caches the results at sensitive points in the program, yielding improved performance and security against tampering attacks. Static context-sensitive backward edge resolution strategies have been explored before for security, but only to improve the accuracy of IDS models based on syscall sequences [184]. In contrast, *PathArmor* shows that enforcing context-sensitive CFG-based policies both on the forward and backward edge at a much finer level of granularity (i.e., control-flow transfers for CFI) is a realistic and efficient option thanks to emerging hardware features. This result contrasts claims in prior work, which, while acknowledging their security advantages, generally dismissed context-sensitive CFI policies as impractical for real-world adoption [15].

Some other approaches have used hardware-supported branch tracing to improve CFI performance. Similar to *PathArmor*, kBouncer [136] and ROPecker [56] rely on Intel's LBR to efficiently implement branch tracing, but only to enforce heuristic CFI policies (based on gadget length and gadget invocation patterns) which can be easily circumvented [49]. CFIMon [190] can enforce hardware-supported CFG-based CFI policies, but relies on the significantly slower Intel BTS [136] and yields high detection latencies, potentially missing attacks [56]. Unlike *PathArmor*, none of these approaches attempt to enforce context-sensitive policies over hardware-monitored control transfers.

Recent work on Control-Flow Bending (CFB) evaluates the general effectiveness of even ideal (context-insensitive) CFI solutions and evidences their limitations against sophisticated CFG-aware attacks [48]. Compared to regular CFI, CCFI makes such attacks harder, given that entire paths (rather than individual CFG edges) are checked for validity. CFB attacks have already been shown to be more difficult against CFI solutions that are complemented by a shadow stack [48]. Compared to such solutions, CCFI does not rely on in-process runtime information and can enforce context-sensitive invariants on both the forward and backward edges, thereby providing improved defenses against CFB attacks.

Concurrently with our work, Schuster et. al. developed the COOP attack [157], showing that CFI solutions which do not precisely consider object-oriented semantics in C++ programs can generally be bypassed. While *PathArmor* as described in this chapter mainly focuses on C rather than C++ programs, follow-up work (implemented as part of a forward-edge CFI system called *TypeArmor*) shows that CCFI can also strengthen forward-edge invariants for use in binary-level protection against COOP-like attacks [181].

4.8 Conclusion

Since the original CFI paper by Abadi et al. [15], it has been known that Context-sensitive CFI (CCFI) can significantly enhance the security of defenses against state-of-the-art control-flow diversion attacks. Despite this knowledge, CCFI has thus far received little attention in the literature because it was perceived as inefficient and impractical for real-world adoption due to prohibitive overhead. This chapter has shown that the three fundamental challenges towards fast and practical CCFI—efficient path monitoring, analysis, and verification—can indeed be effectively addressed in a realistic way on modern commodity platforms.

To substantiate our claims, we implemented *PathArmor*, the first binary-level CCFI solution that efficiently enforces context-sensitive CFI policies on both the backward and forward edges. *PathArmor* addresses all the fundamental CCFI challenges using low-overhead hardware registers to track control flow edges, a scalable on-demand and constraint-driven context-sensitive static analysis, and a path cache verified at sensitive program points.

PathArmor yields comparable or better performance than prior context-insensitive CFI solutions, while enforcing much stronger context-sensitive security invariants. Moreover, *PathArmor* provides a general framework which can be used as a basis for implementing arbitrarily sophisticated CCFI policies.

Chapter 5

Parallax: Implicit Binary Code Integrity Verification Using Return-Oriented Programming

Parallax is a novel self-contained code integrity verification approach, that protects instructions by overlapping Return-Oriented Programming (ROP) gadgets with them. Our technique implicitly verifies integrity by translating selected code (*verification code*) into ROP code which uses gadgets scattered over the binary. Tampering with the protected instructions destroys the gadgets they contain, so that the verification code fails, thereby preventing the adversary from using the modified binary. Unlike prior solutions, *Parallax* does not rely on code checksumming, so it is not vulnerable to instruction cache modification attacks which affect checksumming techniques. Further, unlike previous algorithms which withstand such attacks, *Parallax* does not compute hashes of the execution state, and can thus protect code with non-deterministic state. *Parallax* limits performance overhead to the verification code, while the protected code executes at its normal speed. This allows us to protect performance-critical code, and confine the slowdown to other code regions. Our experiments show that *Parallax* can protect up to 90% of code bytes, including most control flow instructions, with a performance overhead of under 4%.

5.1 Introduction

Code integrity verification (*tamperproofing*) aims to ensure that code executes as intended on a *hostile host* [189], without modification by an adversary. *Self-verifying code* implements such integrity checks without requiring specialized hardware (such as trusted execution modules) or remote verification servers.

Code protection primitives like integrity verification are widely used in practice to delay reverse engineering attacks, and to deter non-persistent adversaries. Code

protection is commonly used by malware to prolong its lifespan and monetization period [38; 153; 154], but it is also used to protect benign programs against reverse engineering [34]. Furthermore, code integrity verification in particular can also defend against certain parasitic malware techniques, which inject inline hooks or code into processes [192] or executables [174].

Recently, the American Institute of Aeronautics and Astronautics launched a code protection initiative to prevent attacks against aviation control systems [11]. The United States Department of Defense has also expressed interest in code protection for use in hardened computing centers, as well as real-time software [180].

Code integrity verification also finds applications in Internet-of-Things (IoT) devices. These are inexpensive devices deployed in the physical world on a massive scale, making them very sensitive to tampering by adversarial users [28; 175]. As described below, our proposed technique is highly suitable for protecting IoT devices, due to its ability to flexibly tune performance overhead so that it does not affect the critical execution path.

Most existing code self-verification algorithms work by computing checksums over protected code regions, and verifying that these checksums are as expected. Using several cross-verifying checksummed code regions, such algorithms can provide fairly strong tamperproofing. Unfortunately, Wurster et al. have shown that all such algorithms are inherently vulnerable to automated attacks which exploit the distinct handling of code and data in modern processors [189]. Wurster et al. implement a kernel patch which allows attackers to freely tamper with the code in the processor's instruction cache, while leaving the data cache entirely untouched. This completely circumvents checksumming algorithms, as these treat code as data, thus fetching it from the data cache instead of the instruction cache.

The foremost code verification technique designed to defeat this attack is *oblivious hashing* (OH) [55; 102]. Instead of directly checking code integrity, oblivious hashing intersperses hashing instructions with the protected code, which build runtime hashes of the execution state. The integrity is then verified by checking that the computed hashes correspond to known correct values. However, this technique can only verify deterministic execution state, of which the expected hash is known at compile time. This means that OH cannot protect code which involves non-deterministic inputs, such as environment parameters or user input. Additionally, the hashes used to verify the state are found using dynamic testing, limiting the protection to code paths exercised in these tests.

We propose a novel code self-verification approach, which is based on *Return-Oriented Programming* (ROP). ROP was originally proposed as an exploitation technique which allows arbitrary code execution in the presence of W$\oplus$X [163]. ROP uses short return-terminated instruction sequences, called *gadgets*, which are chained together by arranging their addresses on the stack such that each terminating return causes a control transfer to the next gadget. If a sufficient set of gadgets is available, ROP is a Turing-complete programming technique which can implement arbitrary computations on top of a host program. A Turing-complete gadget set exists in most

programs [159]. We refer to an arrangement of gadget addresses into a ROP program as a *ROP chain*.

Our code verification approach protects code by overlapping ROP gadgets with it. Then, selected instructions from the protected program are translated into ROP chains which use the overlapping gadgets. Since tampering with the gadgets causes the translated instructions to malfunction, this implicitly verifies the integrity of the protected code. Thus, we refer to these translated instructions as *verification code*. Our notion of verification code can be seen as a generalization of code hiding techniques based on function reuse [117]. We show in Section 5.6 that the verification code is itself also tamper resistant.

Since the verification code uses ROP, it requires a set of gadgets overlapping with the instructions we protect. We both use gadgets already present in the host binary, and statically rewrite the binary to craft new ones. Since a Turing-complete gadget set is already present in most programs [159], the additional tamperproofing gadgets generally do not increase the vulnerability of protected programs to ROP attacks.

We implemented a prototype implementation of our technique for the x86 platform, called *Parallax*. It uses binary rewriting to create protective gadgets, and builds on ROP compiler functionality to generate verification code. Our proof of concept provides the ability to use source to simplify binary rewriting, and also offers the option of selecting verification code at the source level. However, this is not a requirement of our technique, which can be implemented entirely at the binary level.

Contributions Our technique has several advantages over prior work.

- We do not use checksumming, thus preventing the attack of Wurster et al.
- In contrast to oblivious hashing, no prior knowledge of the runtime state is required. Therefore, our technique can protect non-deterministic code regions. Furthermore, we apply this protection statically, so it is oblivious to dynamic code coverage.
- The overlapping gadgets do not slow down code they protect. Instead, performance overhead is confined to the verification code using the gadgets. This makes it possible to tamperproof performance-critical code while confining the performance degradation elsewhere. In contrast, oblivious hashing slows down protected code by interspersing hashing instructions with it.
- We show in Section 5.7 that our technique can protect up to 90% of code bytes at a performance overhead of less than 4%. As argued in Section 5.8, non-deterministic control flow decisions are among the most likely attack targets. Thus, we protect crucial instructions which OH cannot [55].
- In contrast to prior work, including oblivious hashing, our approach lends itself to binary-level implementation, and does not rely on source. This enables the protection of legacy binaries.

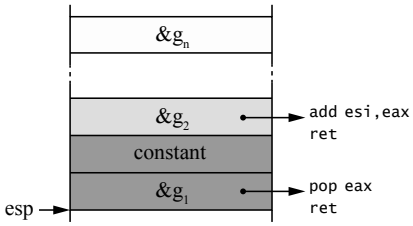


Figure 5.1: An example ROP chain. Gadget g_1 loads a constant into `eax`, which is then added to `esi` by g_2.

5.2 Background

This section describes Return-Oriented Programming, upon which we base our technique. We also describe the threat model which *Parallax* assumes.

5.2.1 Return-Oriented Programming

ROP was originally proposed in 2007 as an exploitation technique designed to circumvent memory protection mechanisms like W⊕X [163]. ROP traditionally makes use of short instruction sequences found in a host program's memory space, called *gadgets*, each of which ends in a return instruction (more modern incarnations like Jump-Oriented Programming also make use of gadgets ending in forward edge instructions [52]). Each gadget typically performs a basic operation, such as addition or logical comparison. Gadgets can be part of the host program's normal instructions, but can also be unaligned instruction sequences embedded within the normal instruction stream. A ROP program consists of a chain of gadget addresses on the stack, such that the return instruction terminating each gadget transfers control to the next gadget in the chain.

Figure 5.1 illustrates an example ROP chain. Initially, the stack pointer (`esp`) points to the address of the first gadget g_1 in the chain. Upon execution of a return instruction, control is transferred to this gadget. It performs a `pop` instruction, which loads a constant arranged on the stack into the `eax` register, and increments `esp` to point to gadget g_2. Then, the `ret` instruction of gadget g_1 transfers control to gadget g_2, which adds the constant in `eax` to the `esi` register. Gadget g_2 then returns to gadget g_3, and so on, until all gadgets $g_1, \ldots, g_n$ have been executed.

5.2.2 Threat Model

Parallax assumes the *hostile host* threat model [189], which is the standard model for tamperproofing techniques. It assumes that the tamperproofed application is executed on a system controlled by a hostile user, which has full control over the runtime environment, and may arbitrarily modify the tamperproofed executable itself. This

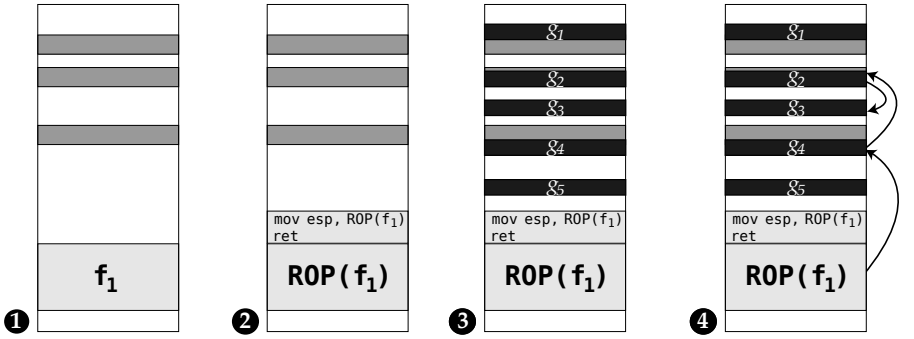


Figure 5.2: A high-level overview of *Parallax*.

includes alterations made during runtime debugging, as well as static code patching. The intent of the hostile user is typically to circumvent access controls in the protected application, such as anti-debugging checks or license verifications. The challenge for our tamperproofing technique is thus to maximize the effort required by the hostile user to successfully tamper with the protected application, without assuming any trusted components in the runtime environment.

5.3 Parallax Overview

This section gives an overview of how *Parallax* implements protection against both static and dynamic code modification. Figure 5.2 illustrates how *Parallax* protects a given binary.

To protect a binary, we select one or more code fragments at the source or binary level for use as verification code (step ① in Figure 5.2). In Section 5.7.2, we describe our strategy to do this automatically. Additionally, we determine a list of instructions to protect. If source is available, these are selected at the statement or function level, and then mapped to the binary level after compilation using debugging symbols. If only a binary is available, protection is assigned at the instruction or function level.

Parallax begins by translating the selected verification code into one or more ROP chains (sequences of ROP gadgets) ②. These chains use placeholder gadget addresses, since the final addresses are not yet known at this point. Eventually, these placeholders will be replaced by gadget addresses in the protected code, so that executing the verification code implicitly verifies that the protected code is still intact. Along with the ROP chains, *Parallax* inserts a loader routine to bootstrap them. Optionally, the binary to protect is compiled from source if available.

Next, *Parallax* creates a collection of all gadgets available in the binary ③. First, any existing gadgets are added to the collection. Then, *Parallax* walks through the list of instructions which were selected for protection. For every such instruction,

Parallax examines if it can be augmented with an overlapping gadget. If so, it inserts a gadget using binary rewriting, and adds this new gadget to the collection. The gadgets are denoted as $g_1, \ldots, g_5$ in Figure 5.2. We discuss our strategy for crafting overlapping gadgets in Section 5.4.

Note that we do not require the inserted overlapping gadgets to form a Turing-complete set, since most binaries already contain a Turing-complete gadget set by default [159]. If not, a standard set of non-overlapping gadgets can be inserted into the binary to augment the protective gadgets already inserted.

Finally, *Parallax* creates a *gadget mapping* which categorizes the available gadgets in the binary into a set of types; for instance, memory stores and register moves. The gadget mapping is then used to recompile the verification code (using a ROP compiler) such that it uses actual gadgets instead of placeholder addresses ④. During compilation of the verification code, overlapping gadgets are always preferred over non-overlapping gadgets. Tampering with the protected instructions modifies the code bytes of the overlapping gadgets, thereby invalidating them. Such changes are implicitly detected by the verification code, which malfunctions if the integrity of the gadgets it uses is violated. We discuss the tampering and analysis resistance of verification code in more detail in Sections 5.5 and 5.6.

5.4 Protecting Code Integrity

This section discusses the creation of ROP gadgets which overlap with existing code, and protect the code integrity. As discussed in Section 5.3, these gadgets need not form a Turing-complete set. Instead, the focus is on gadgets which have maximal overlap with the protected instructions. The creation of verification code which uses the gadgets is discussed in Section 5.5. We provide an example of gadget insertion in Section 5.4.1, and generalize it in Section 5.4.2 by describing the rules which *Parallax* uses to craft overlapping gadgets.

5.4.1 A Tamperproofed `Ptrace` Detector

We provide a running example of a `ptrace` detection function augmented with overlapping gadgets. We compiled this function with `gcc` 4.6.3, and then used *Parallax* to search for locations where overlapping gadgets could be inserted. In the example, we manually chose which instructions to protect from the list of possible locations. To avoid manual effort, it is also possible to input a list of functions to protect, and rely on *Parallax* to overlap gadgets with as many instructions in these functions as possible. Alternatively, if source is available, source-level statements can be marked for protection, and mapped to instructions using debugging symbols. The rules *Parallax* uses to create gadgets are discussed in Section 5.4.2.

Listing 5.1 shows a disassembly dump of our tamperproofed `ptrace` detector. For clarity, we have shortened addresses such as `08048438` to `n+38`. We first describe the purpose of the `ptrace` detector, and then elaborate on how it is protected.

```
n+38 <cleanup_and_exit>:
n+38: 55                        push ebp
n+39: 89 e5                     mov  ebp,esp
n+3b: 83 ec 18                  sub  esp,24
n+3e: 89 04 24                  mov  [esp],eax
n+41: e8 d5 fe ff ff            call exit@plt

n+46 <check_ptrace>:
n+46: 55                        push ebp
n+47: 89 e5                     mov  ebp,esp
n+49: 83 ec 18                  sub  esp,24
n+4c: c7 44 24 0c 00 00 00 00   mov  [esp+0xc],0
n+54: c7 44 24 08 00 00 00 00   mov  [esp+0x8],0
n+5c: c7 44 24 04 00 00 00 00   mov  [esp+0x4],0
n+64: c7 04 24 00 00 00 00      mov  [esp],0
n+6b: e8 cb fe ff ff            call ptrace@plt
n+70: 85 c0                     test eax,eax
n+72: 79 07                     jns  n+7b
n+74: b8 01 00 00 00            mov  eax,1
n+79: eb bd                     jmp  n+38
n+7b: b8 00 00 00 00            mov  eax,0
n+80: c9                        leave
n+81: c3                        ret
```

(a) Original code.

```
n+32 <cleanup_and_exit>:
n+32: 55                        push ebp          ← relocated
n+33: 89 e5                     mov  ebp,esp
n+35: 83 ec 18                  sub  esp,24
n+38: 89 04 24                  mov  [esp],eax
n+3b: e8 d5 fe ff ff            call exit@plt

n+46 <check_ptrace>:
n+46: 55                        push ebp
n+47: 89 e5                     mov  ebp,esp
n+49: 83 ec 18                  sub  esp,24
n+4c: c7 44 24 0c 00 00 00 00   mov  [esp+0xc],0
n+54: c7 44 24 08 00 00 00 00   mov  [esp+0x8],0
n+5c: c7 44 24 04 00 00 00 00   mov  [esp+0x4],0
n+64: c7 04 24 00 00 00 00      mov  [esp],0
n+6b: e8 cb fe ff ff            call ptrace@plt   ← existing far return
n+70: 85 c0                     test eax,eax
n+72: 79 07                     jns  n+7b
n+74: b8 c3 00 00 00            mov  eax,0xc3     ← modified exit argument
n+79: eb c3                     jmp  n+32         ← modified target
n+7b: b8 00 00 00 00            mov  eax,0
n+80: c9                        leave
n+81: c3                        ret
```

(b) Protected code.

Listing 5.1: A `ptrace` detector with gadgets (shaded) overlapping sensitive areas.

```
(gdb) set *(unsigned char*)0x08048479=0x90
(gdb) set *(unsigned char*)0x0804847a=0x90
```

Listing 5.2: An attempt to disable the `ptrace` detector.

The `ptrace` detector checks if a process is being debugged using `ptrace`. To achieve this, the detector calls the `ptrace` system call, requesting a trace of the host process. If a debugger is already attached, this call fails, and the debugger is thus detected. In the example, the detector jumps to a `cleanup_and_exit` function if a debugger is detected.

Attackers commonly attempt to circumvent such anti-debugging code by modifying it at runtime, as shown in Listing 5.2. In the listing, an adversary overwrites the jump to the `cleanup_and_exit` function at address `n+79` with `nop` instructions. The goal of this attack is to redirect control to a successful return even though a debugger is attached.

Overlapping gadgets defend against this attack class, as they are destroyed if the code they overlap with is modified. As mentioned in Section 5.3, this is detected when the verification code using the gadgets fails to execute. Note that an adversary could also modify the call to `check_ptrace` itself. As we show in Section 5.7.1, *Parallax* can protect up to 90% of the binary, allowing us to defend against such attacks by inserting protective gadgets beyond the primary list of instructions to protect. While this example focuses on runtime code modification, *Parallax* also prevents static code patching.

In Listing 5.1, four key code areas which adversaries are likely to target have been protected using three overlapping gadgets. The first two locations are (1) the call to `ptrace` itself, at address `n+6b`, and (2) the first argument to `ptrace`, at address `n+64`, which requests a trace of the host process. An adversary may eliminate the call, so that execution always falls through to the successful return code at the end of the function. Also, an adversary may modify the call argument to request another action from `ptrace` instead of a trace of the host process. Both the call and its first argument are protected by a seven byte long overlapping gadget starting at address `n+66`. This is an already existing gadget, which *Parallax* found without making any code modifications. The gadget consists of the instructions `and al,0; add [eax],al; add al,ch; retf`, and can be used to move the contents of the `ch` register into the `al` register (the memory write can be ignored, since `al` is zeroed out).

Note that it is also possible to protect the remaining `ptrace` arguments at addresses `n+4c` through `n+5c`. One possible way to protect these is to use the immediate splitting rule, discussed in Section 5.4.2. For simplicity of the example, we do not show these modifications in Listing 5.1. However, we provide a separate example of the immediate splitting rule in Section 5.4.2.

Another possible attack location is (3) the jump to `cleanup_and_exit`, at address `n+79`, which is taken if a debugger is detected. Eliminating this jump would

again cause control to fall through to the successful return at the end of the function, even if the call to `ptrace` failed. *Parallax* protects this jump by relocating the `cleanup_and_exit` function, and modifying the jump offset to encode the `ret` instruction for a gadget. The gadget starts at address `n+78`, and contains instructions `add bl,ch; ret`.

Finally, the anti-debugging code could be disabled by (4) rewriting the `jns` instruction at address `n+72` to an unconditional `jmp` instruction, so that the code always jumps to a successful return. *Parallax* identifies two possible ways to protect against this. The first is to modify the immediate operand of the `mov` instruction at address `n+74`, such that its least significant byte encodes a `ret` instruction. This creates a five byte long gadget at address `n+71`, consisting of the instructions `sar byte [ecx+0x7],0xb8; ret`. This gadget fills the memory byte at address `[ecx+0x7]` with the sign of the byte it contains (the bits are either all set to 0, or all set to 1). The `mov` operand is an exit status, and can be safely modified assuming that the exit semantics differentiate only between zero and non-zero (see Section 5.4.2).

An alternative way to protect the `jns` is to inject a spurious instruction directly after it, which encodes the missing part of a partial gadget. In the example, we did not use spurious instructions, in order to show that no added code is needed to protect the function.

5.4.2 Binary Rewriting Rules

This section describes the binary rewriting rules *Parallax* uses to augment instructions with overlapping gadgets. The added gadgets do not induce any significant performance overhead on the protected code, except where explicitly noted. In Section 5.7.1, we measure the coverage of each of these rules. We base our approach on binary rewriting techniques for legacy binaries explored in prior work [114; 199].

Existing gadgets *Parallax* searches for any existing gadgets which can be used to protect code integrity. The use of existing gadgets is advantageous, as it requires no modifications to the protected code regions. In Section 5.7.1, we find that 3%–6% of the code bytes in our test cases is protectable using existing gadgets.

Modified immediate operands One rule used by *Parallax* to create new gadgets is that a partial gadget may be combined with an adjacent immediate operand if this operand can be modified to encode the missing portion of the desired gadget. In Listing 5.1, this rule has been applied in the operand of the instruction at address `n+74`. We distinguish two ways in which immediate operands can be safely modified.

First, depending on the instruction type, immediates can be modified by splitting up their parent instructions. For instance, an addition can be split into two additions or subtractions, where the first takes an arbitrary operand, and the second compensates as required. Similarly, immediate operands of `mov` instructions can

```
mov eax,1          b8 01 01 c3 00  mov eax,0xc30101
                   35 00 01 c3 00  xor eax,0xc30100
```

(a) Original code. **(b)** Protected code.

Listing 5.3: A split `mov` with overlapping gadgets (shaded).

be modified to encode a gadget, and this modification can then be compensated for using bitwise operations on the destination operand. As an example of this rule, Listing 5.3 shows how the immediate operand of a `mov` instruction is modified and combined with an `xor` instruction to compensate for the modifications.

Instruction splitting induces a small performance overhead on the protected code. Additionally, it may require the insertion of code to save and restore the CPU status register, depending on the usage of status flags by subsequent instructions.

Second, it is often (though not always) possible to freely modify immediates which set `eax` before a return, or push the status of the `exit` function. This is because return value and exit status semantics commonly distinguish only between zero and non-zero. This rule requires limited annotation or analysis of the relevant semantics, and can be disabled for conflicting semantics.

Rearranged code and data *Parallax* also attempts to encode missing parts of gadgets in addresses and jump offsets by strategically aligning functions and global variables in memory (e.g., by inserting padding bytes such as `nop` instructions). For instance, in the example shown in Listing 5.1, we have forced the creation of a `ret` instruction by aligning the `cleanup_and_exit` function such that the jump offset at address `n+79` is equal to `0xc3` (the `ret` opcode).

Spurious instructions Spurious instructions which contain (parts of) gadgets can be inserted at any place in the code, as long as care is taken to ensure that their side-effects do not influence the semantics of the original code. This can be ensured by saving and restoring the program state at each location where spurious instructions are inserted. Alternatively, side-effect analysis can be performed on the inserted instructions to determine whether any state needs to be saved [44].

The main benefit of spurious instructions is that they can always be inserted to encode missing parts of gadgets, even when other strategies do not apply. However, the tradeoff is that spurious instructions induce a slowdown on the protected code similar to the inline hashing instructions used in Oblivious Hashing [55]. Given that we aim to minimize overhead, *Parallax* currently does not implement spurious instruction insertion.

Far-return gadgets Far returns (`retf`) are quite rare in compiler-generated x86 code. Nevertheless, gadgets ending in far returns can sometimes be used to protect code bytes, as was done at address `n+66` in Listing 5.1. *Parallax* searches for existing far-return gadgets in the same way as for near-return gadgets.

Using `add` for memory operations One of the most useful instruction families for gadgets is the `add` family. This is because the opcodes of `add` range from `0x00` to `0x05`. As these values are very common in immediate operands, `add`-based gadgets are easy to find or create. Listing 5.1 contains several gadgets which use `add` instructions, such as the gadget protecting the call to `ptrace`.

Next to implementing additions, `add` instructions with memory operands can also be used as loads and stores. For instance, `add [ecx],eax` implements a store of the value in `eax` to the address in `ecx`, if this memory is initially zero.

5.5 Verifying Code Integrity

In Section 5.4, we discussed the creation of overlapping gadgets for code protection. To protect their parent instructions, the integrity of these gadgets must be verified by one or more ROP chains. In this section, we discuss the translation of existing code from the protected program into ROP chains which act as verification code. These ROP chains use the gadgets contained in the protected code regions.

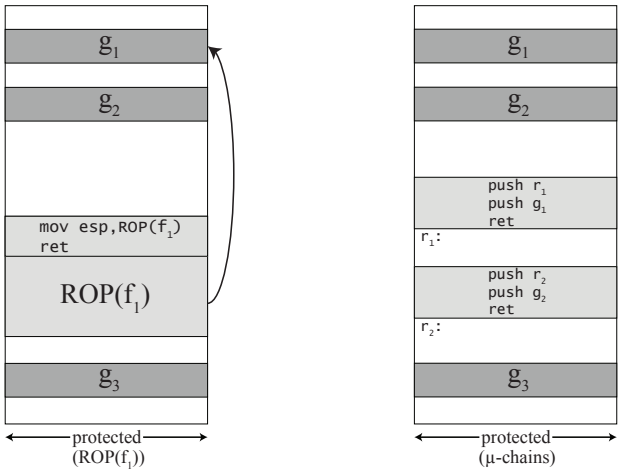
We stress that the verification code does not perform any active verification, checksumming or otherwise. Instead, it detects and responds to tampering in a completely passive way, by malfunctioning if the gadgets in the protected code regions are damaged by tampering attempts.

We implement verification code at function granularity, meaning that whole functions from the original program are translated to ROP code. For brevity, we refer to a function-level verification ROP chain as a *function chain*. In Section 5.5.3, we also briefly report on our experiences with instruction-level verification.

5.5.1 Implementation of Function Chains

Function chains were already briefly discussed in Section 5.3. Figure 5.3a illustrates a binary protected using function chains. As discussed in Section 5.3, the protected binary contains several gadgets, g_1, g_2, g_3, which are crafted such that they overlap with instructions which must be protected. Furthermore, a selected function f_1 from the protected binary's code section is translated into equivalent ROP verification code, denoted as $ROP(f_1)$. Additionally, a small amount of loader code is inserted, which is responsible for starting the execution of the verification code. The minimum operations required for this are (1) pointing the stack pointer to the beginning of $ROP(f_1)$, and (2) executing a return instruction to transfer control to the first gadget in the verification code.

In our *Parallax* prototype, we implemented function-level verification on top of a modified version of the open source ROP compiler ROPC [6], which is based on Q [159]. Our prototype loader code, which bootstraps the execution of the function chains, is more extensive than shown in Figure 5.3a. Particularly, in addition to pointing the stack pointer to the start of a function chain and executing a return, we also ensure that execution continues cleanly after the function chain is complete.



(a) A function chain. **(b)** Scattered μ-chains.

Figure 5.3: Verification at function and instruction level.

To achieve this, the loader code appends an epilogue to each function chain before transferring control to it. The epilogue consists of the address of a `pop esp` gadget, followed by a stack address pointing into the original stack frame of the calling function. At this stack address, we store the return address for the function chain. Before the epilogue's `pop esp` gadget is executed, the stack pointer points inside the function chain. Upon completion of the function chain, the `pop esp` points `esp` back into the calling function frame, to the stack location containing the function chain's return address. Thus, when the function chain returns, this transfers control back to the calling function, and program execution continues normally.

In addition to the epilogue, we perform a `pushad` directly before, and a `popad` directly after each function chain. These instructions save and restore the register state, preventing problems due to registers clobbered by the function chain.

5.5.2 Dynamically Generated Function Chains

Function chains can reside in data memory, which is writable even with W$\oplus$X protection on. This means that it is possible to generate function chains at runtime. *Parallax* implements optional support for this. Dynamic function chain generation has several advantages. (1) It allows for encrypted and self-modifying function chains, which are more resistant to analysis than their non-dynamic counterparts. We evaluate the performance of RC4-encrypted and xor-encrypted function chains in Section 5.7.2. (2) Multiple instances of the same function chain can be generated probabilistically, with each instance using a different set of semantically equivalent gadgets. This allows a small function chain to (probabilistically) verify a large set of gadgets, checking a subset each time it is executed.

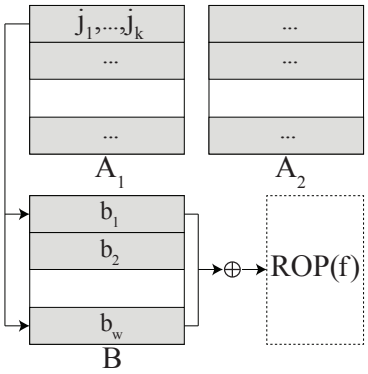


Figure 5.4: Generating a function chain by combining vectors from a basis B, indexed by arrays A_1 and A_2.

Specifically, let $T := \{t_1, \ldots, t_n\}$ be the set of used gadget types in the function chain, and for $1 \leq i \leq n$, define $G_i := \{g \mid g \text{ implements } t_i\}$. Thus, each G_i is the set of all gadgets which implement gadget type t_i. For probabilistically generated function chains, we use an extended notion of the gadget types mentioned in Section 5.3, which defines not only the operation implemented by a gadget, but also its operand registers and memory locations. Then, for every operation, the function chain can probabilistically choose a gadget $g \in G_i$. In total, this yields $\prod_{i=1}^{n} |G_i|$ possible distinct gadget subsets which can be checked by the same function chain. Because the used subset of gadgets is probabilistic, it is hard for an adversary to be sure that his code modifications will work for every execution of the program. This is an especially useful property to protect against scenarios where adversaries aim to widely distribute modified binaries.

Parallax implements probabilistically generated function chains by considering each function chain as a series of vectors $v_1, \ldots, v_n$, where $v_i \in \{0,1\}^w$ for $1 \leq i \leq n$. Here, w is the native memory word length in bits (typically 32 or 64). Intuitively, each vector in a function chain corresponds to a gadget address or constant used by the chain (all constants in our function chains are word-sized). Each vector can be generated using a linear combination of vectors from a basis $B = \{b_1, \ldots, b_w\}$ which spans the vector space $\{0,1\}^w$.

To support dynamic generation of multiple variants of the same function chain, we define a series of N *index arrays* $A_1, \ldots, A_N$, such that each A_i for $1 \leq i \leq N$ is a two-dimensional array of vector indices. The number of index arrays N can be arbitrarily chosen. If the function chain contains l vectors, then each A_i stores l lists of vector indices. If the l-th vector from the function chain is of gadget type t, then the l-th index list in each A_i contains indices $j_1, \ldots, j_k$ which index vectors $b_{j_1}, \ldots, b_{j_k}$ from B such that $b_{j_1} \oplus, \ldots, \oplus b_{j_k} \in \{g \mid g \text{ implements } t\}$. This means we can form N semantic equivalents for each vector in a function chain by choosing randomly between $A_1, \ldots, A_N$ and combining the basis vectors specified there. Fig-

ure 5.4 illustrates this approach to generating function chains. For a function chain of length l, there exist at most N^l variants (assuming that no two A_i store the same index list at any position).

We generate the index arrays by repeatedly compiling the function chain, each time feeding a different gadget mapping to the ROP compiler. By varying the set of gadget addresses used in each mapping, we obtain different compiled variants of the function chain. We then split each vector from every compiled variant into basis vectors, and store the indices of these in the index arrays. Note that the compiled function chains themselves are not stored in the binary, as shown in Figure 5.4. Instead, we use the index arrays to probabilistically generate a function chain variant at runtime, just before the function chain is called. Since we randomly choose between the index arrays at vector granularity, the possible number of function chain variants generated at runtime is greater than the number of compiled variants.

In Section 5.7.2, we discuss our performance experiments on dynamic function chain generation. We report results for function chains encrypted with RC4 and xor, and for probabilistically generated function chains.

5.5.3 Instruction-Level Verification

In addition to function-level verification, we also experimented with instruction-level verification. Instead of translating a whole function, this approach translates many single instructions into short ROP chains, which we refer to as μ-chains. Figure 5.3b compares μ-chains to function chains.

We find μ-chains to be suboptimal for several reasons. (1) To minimize control transfer overhead, μ-chains are best implemented inline in the code section, as shown in Figure 5.3b. This means that, unlike function chains, μ-chains cannot benefit from additional protection by checksumming (due to the attack of Wurster et al. [189]) or self-modification. (2) The inline gadget setup instructions used by μ-chains can be detected through static analysis, and can be exploited by an adversary to pinpoint gadgets used for protection. (3) The overhead of μ-chains exceeds that of function chains by a factor of $2\times$ on average, because each μ-chain contains its own prologue and epilogue. For these reasons, we focus on function-level verification.

5.6 Attack Resistance

This section discusses the resistance of our technique to attacks which attempt to disable, circumvent, or tamper with the verification code. As mentioned, the verification code is a translation to ROP of code from the original program, which is required for the program to correctly execute. The challenge for an adversary is thus to tamper with the protected program in such a way that this is not detected by the verification code, without modifying the verification code functionality. The rest of this section discusses three attack classes.

5.6.1 Code Restoration

An adversary may attempt to evade detection by restoring modified code after it has executed. Such code restore attacks are only relevant in dynamic (runtime) tampering. For static code patching/rewriting scenarios, adversaries cannot use code restore attacks. It is well-recognized in the literature that no self-sufficient tamperproofing algorithm can completely prevent code restore attacks [60]. However, *Parallax* complicates such attacks in several ways. (1) It is critical to use verification functions which are executed repeatedly through the runtime of the protected application (possibly asynchronously, in a separate thread). As we show in Section 5.7.2, *Parallax* achieves this while keeping performance overhead low (up to 4%). (2) By decoupling verification code from protected code, *Parallax* maximizes the difficulty for an adversary to pinpoint which modifications trigger the tamper response.

5.6.2 Verification Code Replacement

Additionally, an adversary may tamper with the code locations where verification code is initialized, and attempt to replace it with another ROP chain, or with non-ROP code. Several factors prevent such attacks. (1) The replacement code must be functionally equivalent to the verification code, while not using the same gadgets. The requirement for functional equivalence imposes a first challenge to the adversary, namely the need to reverse engineer the verification code. This is a time-consuming effort, which is complicated by the lack of analysis tools for ROP code [120]. (2) More fundamentally, *Parallax* increases the reverse engineering effort by using dynamically generated and self-modifying ROP code, as proposed in Section 5.5.2. (3) Because the verification code initialization is deterministic, it could be protected using techniques orthogonal to ours, like oblivious hashing.

5.6.3 Verification Code Modification

Adversaries may also modify the verification code itself. Here, one of the main strengths of *Parallax* becomes apparent: because the verification code resides in data memory, it can be protected by any traditional checksumming technique. At the same time, there is no risk of the attack of Wurster et al. [189], because that attack relies on the handling of code as data. To prevent persistent tampering with the checksumming code, we propose to use a network of cross-verifying checksums, as explored by Chang et al. for code verification [51]. Such a network can be implemented by embedding the checksumming code inside the verification functions, and letting each verification function checksum itself as well as several others. This way, checksumming can also be embedded in dynamically generated verification code (which itself also complicates tampering). As checksumming is not fundamental to our technique, we do not implement it in our proof of concept. We expect that the performance of checksumming will be similar to that of verification code encryption (evaluated in Section 5.7.2).

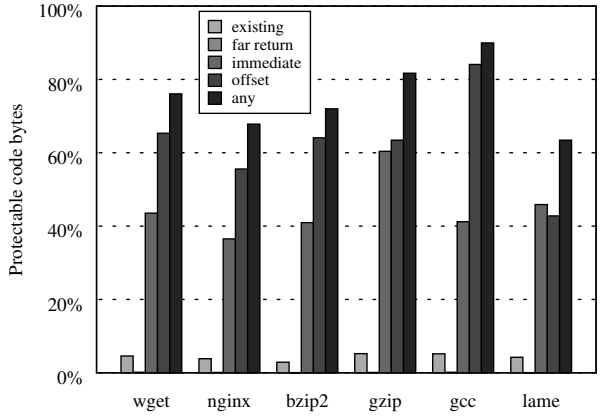


Figure 5.5: Code bytes protectable by rules from Section 5.4.2.

5.7 Evaluation

This section evaluates the performance of *Parallax*, our prototype implementation of ROP-based code integrity verification. In Section 5.7.1, we measure the percentage of code bytes in real-world programs that can be protected using overlapping gadgets. Next, Section 5.7.2 evaluates the runtime overhead induced by the verification code.

5.7.1 Protectable Code Locations

We define a *protectable code byte* as an instruction byte for which we can craft an overlapping gadget using one of the rewriting rules discussed in Section 5.4.2. We used *Parallax* to measure the percentage of protectable code bytes in a set of real-world programs consisting of `wget`, `nginx`, `bzip2`, `gzip`, `gcc`, and `lame`, compiled for x86 using `gcc` 4.6.3.

Figure 5.5 shows the percentage of protectable code bytes using existing near-return gadgets, far-return gadgets, and gadgets created by modifying immediates and jump offsets. Additionally, the figure shows the percentage of code bytes that can be protected using any of these rules. This can be lower than the sum of the per-rule percentages, since code bytes may be protectable using multiple rules.

In our experiments, modifications to immediates were only applied in `add`, `adc`, `sub`, `sbb`, and `mov` instructions. Examples of how we apply such modifications were discussed in Section 5.4.2. Modifications to jump offsets were considered for all variants of the `jmp` and `jcc` instructions, as well as for `call` instructions. No results are shown for the spurious instructions rule, as it is not implemented in *Parallax* and can always be applied. Furthermore, we limited the length of the considered gadgets to six instructions, as longer gadgets are difficult to use in practical ROP chains. Note that it is not necessarily possible to protect all potentially protectable code bytes at once, since the required modifications may conflict.

The lowest protectability rate was 63% (for `lame`), and the highest rate was 90% (for `gcc`). Using any of the rewriting rules, an average of 75% of the code bytes is protectable. As can be seen from Figure 5.5, between 3% and 6% of the code bytes contains an existing overlapping near-return gadget. Additionally, up to 1% of the code bytes in the test programs overlaps with a far-return gadget. The near-return and far-return gadgets add up to protect between 4% and 7% of the code bytes, without requiring any modifications. The protectability rate for the immediate modification rule ranges from 37%–60%, while ranging from 43%–84% for jump modification.

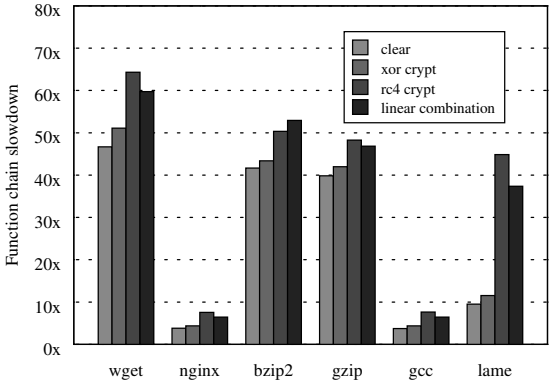
5.7.2 Runtime Overhead

We also evaluated the performance of verification code. To evaluate the performance, we selected one function from each program and measured the performance before and after translating it to ROP code. We use the following (fully automatable) algorithm to select which function to translate in a given program. (1) We first analyze the call graph of the program to find functions which are called repeatedly from several locations. This ensures that the integrity is verified repeatedly. (2) We then profile the program, and select the functions from the previous step which contribute less than a threshold to the total execution time (2% in our experiments). (3) Finally, we select from this the function containing the most types of operations, ensuring good coverage of all gadgets. We considered both application-specific and library functions for translation to function chains.

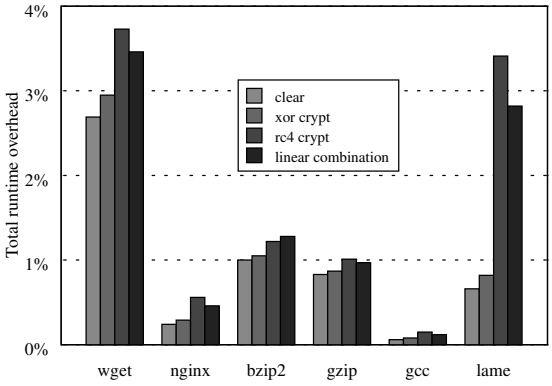
For each selected function, we measured the cleartext slowdown induced purely by the transformation to a function chain. Furthermore, we measured slowdowns for RC4-encrypted and xor-encrypted function chains, as well as function chains generated probabilistically through linear combination (as described in Section 5.5.2). Figures 5.6a and 5.6b show the resulting function chain slowdowns and overall runtime impacts for each of these hardening strategies.

The cleartext function chain slowdown ranges from 3.7× for `gcc` to 46.7× for `wget`. RC4-encrypted function chains have the poorest performance, followed by probabilistically generated and xor-encrypted function chains. The slowdown of RC4-encrypted function chains ranges from 7.6× for `nginx` to 64.3× for `wget`, but the greatest performance decrease compared to other methods is seen in `lame`. This is because the function chain for this test case executes in only 4μs, so that the RC4 initialization phase causes a large slowdown.

Despite the significant slowdown induced on each translated function, the whole-program overheads are limited, ranging from 0.1% for `gcc` to 2.7% for `wget` using cleartext function chains. When using RC4 encryption, the overhead ranges from 0.2% for `gcc` to 3.7% for `wget`. In our experiments, the decryption step (xor, RC4, or linear combination) was performed on each function chain call. Summarizing, the overall runtime overhead of protected binaries is limited, provided that care is taken not to use performance-critical functions as verification code. There is thus a delicate tradeoff between runtime overhead and verification frequency.



(a) Function chain slowdowns.



(b) Whole-program overhead for function chains.

Figure 5.6: Slowdowns and whole-program overheads for function chains.

5.8 Discussion and Limitations

This section discusses the tradeoffs and limitations of *Parallax*. We also compare these tradeoffs to those of other tamperproofing techniques.

5.8.1 Dynamic Circumvention

The goal of our work is to protect code against explicit modifications. Some dynamic analysis primitives, such as software breakpoints and dynamic code patching, are also detected by *Parallax* (see Section 5.4.1). However, *Parallax* does not explicitly defend against dynamic analysis. Specifically, some dynamic analysis tools, such as Pin [101; 121] and DynamoRIO [42], instrument binaries without altering their runtime code section, and are thus not detected by *Parallax*. However, *Parallax* can protect specialized detection code for these tools, developed in related work [89].

5.8.2 Control-Flow Integrity

Prior work has explored the detection of ROP-based exploit code at runtime, using heuristic-based system-level monitoring tools like kBouncer and ROPecker [56; 136]. These tools may conflict with our tamperproofing algorithm, detecting its use of ROP code as if it were malicious. However, recent work has shown that heuristic-based monitoring approaches can be fundamentally circumvented by simple modifications to ROP chains [76; 97; 158]. *Parallax* can employ these same modifications to avoid conflicts. For instance, using a small number of long gadgets or NOP-gadgets is sufficient to allow *Parallax* to operate in unison with heuristic system-level ROP-monitoring tools [97]. Since such gadgets are present by nature in nearly all applications, *Parallax* can use them without opening the application up to ROP attacks any more than it already was.

Stronger Control-Flow Integrity (CFI) approaches [23; 133; 197; 199], including the original work by Abadi et al. [14], are applied at the application level rather than the system level. Full CFI, as proposed by Abadi et al., is difficult to combine with *Parallax*, due to its need to record (and thus reveal to an adversary) all legal control transfers (including those targeting protective gadgets). However, as these are application-level approaches, there is a large amount of leeway for balancing the level of CFI enforcement against the desired level of tamperproofing per binary.

5.8.3 Protection Coverage

Parallax provides different protection tradeoffs than oblivious hashing. (1) As mentioned, OH can only protect code with deterministic execution state [55]. Arguably, non-deterministic code is more likely to be targeted by adversaries than deterministic code. For instance, adversaries commonly modify control flow instructions which depend on external inputs or unpredictable system call return values. Thus, a significant advantage of our technique is that it can protect both deterministic and non-deterministic code. (2) Oblivious hashing covers only code paths of which the state was recorded during testing. In contrast, our technique is completely static, and can be applied even to untested code.

In general, self-sufficient tamperproofing systems cannot implement indefinite attack resistance [27]. Rather, *Parallax* raises the bar for attackers, and increases the effort required to tamper with protected code. A determined adversary may eventually succeed in tampering with code by ensuring one or more of the following conditions. (1) The modifications reside entirely in instructions without overlapping gadgets. As discussed in Section 5.7, *Parallax* attempts to minimize such instructions. (2) Protected code is modified such that the resulting gadgets do not affect the outcome of the verification code. (3) Protected code is altered such that the resulting gadgets are semantically equivalent to the originals (including memory/register allocation). These conditions significantly restrict the modifications that can be safely made, making it harder for an attacker to implement arbitrary modifications.

5.9 Related Work

Traditional anti-tampering algorithms make use of code introspection, typically in the form of checksumming [60]. A highly resilient example of such an algorithm was proposed by Chang et al. [51], who use a network of cross-verifying code regions based on checksumming. Unfortunately, Wurster et al. have shown all such algorithms to be inherently vulnerable to an attack which exploits the distinct handling of code and data in modern processors [189]. The attack completely defeats all introspection-based algorithms by allowing an attacker to freely modify code in the processor's instruction cache, while leaving the data cache untouched. Later work has explored methods to re-enable code self-checksumming by implementing checks to detect the attack of Wurster et al [93]. Unfortunately, these checks require W$\oplus$X protection to be disabled, making the checksummed binary vulnerable to traditional code injection.

The foremost among the few algorithms designed to defeat this attack is oblivious hashing [55; 102]. OH verifies code integrity by checking that hashes of the execution state correspond to known correct values. In principle, it provides strong protection which is difficult to circumvent. However, the execution state is required to be deterministic, preventing OH from protecting code with non-deterministic inputs, like environment parameters or system call return values. The main benefit of our approach compared to OH is that it can protect code regions which OH cannot.

Previous work has proposed overlapping non-gadget instructions for tamperproofing [102; 118]. Instruction-level overlapping is only applicable to architectures with variable-length byte-aligned instructions [102]. In contrast, our ROP-based approach does not have this restriction [45; 52]. Furthermore, overlapping non-gadget instructions requires the insertion of additional jumps and partial instructions in the protected code, which leads to whole-program slowdowns of up to 3$\times$ [102]. Our approach provides better overall performance, and can keep performance overhead isolated from the protected code itself. Another approach to overlapping is to share common code blocks between functions. The usefulness of this approach is limited, as most common code blocks found in real-world binaries are non-sensitive instruction sequences like function prologues. It is typically not possible to protect non-trivial code blocks longer than one instruction using this approach [102].

Concurrently with our work, Lu et al. have explored the use of ROP for code obfuscation [119]. However, they do not consider tamperproofing, and thus do not explore how to maximize the coverage of protective gadgets, or how to craft gadgets which overlap with sensitive instructions. Instead, their work focuses on the use of existing (partial) gadgets to create ROP chains which are embedded with the intent of hiding functionality. Furthermore, Lu et al. do not attempt to prevent adversaries from tampering with their ROP chains once these are discovered. Similarly, prior work has proposed code hiding techniques based on function reuse, but this work has not focused on extending this to tamperproofing [117].

5.10 Conclusion

We introduced a novel code self-verification technique based on overlapping ROP gadgets with selected code. Several rewriting rules can be used to increase the coverage of protective gadgets, such that up to 90% of all code bytes are protectable. This coverage exceeds that of oblivious hashing, and our technique provides better protection for commonly attacked non-deterministic control flow instructions. Unlike code introspection-based verification algorithms, our approach is not vulnerable to direct instruction cache modification attacks. Furthermore, in contrast to oblivious hashing algorithms, our approach can protect non-deterministic code. The performance overhead of our approach can be confined to verification code which is separate from the protected code. Thus, performance-sensitive code is protectable without any slowdown, confining the performance penalty to other code. The performance overhead for programs protected using our technique is less than 4%.

Discussion

In Chapters 3–5, we have explored several methods for the efficient and safe implementation of security solutions for binaries. We now briefly recapitulate these methods, discussing their tradeoffs and applications for future binary-based work. For the purposes of this discussion, we differentiate between strategies for (1) maximizing crash-safety, and (2) minimizing runtime and analysis overhead.

Crash-safety

We have discussed several approaches for using binary rewriting to add security to binaries, while ensuring that these modifications will not cause crashes or undefined behavior, despite potential inaccuracies in the underlying binary analysis primitives. Chapter 6 provides more details on the precision yielded by modern binary analysis platforms for commonly used binary analysis primitives, providing more insight into the potential inaccuracies to anticipate while selecting which specific crash-safety methods to apply.

Overapproximation As evaluated and discussed in detail in Chapter 6, it is typically not reasonable to expect perfect precision in primitives such as the (I)CFG. Thus, binary analyses must often choose to either *overapproximate* or *underapproximate* a used primitive. Overapproximation is useful in situations where an omission in a binary analysis primitive leads to false positive or false negative security alerts, or even crashes or undefined behavior.

For instance, *PathArmor* (Chapter 4) applies overapproximation on the ICFG and on the estimation of may-call semantics for indirect calls. This is a sensible strategy for CFI solutions, as a missing control transfer will lead the CFI system to mistakenly label some legal flows as illegal. As a tradeoff in this case, overapproximation typically leads to the (conservative) inclusion of some edges or paths which are not actually legal, thus reducing security by introducing new paths which can potentially be abused by an attacker.

Another example of overapproximation is seen in *StackArmor*'s SP analyzer (Chapter 3), which statically analyzes stack frames for vulnerabilities to determine the degree of protection needed. Here, the tradeoff is not a reduction in security, but

a potential increase in runtime overhead, as *StackArmor* may conservatively protect some stack frames which do not actually require protection.

Underapproximation Overapproximation aims to minimize false negatives in a disassembly primitive (such as missing edges in a CFG), at the cost of introducing false positives. In contrast, underapproximation minimizes false positives, at the cost of false negatives.

To see why this is sometimes useful, consider *StackArmor*'s DA and BR analyzers, which perform binary-level data structure analysis to determine which stack objects to isolate from each other. Here, we would prefer that the analysis fails to distinguish some stack objects from each other (underapproximation), instead of mistakenly splitting a single object into two (overapproximation). *StackArmor* is designed such that the failure to isolate some stack objects gracefully reduces security, without any other adverse effects.

Minimizing instrumentation errors A given security policy is typically achievable through multiple instrumentation methods. For instance, coarse-grained CFI policies [197; 199] can be built upon function-level analysis, or alternatively, by instrumenting all `call` and `ret` sites. However, given that function detection is far more error-prone than instruction-level analysis (as discussed in Chapter 6), the latter method of instrumentation leads to a more reliable implementation. Thus, while function-level analysis cannot always be avoided (and indeed, is used in both *PathArmor* and *StackArmor*), it is best used only when necessary. When using imprecise binary analysis primitives such as function detection is unavoidable, other methods of ensuring instrumentation safety should be used in unison.

Runtime Analysis Given the difficulty of static analysis, we sometimes use limited runtime analysis to achieve precision which is hard to obtain statically. For instance, *PathArmor* uses load-time detection of PLT/GOT entries and library base addresses. We minimize our usage of runtime analysis and error correction to avoid the large overheads imposed by full-scale dynamic analysis (as discussed in Chapter 2).

Policy-driven Checks As discussed in Chapter 2, it is generally not reasonable to rely on the presence of source-level information or debugging symbols, as this information is likely to be absent for legacy or proprietary binaries. However, because symbolic information can greatly improve the ease and precision of binary analysis, it is typically worth it to devote additional implementation effort to make use of such information when it is available. Therefore, both *PathArmor* and *StackArmor* implement a *policy-driven* approach, which allows them to gracefully scale analysis precision and security guarantees up or down depending on the level of symbolic information available. *Parallax* also makes (optional) use of source-level information to aid the selection of code to be used for protection and verification, and to ease binary rewriting.

Runtime and Analysis Performance

In addition to minimizing instrumentation errors, we also aim to keep runtime overhead within tolerable limits. Moreover, (static) analysis time should be kept within reasonable limits; while this is typically of lesser importance, we want to avoid non-scalable analyses.

Lightweight Instrumentation Given the substantial runtime overheads imposed by both static and dynamic instrumentation, the amount of instrumentation code should be reduced to a minimum to avoid unnecessary performance impact. For instance, PEBIL conservatively saves all registers that may be clobbered before jumping to instrumentation code, and restores them afterwards [114]. As such save/restore operations are quite expensive, we reduced these operations to save only the registers actually used in the *StackArmor* instrumentation, leading to a significant performance increase. Moreover, it is sometimes possible to avoid the need for instrumentation altogether by implementing more advanced offline static analysis, such as *StackArmor*'s SP analyzer which performs heavyweight instrumentation only for stack frames that are (conservatively) determined to need it.

Runtime Analysis When using context-sensitive or path-sensitive static analysis, the analysis time required increases exponentially with the number of branches in the binary. Even for moderately-sized binaries, the analysis quickly becomes unscalable, necessitating the postponing of some analysis steps until runtime, when more complete information about the taken control flow path is available. We implemented this approach to great effect in *PathArmor*'s path analyzer component, which performs path analysis on a Just-in-Time (JIT) basis only for paths executed at runtime.

II Evaluating and Improving Disassembly

Outline

In Part I of this thesis, we have demonstrated the feasibility of binary-level methods for adding security to binaries, even based on imperfect binary analysis primitives. We derived several strategies for ensuring crash-safety and analysis efficiency in binary-based security systems.

To facilitate the design of future systems, we now focus our attention on quantifying the precision of commonly used disassembly primitives in modern binary analysis systems, and measuring the true prevalence of complex cases in real-world binaries. In addition, we implement a novel function detection strategy, which improves the precision of function identification (as we will show, a much used but highly imprecise primitive) while eliminating the dependence on compiler-/architecture-specific signatures. Our main contributions in this part are thus as follows.

C(1): Quantifying the precision of disassembly primitives allows us to design more efficient and more secure systems, by improving our understanding of precision and performance tradeoffs and the effects of the safety and efficiency methods explored in Part I.

C(2): As mentioned in Chapter 1, our evaluation of disassembly precision addresses common concerns among reviewers and researchers of binary-level systems, regarding the applicability and reliability of such systems in practice.

C(3): Our improved function detection approach translates directly to improved precision (and thus, in many cases, improved security) in function-level systems, including CFI systems and automated bug detection systems.

There is thus a symbiotic relationship between Part I and Part II of this thesis: Part II provides a foundation and improved primitives for building such systems as described in Part I, while Part I has made clear the concerns regarding binary analysis work, and the binary analysis properties on which to focus our attention in Part II. We subdivide the following discussion as follows.

(1) Chapter 6 focuses on an in-depth evaluation of x86/x64 disassembly on realistic compiler-generated binaries, addressing C(1) and C(2).

(2) We present our function detection work in Chapter 7, addressing C(3).

Chapter 6

An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries

It is well-known that static disassembly is an unsolved problem, but how much of a problem is it in real software—for instance, for binary protection schemes? This chapter studies the accuracy of nine state-of-the-art disassemblers on 981 real-world compiler-generated binaries with a wide variety of properties. In contrast, prior work focuses on isolated corner cases; we show that this has led to a widespread and overly pessimistic view on the prevalence of complex constructs like inline data and overlapping code, leading reviewers and researchers to underestimate the potential of binary-based research. On the other hand, some constructs, such as function boundaries, are much harder to recover accurately than is reflected in the literature, which rarely discusses much needed error handling for these primitives. We study 30 papers recently published in six major security venues, and reveal a mismatch between expectations in the literature, and the actual capabilities of modern disassemblers. Our findings help improve future research by eliminating this mismatch.

6.1 Introduction

The capabilities and limitations of disassembly are not always clearly defined or understood, making it difficult for researchers and reviewers to judge the practical feasibility of techniques based on it. At the same time, disassembly is the backbone of research in static binary instrumentation [31; 114; 149], binary code lifting to LLVM IR (for reoptimization or analysis) [179], binary-level vulnerability search [140], and binary-level anti-exploitation systems [23; 54; 144; 197]. Disassembly is thus crucial for analyzing or securing untrusted or proprietary binaries, where source code is simply not available.

The accuracy of disassembly strongly depends on the analyzed binary. In the most general case, the disassembler can make very few assumptions on the structure of a binary—high-level concepts like functions and loops have no real significance at the binary level [25]. Moreover, the binary may contain complex constructs, such as overlapping or self-modifying code, or inline data in executable regions. This is especially true for obfuscated binaries, making disassembly of such binaries extremely challenging. Disassembly in general is undecidable [188]. On the other hand, one might expect that compilers emit code with more predictable properties, containing a limited set of patterns that the disassembler may try to identify.

Whether this is true is not well recognized, leading to a wide range of views on disassembly. These vary from the stance that disassembly of benign binaries is a solved problem [199], to the stance that complex cases are rampant [125]. It is unclear which view is justified in a given situation. The aim of our work is thus to study binary disassembly in a realistic setting, and more clearly delineate the capabilities of modern disassemblers.

It is clear from prior work that obfuscated code may complicate disassembly in a myriad of ways [110; 118]. We therefore limit our study to non-obfuscated binaries compiled on modern x86 and x64 platforms (the most common in binary analysis and security research). Specifically, we focus on binaries generated with the popular `gcc`, `clang` and Visual Studio compilers. We explore a wide variety of 981 realistic binaries, including stripped, optimized, statically linked, and link-time optimized binaries, as well as library code that includes handcrafted assembly. We disassemble these binaries using nine state-of-the-art research and industry disassemblers, studying their ability to recover all disassembly primitives commonly used in the literature: instructions, function start addresses, function signatures, Control Flow Graphs (CFG) and callgraphs. In contrast, prior studies focus strongly on complex corner cases in isolation [125; 135]. Our results show that such cases are exceedingly rare, even in optimized code, and that focusing on them leads to an overly pessimistic view on disassembly.

We show that many disassembly primitives can be recovered with better accuracy than previously thought. For instance, instruction accuracy often approaches 100%, even using linear disassembly. On the other hand, we also identify some primitives which are more difficult to recover—most notably, function start information.

To facilitate a better match between the capabilities of disassemblers and the expectations in the literature, we comprehensively study all binary-based papers published in six major security conferences in the last three years (2013–2015). Ironically, this study shows a focus in the literature on rare complex constructs, while little attention is devoted to error handling for primitives that really are prone to inaccuracies. For instance, only 25% of Windows-targeted papers that rely on function information discuss potential inaccuracies, even though the accuracy of function detection regularly drops to 80% or less. Moreover, less than half of all papers implement mechanisms to deal with inaccuracies, even though in most cases errors can lead to malignant failures like crashes.

Contributions & Outline The contributions in this chapter are as follows.

- We study disassembly on 981 full-scale compiler-generated binaries, to clearly define the true capabilities of modern disassemblers (Section 6.3) and the implications on binary-based research (Section 6.4).
- Our results allow researchers and reviewers to accurately judge future binary-based research—a task currently complicated by the myriad of differing opinions on the subject. To this end, we release all our raw results and ground truth for use in future evaluations of binary-based research.[1]
- We analyze the quality of all recent binary-based work published in six major security venues by comparing our results to the requirements and assumptions of this work (Section 6.5). This shows where disassembler capabilities and the literature are mismatched, and how this mismatch can be resolved moving forward (Section 6.6).

6.2 Evaluating Real-World Disassembly

This section outlines our disassembly evaluation approach. We discuss our results, and the implications on binary-based research, in Sections 6.3–6.4. Sections 6.5–6.6 discuss how closely expectations in the literature match our results.

6.2.1 Binary Test Suite

We focus our analysis on non-obfuscated x86 and x64 binaries generated with modern compilers. Our experiments are based on Linux (ELF) and Windows (PE) binaries, generated with the popular `gcc` v5.1.1, `clang` v3.7.0 and Visual Studio 2015 compilers—the most recent versions at the time of writing. The x86/x64 instruction set is the most common target in binary-based research. Moreover, x86/x64 is a variable-length instruction set, allowing unique constructs such as overlapping and "misaligned" instructions which can be difficult to disassemble. We exclude obfuscated binaries, as there is no doubt that they can wreak havoc on disassembler performance and we hardly need confirm this in our experiments.

We base our experiments on a test suite composed of the SPEC CPU2006 C and C++ benchmarks, the widely used and highly optimized `glibc-2.22` library, and a set of popular server applications consisting of `nginx` v1.8.0, `lighttpd` v1.4.39, `opensshd` v7.1p2, `vsftpd` v3.0.3 and `exim` v4.86. This test suite has several properties which make it representative: (1) It contains a wide variety of realistic C and C++ binaries, ranging from very small to large; (2) These correspond to binaries used in evaluations of other work, making it easier to compare our results; (3) The tests include highly optimized library code, containing handwritten assembly and complex cases which regular applications do not; (4) SPEC CPU2006 compiles

[1] Our data set and documentation are available at https://www.vusec.net/projects/disassembly/.

on both Linux and Windows, allowing a fair comparison of results between `gcc`, `clang`, and Visual Studio.

To study the impact of compiler options on disassembly, we compile the SPEC CPU2006 part of our test suite multiple times with a variety of popular configurations. Specifically: (1) Optimization levels `O0`, `O1`, `O2` and `O3` for `gcc`, `clang` and Visual Studio; (2) Optimization for size (`Os`) on `gcc` and `clang`; (3) Static linking and link-time optimization (`-flto`) on 64-bit `gcc`; (4) Stripped binaries, as well as binaries with symbols. We compile the servers for both x86 and x64 with `gcc` and `clang`, leaving all remaining settings at the Makefile defaults. Finally, we compile `glibc-2.22` with 64-bit `gcc`, to which it is specifically tailored. In total, our test suite contains 981 binaries and shared objects.

6.2.2 Disassembly Primitives

We test all five common disassembly primitives used in the literature (see Section 6.5). Some of these go well beyond basic instruction recovery, and are only supported by a subset of the disassemblers we test. Specifically, we study disassembly accuracy for the following primitives: (1) *Instructions*, (2) *Function starts*, (3) *Function signatures*, (4) the *Interprocedural Control Flow Graph* (ICFG), and (5) the *Callgraph*. These primitives are defined in Chapter 2.1.

We focus on the ICFG (the union of the per-function CFGs, see Chapter 2.1), rather than individual CFGs, because disassemblers often deviate from the traditional CFG; typically by omitting indirect edges, and sometimes by defining a global ICFG only rather than per-function CFGs. Focusing our measurements on the coverage of basic blocks in the ICFG allows us to abstract from the disassemblers' varying CFG definitions. We pay special attention to hard-to-resolve basic blocks, such as the heads of address-taken functions and switch/case blocks reached via jump tables.

Similarly to the CFG, many disassemblers deviate from the traditional callgraph definition by including only direct call edges (largely due to the difficulty of statically resolving indirect flows). In accordance with our definitions from Chapter 2.1, our experiments therefore measure the completeness of this *direct callgraph*, and we consider indirect calls and tailcalls separately in our complex case analysis.

6.2.3 Complex Constructs

We also study the prevalence in real-world binaries of complex corner cases which are often cited as particularly harmful to disassembly [31; 125; 161]. We study the following complex cases: (1) *Overlapping/shared basic blocks*, (2) *Overlapping instructions*, (3) *Inline data and jump tables*, (4) *Switches/case blocks* (in the context of indirect edge resolution), (5) *Alignment/padding bytes*, (6) *Multi-entry functions*, and (7) *Tail calls*. These complex cases are described in Chapter 2.7. Note that in this chapter, we study only the complex cases relevant to disassembly in general; we do not discuss cases which are specific to function detection. These cases (such as non-

contiguous functions and alternative prologues/epilogues), and methods for dealing with them, are discussed in more detail in the context of our function detection work in Chapter 7.

6.2.4 Disassembly & Testing Environment

We conducted all disassembly experiments on an Intel Core i5 4300U machine with 8GB of RAM, running Ubuntu 15.04 with kernel 3.19.0-47. We compiled our `gcc` and `clang` test cases on this same machine. The Visual Studio binaries were compiled on an Intel Core i7 3770 machine with 8GB of RAM, running Windows 10.

We tested nine popular industry and research disassemblers: *IDA Pro v6.7* [83], *Hopper v3.11.5* [69], *Dyninst v9.1.0* [31], *BAP v0.9.9* [44], *ByteWeight v0.9.9* [26], *Jakstab v0.8.4* [106], *angr v4.6.1.4* [165], *PSI v1.1* [198] (successor of BinCFI [199]), and *objdump v2.22* [95].

ByteWeight yields only function starts, while Dyninst and PSI support only ELF binaries (for Dyninst, this is due to our Linux testing environment). Jakstab supports only x86 PE binaries. We omit angr results for x86, as angr is optimized for x64. PSI is based on objdump, with added error correction. Section 6.3 shows that PSI (and all linear disassemblers) perform equivalently to objdump; therefore, we group these under the name *linear disassembly*. All others are recursive descent disassemblers, as defined in Chapter 2.4.

6.2.5 Ground Truth

Our experiments require precise ground truth on instructions, basic blocks and function starts, call sites, function signatures and switch/case addresses. Much of this information is normally only available at the source level. Clearly, we cannot obtain our ground truth from any disassembler, as this would bias our experiments.

We base our ELF ground truth on information collected by an LLVM analysis pass, and on DWARF v3 debugging information. Specifically, we use LLVM to collect source-level information, such as the source lines belonging to functions and switch statements. We then compile our test binaries with DWARF information, and link the source-level line numbers to the binary-level addresses using the DWARF line number table. We also use DWARF information on function parameters for our function signature analysis. We strip the DWARF information from the binaries before our disassembly experiments.

The line number table provides a full mapping of source lines to binary, but not all instructions correspond directly to a source line. To find these instructions, we use *Capstone v3.0.4* to start a conservative linear disassembly sweep from each known instruction address, stopping at control flow instructions unless we can guarantee the validity of their destination and fall-through addresses. For instance, the target of a direct unconditional jump instruction can be guaranteed, while its fall-through block cannot (as it might contain inline data).

This approach yields ground truth for over 98% of code bytes in the tested binaries. We manually analyze the remaining bytes, which are typically alignment code unreachable by control flow. The result is a ground truth file for each binary test case, that specifies the type of each code byte, as well as instruction and function starts, switch/case addresses, and function signatures.

We use a similar method for the Windows PE tests, but based on information from PDB (Program Database) files produced by Visual Studio instead of DWARF. This produces files analogous to our ELF ground truth format.

We release all our ground truth files and our test suite, to aid in future evaluations of binary-based research and disassembly.

6.3 Disassembly Results

This section describes the results of our disassembly experiments, using the methodology as outlined in Section 6.2. We first discuss our results for application binaries (SPEC CPU2006 and servers), followed by a separate discussion on highly optimized libraries. Finally, we discuss the impact of static linking and link-time optimization. We release all our raw results as part of our data set, and present aggregated results here for space reasons.

6.3.1 Application Binaries

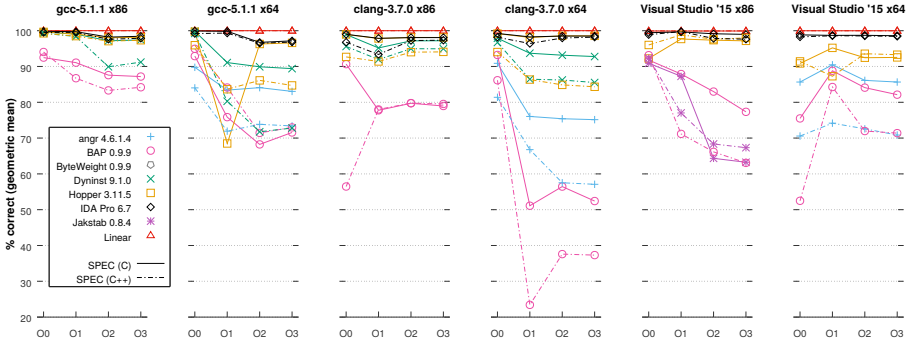
This section presents disassembly results for application code. We discuss accuracy results for all primitives, and also analyze the prevalence of complex cases.

6.3.1.1 SPEC CPU2006 Results

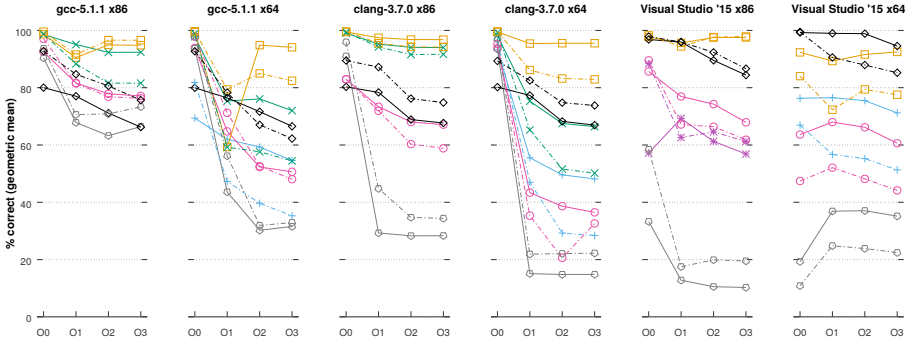
Figures 6.1a–6.1e show the accuracy for the SPEC CPU2006 C and C++ benchmarks of the recovered instructions, function starts, function signatures, CFGs and call-graphs, respectively. We show the percentage of correctly recovered (true positive) primitives for each tested compiler at optimization levels `O0`–`O3`. Note that the legend in Figure 6.1a applies to Figures 6.1a–6.1e. All lines are geometric mean results (simply referred to as "mean" from this point); arithmetic means and standard deviations are discussed in the text where they differ significantly. We show separate results for the C and C++ benchmarks, to expose variations in disassembly accuracy that may result from different code patterns.

Some disassemblers support only a subset of the tested primitives. For instance, linear disassembly provides only instructions, and IDA Pro is the only tested disassembler that provides function signatures. Moreover, some disassemblers only support a subset of the tested binary types (for instance, only x64 ELF), and are therefore only shown in the plots where they are applicable. For clarity, the graphs only show results for stripped binaries; our tests with standard symbols (not DWARF information) are discussed in the text.

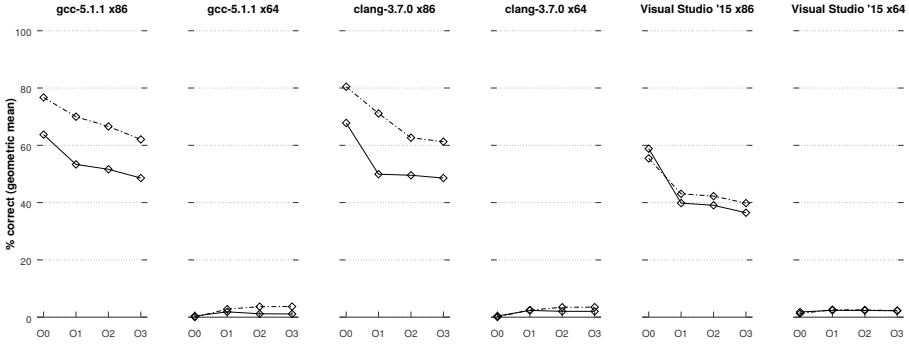
Figure 6.1: Disassembly results. The legend in Figure 6.1a applies to Figures 6.1a–6.1e. Section 6.2.4 describes which platforms are supported by each tested disassembler.



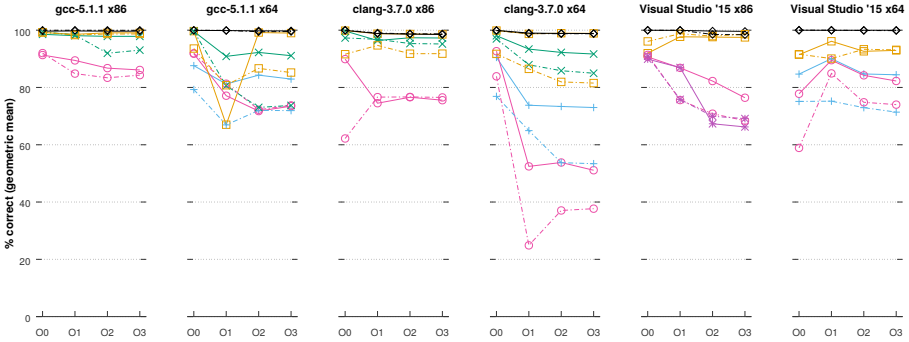
(a) Correctly disassembled instructions.



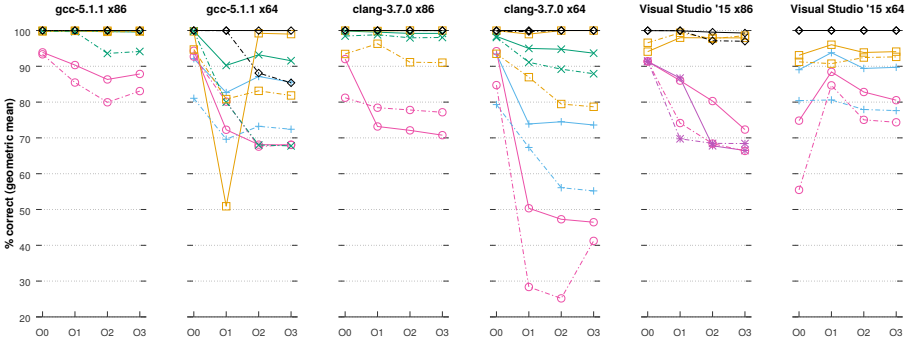
(b) Correctly detected function start addresses.



(c) Correctly detected non-empty function argument lists (IDA Pro only).



(d) Correct and complete basic blocks for the ICFG.



(e) Correctly resolved direct function calls (indirect calls discussed separately).

Instruction boundaries Figure 6.1a shows the percentage of correctly recovered instructions. Interestingly, linear disassembly consistently outperforms all other disassemblers, finding 100% of the instructions for `gcc` and `clang` binaries (without false positives), and 99.92% in the worst case for Visual Studio.

Linear disassembly The perfect accuracy for linear disassembly with `gcc` and `clang` owes to the fact that these compilers never produce inline data, not even for jump tables. Instead, jump tables and other data are placed in the `.rodata` section.

Visual Studio *does* produce inline data, typically jump tables. This leads to some false positives with linear disassembly (data treated as code), amounting to a worst-case mean of 989 false positive instructions (0.56% of the disassembled code) for the x86 C++ tests at `O3`. The number of missed instructions (due to desynchronization) is much lower, at a worst-case mean of 0.09%. This is because x86/x64 disassembly typically resynchronizes itself within two or three instructions [118].

Recursive disassembly The most accurate recursive disassembler in terms of instruction recovery is IDA Pro 6.7, which closely follows linear disassembly with an instruction coverage exceeding 99% at optimization levels `O0` and `O1`, dropping

to a worst case mean of 96% for higher optimization levels. The majority of missed instructions at higher optimization levels are alignment code for functions and basic blocks, which is quite common in optimized binaries. It consists of various (long) `nop` instructions for `gcc` and `clang`, and of `int 3` instructions for Visual Studio, and accounts for up to 3% of all code at `O2` and `O3`. Missing these instructions is not harmful to common binary analysis operations, such as binary instrumentation, manual analysis or decompilation.

False positives in IDA Pro are less prevalent than in linear disassembly. On `gcc` and `clang`, they are extremely rare, amounting to 14 false positives in the worst test case, with a mean of 0. Visual Studio binaries produce more false positives, peaking at 0.16% of all recovered instructions. Overall, linear disassembly provides the most complete instruction listing, but at a relatively high false positive rate for Visual Studio. IDA Pro finds only slightly fewer instructions, with significantly fewer false positives. These numbers were no better for binaries with symbols.

Dyninst and Hopper achieve best case accuracy comparable to IDA, but not quite as consistently. Some disassemblers, notably BAP, appear to be optimized for `gcc`, and show large performance drops when used on `clang`. The BAP authors informed us that BAP's results depend strongly on the disassembly starting points (i.e., function starts), provided by ByteWeight. We used the default ELF and PE signature files shipped with ByteWeight v0.9.9. Our angr results are based on the *CFGFast* analysis recommended to us by the angr authors.

Overall, IDA Pro, Hopper, Dyninst and linear disassembly show arithmetic mean results which are extremely close to the geometric means, exhibiting standard deviations below 1%. The other disassemblers have larger standard deviations, typically around 15%, with outliers up to 36% (for BAP on `clang` x86, as visible in Figure 6.1a).

C versus C++ Accuracy between C and C++ differs most in the lower scoring disassemblers, but the difference largely disappears in the best performing disassemblers. The largest relative difference appears for `clang`.

Function starts The results for function start detection are far more diffuse than those for instruction recovery. Consider Figure 6.1b, which shows the mean percentage of correctly recovered function start addresses. No one disassembler consistently dominates these results, though Hopper is at the upper end of the spectrum for most compiler configurations in terms of true positives. Dyninst also provides high true positive rates, though not as consistently as Hopper. However, as shown in Figure 6.2, both Hopper and Dyninst suffer from high false positive rates, with worst case mean false positive rates of 28% and 19%, respectively. IDA Pro provides lower false positive rates of under 5% in most cases (except for x86 Visual Studio, where it peaks at 20%). However, its true positive rate is substantially lower than those of Hopper and Dyninst, regularly missing 20% or more of functions even at low optimization levels. As with instruction recovery, the results for BAP and ByteWeight

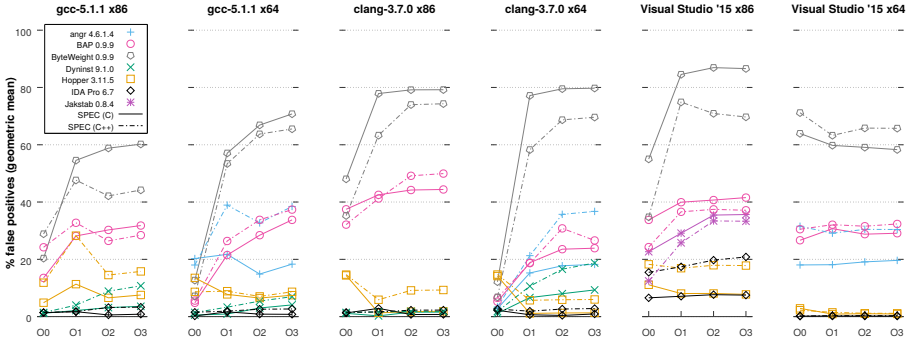


Figure 6.2: False positives for function start detection (percentage of total detected functions).

depend heavily on the compiler configuration, ranging from over 90% accuracy on `gcc` x86 at `O0`, to under 20% on `clang` x64.

Even for the best performing disassemblers, function start identification is far more challenging than instruction recovery. Accuracy drops particularly as the optimization level increases, repeatedly falling from close to 99% true positives at `O0`, to only 82% at `O3`, and worsened by high false positive rates. For IDA Pro, the worst case mean true positive rate is even lower, falling to 62% for C++ on x64 `gcc` at `O3`. Moreover, the standard deviation increases to over 15% even for IDA Pro.

False negatives The vast majority of false negatives is caused by indirectly called or tailcalled functions (reached by a `jmp` instead of a `call`), as shown in Listing 6.1. This explains why the true positive rate drops steeply at high optimization levels, where tail calls and functions lacking standard prologues are common (see Section 6.3.1.3). Symbols, if available, help greatly in improving accuracy. They are used especially effectively by IDA Pro, which consistently yields over 99% true positives for binaries with symbols, even at higher optimization levels.

False positives Several factors contribute to the false positive rate. We analyzed a random sample of 50 false positives for Dyninst, Hopper and IDA Pro, the three best performing disassemblers in function detection.

For Dyninst, false positives are mainly due to erroneously applied signatures for function prologues and epilogues. As an example, Listing 6.2 shows a false positive in Dyninst due to a misidentified prologue: Dyninst scans for the `push %r15` instruction (as well as several other prologue signatures), missing preceding instructions in the function. We observe similar cases for function epilogues. For instance, as shown in Listings 6.3 and 6.4, Dyninst assumes a new function following a `ret; nop` instruction sequence. This is not always correct: as shown in the examples, the same code pattern can result from a multi-exit function with padding between basic blocks. Note that both examples could be handled correctly by control

```
6caf10 <ix86_fp_compare_mode>:
  6caf10:  mov  0x3f0dde(%rip),%eax
  6caf16:  and  $0x10,%eax
  6caf19:  cmp  $0x1,%eax
  6caf1c:  sbb  %eax,%eax
  6caf1e:  add  $0x3a,%eax
  6caf21:  retq
```

Listing 6.1: False negative indirectly called function for IDA Pro in `gcc`, compiled with `gcc` at `O3` for x64 ELF. Note the lack of a standard prologue.

```
480970 <autohelperowl_defendpat156>:
  480970:  push   %rbp
  480971:  push   %r15
  480973:  push   %r14
  480975:  push   %rbx
  480976:  push   %rax
```

Listing 6.2: False positive function (shaded) for Dyninst, due to misapplied prologue signature, `gobmk` compiled with `clang` at `O1` for x64 ELF.

```
  8060985:  pop    %ebx
  8060986:  pop    %esi
  8060987:  ret
  8060988:  nop
  8060989:  lea    0x0(%esi,%eiz,1),%esi
```

Listing 6.3: False positive function (shaded) for Dyninst, due to code misinterpreted as epilogue, `sphinx` compiled with `gcc` at `O2` for x86 ELF. In this case, the instruction at address `0x8060989` is actually a do-nothing instruction emitted for padding.

flow and semantics-aware disassemblers. In Listing 6.4, there are intraprocedural jumps towards the basic block at `0x46bb50`, showing that it is not a new function. The false positive in Listing 6.3 is in effect a `nop` instruction, emitted for padding by `gcc` on x86.

All false positives we sampled for Hopper are located directly after padding code (mistakenly interpreted as padding between functions), or after a direct `jmp` (without a fallthrough edge), and are not directly reached by other instructions. An example is shown in Listing 6.5. Since these instructions are never reached directly, Hopper assumes that they represent function starts. This is not always correct; for instance, the same pattern frequently results from case blocks belonging to switch statements, as seen in Listing 6.5.

Similarly, the majority of false positives for IDA Pro are also caused by unreachable code assumed to be a new function. However, these cases are far less common in IDA Pro than in Hopper, as IDA Pro more accurately resolves difficult control flow constructs such as switches. Interestingly, the false positive rate for IDA Pro drops to a mean of under 0.3% for x64 Visual Studio 2015. This is because 64-bit Visual Studio uses just one well-defined calling convention, while other compilers use a variety [124].

```
46b990 <Perl_pp_enterloop>:
          [...]
  46ba02:  ja     46bb50 <Perl_pp_enterloop+0x1c0>
  46ba08:  mov    %rsi,%rdi
  46ba0b:  shl    %cl,%rdi
  46ba0e:  mov    %rdi,%rcx
  46ba11:  and    $0x46,%ecx
  46ba14:  je     46bb50 <Perl_pp_enterloop+0x1c0>
          [...]
  46bb47:  pop    %r12
  46bb49:  retq
  46bb4a:  nopw   0x0(%rax,%rax,1)
  46bb50:  sub    $0x90,%rax
```

Listing 6.4: False positive function (shaded) for Dyninst, due to code misinterpreted as epilogue, `perlbench` compiled with `gcc` at `O3` for x64 ELF.

```
  42cec3:  movss  %xmm0,-0x340(%rbp)
  42cecb:  jmpq   42cfc8 <P7PriorifyTransitionVector+0x622>
  42ced0:  mov    -0x344(%rbp),%eax
```

Listing 6.5: False positive function (shaded) for Hopper, due to misclassified switch case block, `hmmer` compiled with `gcc` at `O0` for x64 ELF.

Function signatures Of the tested disassemblers, only IDA Pro supports function signature analysis. Figure 6.1c shows the percentage of non-empty function argument lists where IDA Pro correctly identified the number of arguments. We focus on non-empty argument lists because IDA Pro defaults to an empty list, skewing our results if counted as correct.

Argument recovery is far more accurate on x86 code, where parameters are typically passed on the stack, than it is on the register-oriented x64 architecture. For x86 code generated by `gcc` and `clang`, IDA Pro correctly identifies between 64% and 81% of the argument lists on non-optimized binaries, dropping to 48% in the worst case at `O3`. Results for Visual Studio are slightly worse, ranging from 36% worst case to 59% in the best case. As for function starts, the standard deviation is just over 15%. On x64 code, IDA Pro recovers almost none of the argument lists, with accuracy between 0.38% and 1.87%.

Performance is significantly better for binaries with symbols, even on x64, but only for C++ code. For instance, IDA Pro's accuracy for `gcc` x64 increases to a mean of 44% for C++, peaking at 75% correct argument lists. This is because IDA Pro parses mangled function names that occur in C++ symbols, which encode signature information in the function name.

Control Flow Graph accuracy Figure 6.1d presents the accuracy of basic blocks in the ICFG, the union of all function-level CFGs. We found these results to be representative of the per-function CFG accuracy. The accuracy of the ICFG is strongly correlated with instruction discovery; indeed, recursive disassemblers typically find instructions through the process of expanding the ICFG itself. Thus, the disassemblers that perform well in instruction recovery also perform well in CFG construc-

tion. For some disassemblers, such as IDA Pro, the basic block true positive rate at high optimization levels even exceeds the raw instruction recovery results (Figure 6.1a). This is because for the ICFG, we did not count missing `nop` instructions as false negatives.

IDA Pro consistently achieves a basic block recovery rate of between 98–100%, even at high optimization levels. Even at moderate optimization levels, the results for Hopper and Dyninst are considerably less complete, regularly dropping to 90% or less. For the remaining disassemblers, basic block recovery rates of 75% or less are typical.

All disassemblers except IDA Pro show a considerable drop in accuracy on `gcc` and `clang` for x64, compared to the x86 results. This is strongly correlated with the diminishing instruction and function detection results for these disassembler/architecture combinations (see Figures 6.1a–6.1b). This implies that when functions are missed, these disassemblers also fail to recover the instructions and basic blocks contained in the missed functions. In contrast, IDA Pro disassembles instructions even when it cannot attribute them to any function. The difference between x86/x64 and C/C++ results is less pronounced for Visual Studio binaries than for `gcc`/`clang`.

Callgraph accuracy Like ICFG accuracy, callgraph accuracy depends strongly on the completeness of the underlying instruction analysis. As mentioned, the callgraphs returned by the tested disassemblers contain only the direct call edges, and do not deal with address-taken functions. For this reason, Figure 6.1e presents results for the direct component of the callgraph only. We study the impact of indirect calls on function identification accuracy in our complex case analysis instead (see Section 6.3.1.3).

The direct callgraph results in Figure 6.1e again show IDA Pro to be the most accurate at a consistent 99% function call resolve rate (linking function call edges to function starts), in most cases followed closely by Dyninst and Hopper. This illustrates that the lower accuracy for function starts (Figure 6.1b) is mainly due to indirectly called functions (such as those called via function pointers) and tail call optimizations, as confirmed by our results in Section 6.3.1.2.

6.3.1.2 Server Results

Table 6.1 shows disassembly results for the servers from our test suite. For space reasons, and because the relative accuracy of the disassemblers is the same as for SPEC, we only show results for IDA Pro, the best overall disassembler. All other results are available as part of our data set, as mentioned at the start of Section 6.3. We compiled all servers for both x86 and x64 with `gcc` and `clang`, using their default Makefile optimization levels.

The server tests confirm that the SPEC results from Section 6.3.1.1 are representative; all results lie well within the established bounds. As with SPEC, linear dis-

	Instructions	Functions	Signatures	ICFG	Callgraph	Instructions	Functions	Signatures	ICFG	Callgraph
	x86					x64				
gcc-5.1.1										
nginx	99.9	65.5	49.6	100	100	99.9	59.2	0.9	99.9	100
lighttpd	99.9	99.5	85.9	99.9	100	99.9	99.5	0.0	99.9	100
vsftpd	95.4	93.4	73.6	95.9	99.5	93.0	92.5	4.3	99.9	100
opensshd	99.9	86.2	74.9	100	100	99.9	86.2	0.0	100	100
exim	99.9	90.1	58.2	99.9	100	99.9	89.9	4.5	99.9	100
clang-3.7.0										
nginx	98.5	57.5	44.0	99.5	100	98.6	53.0	0.7	99.4	100
lighttpd	98.7	99.5	87.9	99.9	100	99.0	99.5	0.0	99.9	100
vsftpd	96.8	93.3	72.9	99.8	100	97.0	92.0	6.6	99.5	99.9
opensshd	98.9	86.5	78.1	100	100	99.2	86.3	0.0	100	100
exim	99.0	82.7	54.6	99.3	100	99.1	81.7	5.4	99.4	100

Table 6.1: IDA Pro 6.7 disassembly results for server tests (% correct, per test case).

assembly achieved 100% correctness. The `nginx` results warrant closer inspection; given its optimization level `O1`, the function start and argument information is on the low end of the accuracy spectrum. Closer analysis shows that this results from extensive use in `nginx` of indirect calls through function pointers; Section 6.3.1.1 shows that this negatively affects function information. Indeed, for all tested servers, the accuracy of function start detection is inversely proportional to the ratio of address-taken functions to the total number of instructions. This shows that coding style can carry through the compilation process to have a strong effect on disassembler performance for the resulting binary.

6.3.1.3 Prevalence of Complex Constructs

Figure 6.3 shows the prevalence of complex constructs in SPEC CPU2006, which pose special disassembly challenges. We also analyzed these constructs in the server binaries, finding no significantly different results.

We did not encounter any overlapping or shared basic blocks in either the SPEC or server tests on any compiler. This is surprising, as these constructs are frequently cited in the literature [31; 106; 125]. Closer inspection showed that all the cited cases of overlapping blocks are due to constructs which we classify more specifically, namely overlapping instructions and multi-entry functions. These constructs are exceedingly rare, and occur almost exclusively in library code (discussed in Section 6.3.2.2). This finding fits with the examples seen in the literature, which all stem from library code, most commonly `glibc`.

No overlapping instructions occur in Linux application code, and only a handful in Windows code (with a mean of zero, and a maximum of 3 and 10 instructions for x86 and x64 Visual Studio, respectively). Multi-entry functions are somewhat more common. All cases we found consisted of functions with optional basic blocks that can execute before the main function body, and finish by jumping over the main

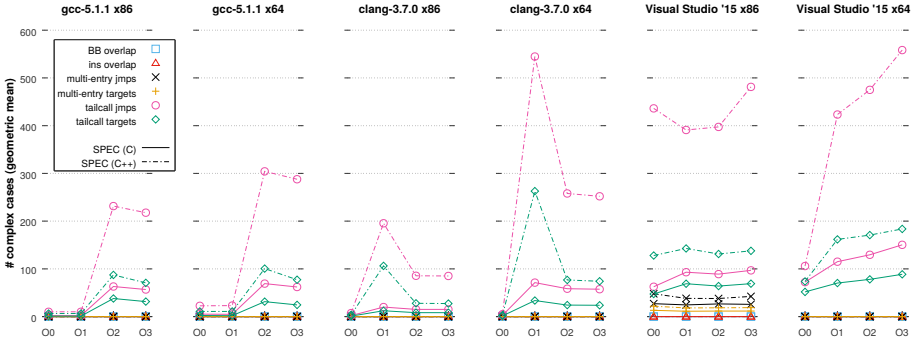


Figure 6.3: Prevalence of complex constructs in SPEC CPU2006 binaries.

function body prologue. Figure 6.3 lists such jumps as *multi-entry jumps*, and shows the targeted main function bodies as *multi-entry targets*. In binaries compiled with `gcc` and `clang`, we found up to 18 multi-entry jumps for C code, and up to 64 for C++, with the highest prevalence in x64 binaries. Visual Studio produced up to 172 multi-entry jumps for C, and up to 88 for C++, the construct being most prevalent in x86 code. This kind of multi-entry function is handled well by disassemblers in practice, producing no notable decrease in disassembly accuracy compared to other (single-entry) functions.

Tailcalls form the most prevalent complex case, and do negatively affect function start detection if the target function is never called normally (see Section 6.3.1.1, and the false negative analysis in Chapter 7). The largest number of tailcalls (listed as *tailcall jumps* in Figure 6.3) is found in `clang` x64 C++ binaries, at a mean of 545 cases. Visual Studio produces a similar number of tailcalls. For `clang`, the number of tailcalls peaks at optimization level `O1`, while Visual Studio peaks at `O3`. For `clang` (and to a lesser extent `gcc`), higher optimization levels can lead to a decrease in tailcalls through other modifications like code merging and code elimination.

Jump tables (due to switches) are by far the most common case of inline data. They occur as inline data only on Visual Studio (`gcc` and `clang` place jump tables in the `.rodata` section). As seen in Section 6.3.1.1, inline data causes false positive instructions especially in linear disassembly (peaking at 0.56% false positives).

Another challenge due to jump tables is locating all case blocks belonging to the switch; these are typically reached indirectly via a jump that loads its target address from the jump table. Linear disassembly covers 100% of case blocks correctly on `gcc` and `clang` (see Section 6.3.1.1), and also achieves very high accuracy for Visual Studio. The best performing recursive disassemblers, most notably IDA Pro, also achieve very high coverage of switch/case blocks; coverage of these blocks is comparable to the overall instruction/basic block recovery rates. This is because many recursive disassemblers have special heuristics for identifying and parsing standard jump tables and the corresponding code patterns.

	Instructions	Functions	Signatures	ICFG	Callgraph
gcc-5.1.1 x64					
angr	64.4	75.6	—	70.2	87.9
BAP	65.3	79.6	—	72.4	84.8
ByteWeight	—	29.3	—	—	—
Dyninst	79.7	85.2	—	87.6	95.5
Hopper	84.3	93.3	—	90.6	93.9
IDA Pro	96.0	92.0	5.4	99.9	99.9
Linear	99.9	—	—	—	—

Table 6.2: Disassembly results for `glibc` (% correct).

6.3.1.4 Optimizing for Size

At optimization levels `O0`–`O3`, no overlapping or shared basic blocks occur. A reasonable hypothesis is that compilers might more readily produce such blocks when optimizing for size (optimization level `Os`) rather than for performance. To verify this, we recompiled the SPEC C and C++ benchmarks with size optimization, and repeated our disassembly tests.

Even for size-optimized binaries, we did not find any overlapping or shared blocks. Moreover, the accuracy of the instruction boundaries, callgraph and ICFG did not significantly differ from our results for optimization levels `O0`–`O3`. Function starts and argument lists were comparable in precision to those for performance-optimized binaries (`O2`–`O3`).

6.3.2 Shared Library Objects

This section discusses our disassembly results and complex case analysis for library code. Libraries are often highly optimized, and therefore contain more complex (handcrafted) corner cases than application code. We focus our analysis on `glibc-2.22`, the standard C library used in GNU systems, compiled in its default configuration (`gcc` with optimization level `O2`). This is one of the most widespread and highly optimized libraries, and is often cited as a highly complex case [31; 125].

6.3.2.1 Disassembly Results

Table 6.2 shows disassembly results for `glibc-2.22`, for all tested disassemblers that support 64-bit ELF binaries. Nearly all disassemblers display significantly lower accuracy on instruction boundaries than the mean for application binaries in equivalent compiler configurations. Only IDA Pro and linear disassembly are on par with their performance on application code, achieving very good accuracy without any false positives. Note that objdump achieves 99.9% accuracy instead of the usual 100% for ELF binaries. This is because unlike IDA Pro, it does not explicitly separate the overlapping instructions that occur in `glibc` (see Section 6.3.2.2).

```
7b05a:  cmpl         $0x0,%fs:0x18
7b063:  je           7b066
7b065:  lock cmpxchg %rcx,0x3230fa(%rip)
```

Listing 6.6: Overlapping instruction in `glibc-2.22`.

```
e9a30 <splice>:
  e9a30:  cmpl   $0x0,0x2b9da9(%rip)
  e9a37:  jne    e9a4c <__splice_nocancel+0x13>
e9a39 <__splice_nocancel>:
  e9a39:  mov    %rcx,%r10
  e9a3c:  mov    $0x113,%eax
  e9a41:  syscall
  e9a43:  cmp    $0xfffffffffffff001,%rax
  e9a49:  jae    e9a7f <__splice_nocancel+0x46>
  e9a4b:  retq
  e9a4c:  sub    $0x8,%rsp
  e9a50:  callq  f56d0 <__libc_enable_asynccancel>
  [...]
```

Listing 6.7: Multi-entry function in `glibc-2.22`.

Function start results are on par with, or even exceed the mean for application binaries; this holds true for all disassemblers. Moreover, the accuracy of function argument lists (5.4%) is much higher than one would expect from the x64 SPEC CPU2006 results (under 1% accuracy). This is because IDA Pro comes with a set of code signatures designed to recognize standard library functions that are statically linked into binaries.

For the ICFG, we see the same pattern as for instructions: all disassemblers perform worse than for application code, while IDA Pro delivers comparable accuracy. Callgraph accuracy is below the mean for most disassemblers, though IDA Pro and Dyninst perform very close to the mean, and BAP well exceeds it.

6.3.2.2 Complex Constructs

Overall, we found the `glibc-2.22` code to be surprisingly well-behaved. Our analysis found no overlapping or shared basic blocks, and no inline data. Indeed, the `glibc` developers have taken special care to prevent this, explicitly placing data and jump tables in the `.rodata` section even when manually declared in handwritten assembly code. Prior work has analysed earlier versions of `glibc`, showing that inline jump tables *are* present in `glibc-2.12` [125]. Moreover, inline zero-bytes used for function padding are confirmed in versions up to 2.21. This is worth noting, as older `glibc` versions may still be encountered in practice. Our analysis of `glibc` versions ranging from 2.12 to 2.22 shows consistently improving disassembler-friendliness over time.

We did find some complex constructs that do not occur in application code, the most notable being overlapping instructions. We found 31 such instructions in `glibc`. All of these are instructions with optional prefixes, such as the one shown in Listing 6.6. These overlapping instructions are defined manually in handcrafted assembly code, and typically use a conditional jump to optionally skip a `lock` prefix. They correspond to frequently cited complex cases in the literature [31; 125].

	Instructions	Functions	Signatures	ICFG	Callgraph
gcc-5.1.1 x64 with -static					
SPEC/C `O0`	96.2	69.4	0.1	98.3	98.2
SPEC/C `O1`	96.2	68.4	0.2	98.6	98.4
SPEC/C `O2`	95.5	67.1	0.2	98.8	98.9
SPEC/C `O3`	95.6	65.7	0.2	98.7	98.7
SPEC/C `Os`	95.9	67.8	0.2	98.7	98.4
gcc-5.1.1 x64 with -static and -flto					
SPEC/C `O0`	96.3	69.3	0.2	98.5	98.3
SPEC/C `O1`	96.0	68.6	0.3	98.6	98.4
SPEC/C `O2`	95.0	67.4	0.3	98.3	98.0
SPEC/C `O3`	95.2	66.9	0.3	98.3	98.4
SPEC/C `Os`	95.5	67.8	0.2	98.4	97.7

Table 6.3: IDA Pro 6.7 disassembly results for static and link-time optimized SPEC C benchmarks (% correct, geometric mean).

In addition, we found 508 tailcalls resulting from the compiler's normal optimization; a number comparable to application binaries of similar size as `glibc`. We also found significantly more multi-entry functions than in the SPEC benchmarks. Most of these belong to the `_nocancel` family, explicitly defined in `glibc`, an example of which is shown in Listing 6.7. These functions provide optional basic blocks which can be prefixed to the main function body to choose a threadsafe variant of the function. These prefix blocks end by jumping over the start of the main function body, a pattern also sometimes seen in application code.

Given that all non-standard complex constructs in `glibc` are due to handwritten assembly, we manually analyzed all assembly code in `libc++` and `libstdc++`. However, the amount of assembly in these libraries is very limited and revealed no new complex constructs. This suggests that the optimization constructs in `glibc` are typical for low-level libraries, and less common in higher-level ones such as the C++ standard libraries.

6.3.3 Static Linking & Link-time Optimization

Static linking can reduce disassembler performance on application binaries by merging complex library code into the binary. Link-time optimization performs intermodular optimization at link-time, as opposed to more local compile-time optimizations. It is a relatively new feature that is gaining in popularity, and could worsen disassembler performance if combined with static linking, by optimizing application and library code as a whole. To study the effects of these options, we recompiled the SPEC CPU2006 C benchmarks, statically linking them with `glibc-2.22` using `gcc`'s `-static` flag. Subsequently, we repeated the process with both static linking and link-time optimization (`gcc`'s `-flto`) enabled.

As expected, static linking merges complex cases from `glibc` into SPEC, including overlapping instructions. The effect on disassembly performance is shown

in Table 6.3 for IDA, the overall best performing disassembler in our `glibc` tests. The impact is slight but noticeable, with an instruction accuracy drop of up to 3 percentage points compared to baseline SPEC; about the same as for `glibc`. As can be seen in Table 6.3, link-time optimization does not significantly decrease disassembly accuracy compared to static linking only.

Function start detection suffers from static linking mostly at lower optimization levels, dropping from a mean of 80% to just under 70% for `O0`; at level `O3` the performance is not significantly reduced. Again, link-time optimization does not worsen the situation compared to pure static linking. For the ICFG and callgraph tests, a small accuracy drop is again seen at lower optimization levels, again with no more adverse effects due to link-time optimization. For instance, ICFG accuracy drops from close to 100% mean in baseline SPEC to just over 98% in statically linked SPEC at `O0`, while the results at `O2` and `O3` show no negative impact. We suspect that this is a result of optimized library code being linked in even at lower optimization levels. Overall, we do not expect any significant adverse impact on binary-based research as link-time optimization gains in popularity.

6.4 Implications of Results

This section discusses the implications of our results for three popular directions in binary-based research, namely: (1) Control-Flow Integrity (CFI), (2) Decompilation, and (3) Automatic bug search. A detailed comparison of our results to assumptions in the literature is given in Section 6.5.

6.4.1 Control-Flow Integrity

Control-Flow Integrity (CFI) is currently one of the most popular research directions in systems security, as shown in Table 6.5. Binary-level CFI typically relies on binary instrumentation to insert control flow protections [23; 75; 126; 144; 185; 196; 197; 199]. Though a wide variety of CFI solutions has been proposed, most of these have similar binary analysis requirements, due to their common aim of protecting indirect jumps, indirect calls, and returns. We structure our discussion around what is needed to analyze and protect each of these control edge types.

Indirect calls Typically, protecting an indirect call requires instrumenting both the call site (the `call` instruction itself, possibly including parameters), and the call target (the called function). Finding call sites relies mainly on accurate and complete disassembly of the basic instructions. As shown in Figure 6.1a, these can be recovered with extremely high accuracy, even 100% accuracy for linear disassembly on `gcc` and `clang` binaries. Thus, a binary-level CFI solution is unlikely to encounter problems analyzing and instrumenting call sites.

For Visual Studio binaries, there is a chance that a small percentage of call sites may be missed. Depending on the specific CFI solution, it may be possible to detect

calls from uninstrumented sites in the target function, triggering a runtime error handling mechanism (see Section 6.5). Since these cases are rare, it is then feasible to perform more elaborate (slow path) alternative security checks.

The main challenge is to accurately detect all possible target functions for each indirect call. As a basic prerequisite, this requires finding the complete set of indirectly called functions. As shown in Section 6.3.1.1 and Figure 6.1b, this is one of the most challenging problems in disassembly—at high optimization levels, 20% or more of all functions are routinely missed.

Moreover, fine-grained CFI systems must perform even more elaborate analysis to decide which functions are legal targets for each indirect call site. Overestimating the set of legal targets leads to attacks which redirect indirect calls to unexpected functions [88]. Matching call sites to a set of targets typically requires an accurate (I)CFG, so that control-flow and data-flow analysis can be performed to determine which function pointers are passed to each call site. Figure 6.1d and Sections 6.3.1.1–6.3.1.3 show that an accurate and complete ICFG is typically available, including accurate resolution of switch/jump tables in the best disassemblers. Although this type of analysis remains extremely challenging, especially if done interprocedurally (requiring accurate indirect call resolution), it is at least not limited by the accuracy of basic blocks or direct control edges.

Additionally, fine-grained CFI systems can benefit from function signature information, to further narrow down the set of targets per call site by matching the function prototype to parameters passed at the call site [181]. Though signature information is often far from complete (Figure 6.1c), especially on x64, the information which is available can still be useful—even with incomplete information, the target set can be reduced, directly leading to security improvements. However, care must be taken to make the analysis as conservative as possible; if this is not done, the inaccuracy of function signature information can easily cause illegal function calls to be allowed, or worse, can cause legal calls to be inadvertently blocked.

Indirect jumps Protecting indirect jumps requires analysis similar to the requirements for indirect calls. However, as indirect jumps are typically intraprocedural, protecting them usually does not rely on function detection. Instead, accurate switch/jump table resolution is required, which is available in disassemblers like IDA Pro (Section 6.3.1.3).

Return instructions Return instructions are typically protected using a shadow stack (as discussed in Chapter 3), which requires instrumenting all call and return sites (and jumps, to handle tailcalls) [54]. Given the accurate instruction recovery possible with modern disassemblers (Figure 6.1a), it is possible to accurately and completely instrument these sites.

Summarizing, the main challenge for modern CFI lies in accurately and completely protecting indirect call sites. The reasons for this are twofold: (1) Function detection is one of the most inaccurate primitives (especially for indirectly called

functions), even in state of the art disassemblers, and (2) It is currently very difficult to recover rich information, such as function signature information, through disassembly. This makes it extremely challenging to accurately couple indirect call sites with valid targets.

6.4.2 Decompilation

Instead of translating a binary into assembly instructions, decompilers lift binaries to a higher-level language, typically (pseudo-)C. Decompilers are typically built on top of a disassembler, relying heavily on the quality of the disassembly [160; 191].

As most decompilers operate at function granularity, they rely on accurate function start information. Moreover, they must translate all basic blocks belonging to a function, requiring knowledge of the function's CFG. In effect, this requires not only accurate function start detection, but accurate function boundary detection. As described in Chapter 7, function boundary detection is even more challenging than function start detection, as it additionally requires locating the end address of each function. This is difficult, especially in optimized binaries, where tailcalls often blur the boundaries between functions (since the `jmp` instructions used in tailcalls can easily be mistaken for intraprocedural control transfers).

In addition to function detection, decompilers rely on accurate instruction disassembly, and can also greatly benefit from function signature/type information. Moreover, switch detection is required to correctly attribute all switch case blocks to their parent function. Finally, callgraph information is useful to understand the connections between decompiled functions.

The impact of inaccuracies for decompilation is not as severe as for CFI systems, since decompiled code is typically intended for use in manual reverse engineering rather than automated analysis. However, disassembly errors can still affect the decompilation process itself, especially in later passes (such as stack frame analysis or data type analysis passes) over the raw decompiled function. Such analysis phases, as well as human reverse engineers, must take into account the high probability of errors in function boundary and signature information.

6.4.3 Automatic Bug Search

The binary analysis requirements of automatic bug search systems depend on the type of bug being searched for, and the granularity of the search. In practice, many such systems operate at the function level, both for ease of analysis, and because it is a suitable search-granularity for common bugs, such as stack-based bugs [98; 140; 201]. Operating at the function level is also useful for interoperability with other binary analysis primitives, such as symbolic execution, which are powerful tools for semantic analysis but do not scale to full binaries [98].

Thus, like decompilation, many automatic bug search systems rely on accurate function boundary information as well as per-function CFGs. Fortunately, despite

	# Papers	Instructions	Functions	Signatures	CFG	Callgraph
angr	0	0	0	0	0	0
BAP	2	1	2	1	2	0
ByteWeight	0	0	0	0	0	0
Dyninst	1	1	0	0	1	1
Hopper	0	0	0	0	0	0
IDA Pro	13	11	6	2	11	4
Jakstab	0	0	0	0	0	0
PSI/BinCFI	4	3	3	0	3	2
Linear	2	2	1	0	1	1
Other/Custom	8	7	2	0	6	3
Total	30	25	14	3	24	11

Table 6.4: Primitives/disassemblers used in the literature.

the relatively large inaccuracies in the input information, the output of bug detection systems tends to degrade gracefully: input inaccuracies may lead to bugs being missed, but typically do not affect the correctness of the analysis for other parts of the code. Quantifying the accuracy of the inputs (disassembly, CFG, function boundaries), as is our goal in this chapter, helps users to determine the expected output completeness of automatic bug search systems.

6.5 Disassembly in the Literature

Given our disassembly results, we studied recent binary-based research to determine how well the capabilities of disassemblers match the expectations in the literature. Our study covers research published between 2013 and 2015 in all top-tier systems security conferences, namely S&P (Oakland), CCS, NDSS and USENIX Security. In addition, we cover research published in these same years at RAID and ACSAC, two other major conferences which are popular targets for such research.

We found 30 papers on binary-based research published in these venues, summarized in Table 6.5. The rest of this section presents aggregated findings to provide a degree of anonymization for these papers.

Table 6.4 shows the primitives and disassemblers used in these papers. IDA Pro is by far the most popular, for all primitives; our disassembly results (Section 6.3) justify this choice. Despite its good accuracy, linear disassembly is among the least used, even for papers that handle only ELF binaries. This may result from the widespread belief that inline data causes far more problems than we found.

Instructions are the most often needed primitive, used by 25 of the 30 papers. It is followed by the CFG (24 papers) and function starts (14 papers). Function signature information is needed by only 3 of the analyzed papers. One paper used linear disassembly as a basis for building a CFG and callgraph, and scanning for function starts.

Title	First author	Venue	Year	Top-Tier
A Principled Approach for ROP Defense [146]	Qiao	ACSAC	2015	
Binary Code Continent: Finer-Grained Control Flow Integrity (...) [185]	Wang	ACSAC	2015	
Blanket Execution: Dynamic Similarity Testing for Program (...) [84]	Egele	Sec.	2014	✓
BYTEWEIGHT: Learning to Recognize Functions in Binary Code [26]	Bao	Sec.	2014	✓
CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying (...) [37]	Bonfante	CCS	2015	✓
Control Flow and Code Integrity for COTS binaries [200]	Zhang	ACSAC	2015	
Control Flow Integrity for COTS Binaries [199]	Zhang	Sec.	2013	✓
Cross-Architecture Bug Search in Binary Executables [140]	Pewny	S&P	2015	✓
DUET: Integration of Dynamic and Static Analyses for Malware (...) [99]	Hu	ACSAC	2013	
Dynamic Hooks: Hiding Control Flow Changes within (...) [183]	Vogl	Sec.	2014	✓
Hardware-Assisted Fine-Grained Control-Flow Integrity (...) [75]	Davi	RAID	2015	
Heisenbyte: Thwarting Memory Disclosure Attacks using (...) [176]	Tang	CCS	2015	✓
High Accuracy Attack Provenance via Binary-based (...) [115]	Hyung Lee	NDSS	2013	✓
Improving Accuracy of Static Integer Overflow Detection in Binary [201]	Zhang	RAID	2015	
Leveraging Semantic Signatures for Bug Search in Binary Programs [141]	Pewny	ACSAC	2014	
Native x86 Decompilation Using Semantics-Preserving (...) [160]	Schwartz	Sec.	2013	✓
No More Gotos: Decompilation Using Pattern-Independent (...) [191]	Yakdan	NDSS	2015	✓
Opaque Control-Flow Integrity [126]	Mohan	NDSS	2015	✓
Oxymoron Making Fine-Grained Memory Randomization Practical (...) [24]	Backes	Sec.	2014	✓
Practical Context-Sensitive CFI [23]	Andriesse	CCS	2015	✓
Practical Control Flow Integrity & Randomization for (...) [197]	Zhang	S&P	2013	✓
Reassembleable Disassembling [186]	Wang	Sec.	2015	✓
Recognizing Functions in Binaries with Neural Networks [164]	Chul	Sec.	2015	✓
ROPecker: A Generic and Practical Approach for Defending (...) [56]	Cheng	NDSS	2014	✓
StackArmor: Comprehensive Protection from Stack-based (...) [54]	Chen	NDSS	2015	✓
Towards Automated Integrity Protection of C++ Virtual Function (...) [92]	Gawlik	ACSAC	2014	
Towards Automatic Software Lineage Inference [103]	Jang	Sec.	2013	✓
vfGuard: Strict Protection for Virtual Function Calls (...) [144]	Prakash	NDSS	2015	✓
VTint: Protecting Virtual Function Tables' Integrity [196]	Zhang	NDSS	2015	✓
X-Force: Force-Executing Binary Programs for Security (...) [139]	Peng	Sec.	2014	✓

Table 6.5: Set of papers discussed in the literature study.

Table 6.6 provides more details on the properties of the papers we analyzed. We distinguish between papers that target Windows PE binaries, and those that target Linux ELF. This is because some complex cases, such as inline data, are more often generated by Visual Studio, deserving closer attention in Windows papers.

Most papers that support obfuscated binaries target Windows (33% of papers versus 10% for Linux). This is because obfuscation typically occurs in malware, which is more prevalent on Windows. Though we do not consider obfuscated binaries in our tests, it is still interesting to know how many papers target such binaries. After all, these papers should pay special attention to disassembly errors and complex corner cases. Unfortunately, this is not the case; only 50% of papers that support obfuscation discuss potential errors, while 33% implement error handling. This is no better than the overall number. Moreover, only 17% of these papers explicitly discuss complex cases; far below the overall rate for Windows.

Nearly all papers support optimized binaries (90% or more for both Linux and Windows, overall as well as top-tier). Stripped binaries are supported by an equally large majority of papers on Windows, and by a slightly smaller majority on Linux. Curiously, the number of top-tier papers that support stripped binaries on Linux (70%) is significantly less than the overall number (79%).

Property	Subproperty	All papers #	All papers %	Top-tier #	Top-tier %
Windows PE x86/x64 (16 papers, 12 top-tier)					
Obfuscated code		5	31%	4	33%
Optimized binaries		14	88%	11	92%
Stripped binaries		15	94%	11	92%
Recursive disassembly		16	100%	12	100%
Needs relocation info		2	12%	2	17%
Primitive errors discussed	Instructions	5 (13)	38%	5 (9)	56%
	Functions	1 (5)	20%	1 (4)	25%
	Signatures	0 (2)	0%	0 (2)	0%
	Callgraph	4 (5)	80%	4 (5)	80%
	CFG	5 (13)	38%	5 (10)	50%
Complex cases discussed		5	31%	5	42%
Primitive errors handled	Overestimate	4	25%	4	33%
	Underestimate	3	19%	2	17%
	Runtime	1	6%	1	8%
Errors are fatal		13	81%	11	92%
Linux ELF x86/x64 (14 papers, 10 top-tier)					
Obfuscated code		1	7%	1	10%
Optimized binaries		13	93%	9	90%
Stripped binaries		11	79%	7	70%
Recursive disassembly		12	86%	8	80%
Primitive errors discussed	Instructions	6 (12)	50%	6 (9)	67%
	Functions	3 (9)	33%	3 (6)	50%
	Signatures	1 (1)	100%	1 (1)	100%
	Callgraph	2 (6)	33%	2 (4)	50%
	CFG	5 (11)	45%	5 (8)	62%
Complex cases discussed		1	7%	1	10%
Primitive errors handled	Overestimate	4	29%	3	30%
	Underestimate	0	0%	0	0%
	Runtime	1	7%	1	10%
Errors are fatal		8	57%	6	60%

Table 6.6: Properties of binary-based papers (number and percentage of papers). Numbers in parentheses indicate the total number of papers that use this primitive.

The vast majority of papers use recursive disassembly (100% on Windows and 86% on Linux), with IDA Pro being the most popular disassembler. The few papers that do use linear disassembly are based on objdump, and augment it with a layer of error correction. Interestingly, these papers claim perfect (100% accurate) or close to perfect disassembly. As shown in Section 6.3.1.1, this precision on Linux binaries owes entirely to the core linear disassembly, making any error correction redundant other than for a few corner cases in library code (and obfuscated code, which these papers do not consider).

A relatively small percentage of Windows papers use relocation information to find disassembly starting points. At 17%, this number is slightly higher for top-tier papers than overall.

Discussion on disassembly errors and complex cases is somewhat lacking in the analyzed papers. For most primitives on Windows, at best 50% of papers discuss what happens if the primitive is not recovered perfectly. This number applies to the top-tier papers; overall, the number is even lower. The number for Linux-based papers is slightly better, though even here only a small majority of papers devote

significant attention to potential problems. One would expect more thorough discussion, especially given that between 80% and 90% of Windows papers, and around 60% of Linux papers, may suffer malignant failures given imperfect primitives. The issue is most apparent in the Windows papers that require function start information. Only 25% of the top-tier papers that require function starts consider potential errors in this information, even though Section 6.3.1.1 shows that function starts are quite challenging to recover accurately.

The percentage of Windows papers that discuss complex cases such as inline data varies from 31% overall to 42% for top-tier papers. Again, this is less than we would expect given the prevalence of inline jump tables generated by Visual Studio. The number for papers that target Linux is even lower, though this causes fewer issues as complex cases in ELF binaries are rare.

There is a strong correlation within all papers between discussion of errors and complex cases, and support for error handling. Papers that discuss such cases also tend to implement some mechanism for dealing with errors if they occur. Conversely, papers that do not implement error handling nearly always fail to discuss potential errors at all.

We identified three popular and recurring categories of error handling mechanisms used in the literature (discussed in more detail in the Discussion at the end of Part I of this thesis).

(1) *Overestimation*: For instance, CFG and callgraph overestimation are popular in papers that build binary-level security; it minimizes the risk of accidentally prohibiting valid edges, though the precision of security policies may suffer slightly.

(2) *Underestimation*: This is used in papers where soundness is more important than completeness.

(3) *Runtime augmentation*: Some papers use static analysis to approximate a primitive, and use low-cost runtime checks to fix errors in the primitive where needed.

Overestimation is the most popular error handling strategy, used in around 30% of top-tier papers. It is followed by underestimation and runtime augmentation.

6.6 Discussion

Our findings show a dualism in the stance on disassembly in the literature. On the one hand, the difficulty of pure (instruction-level) disassembly is often exaggerated. The prevalence of complex constructs like overlapping basic blocks, inline data, and overlapping instructions is frequently overestimated, especially for `gcc` and `clang` [31; 125]. This leads reviewers and researchers to underestimate the effectiveness of binary-based research.

We showed that unless binaries are deliberately obfuscated, instruction recovery is extremely accurate, especially in ELF binaries generated with `gcc` or `clang`. We did not find any inline data for these binaries, even in optimized library code; even jump tables are explicitly placed in the `.rodata` section. Moreover, in Visual

Studio binaries with jump tables in the code section, modern disassemblers like IDA Pro recognize and resolve them quite accurately. The rare overlapping instructions in handcrafted library code take on a limited number of forms, typically using a direct conditional jump over a prefix. These are resolved without problems by IDA Pro and Dyninst, among others. The same is true for multi-entry functions, which are also rare. Moreover, overlapping/shared basic blocks (commonly cited as particularly challenging for binary analysis), do not appear in our findings at all.

On the other hand, some primitives really do often suffer from inaccuracies. Some recursive disassemblers used for binary instrumentation (notably Dyninst) regularly miss up to 10% of basic blocks in optimized binaries, calling for special attention in systems which rely on basic block-level binary instrumentation. Additionally, function signatures in 64-bit code are extremely inaccurate; fortunately, they are also rarely used in the literature.

However, function starts *are* regularly needed, though the false negative rate regularly rises to 20% or more even for the best performing disassemblers. This is especially true in optimized binaries, or in coding styles that make extensive use of function pointers. Worse, false positive function starts are almost as common. This can lead to problems in some binary-based research, especially binary instrumentation, if care is not taken to ensure graceful failure in the event of misdetected function starts. Symbols offer a great deal of help, especially in reducing the false negative rate. Unfortunately, they are rarely available in practice.

It is surprising then, to find that only 20% to 25% (top-tier) of Windows papers that use function starts, and 33% to 50% (top-tier) of the Linux papers, devote any attention to discussing these problems. A similarly small number of papers implement error handling, even though errors can cause malignant failures in a majority of papers. While it is not impossible to base well-functioning binary-based systems on function start information (or other primitives), it is crucial that such work implement mechanisms for handling inaccuracies. Three effective classes of error handling (depending on the situation) are already used in the literature: overestimation, underestimation, and runtime augmentation. We provide a detailed overview of error handling strategies in the Discussion at the end of Part I of this thesis.

We hope our study will facilitate a better match between expectations on disassembly in future research, and the performance actually delivered by modern disassemblers. We believe our findings can be used to better judge where problems are to be expected, and to implement effective mechanisms for dealing with them.

6.7 Related Work

Prior work on disassembly precision focused on complex corner cases [31; 125; 135] or obfuscated code [110; 161], showing that these can strongly reduce disassembly accuracy. We focus instead on the performance of modern disassemblers given realistic full-scale binaries without active anti-disassembly techniques.

Miller et al. center their analysis around complex cases in `glibc-2.12` [125]. Their findings largely correspond to our own, though we found no inline jump tables in `glibc-2.22`. In addition to their `glibc` analysis, Miller et al. find complex cases in SPEC CPU2006; however, this analysis focuses exclusively on statically linked binaries. We show in Section 6.3.3 that these cases are entirely due to embedded library code, and are extremely rare in non-statically linked applications.

Our finding that function starts are among the most challenging primitives to recover is in agreement with results by Bao et al. [26].

Paleari et al. study instruction decoders used in disassemblers [135], which parse individual x86 instructions. Specific instructions that are sometimes wrongly parsed have also been outlined by the authors of Capstone [147].

Complex constructs in obfuscated code are discussed by Schwarz et al. [161], Linn et al. [118] and Kruegel et al. [110]. We show that these worst-case complex constructs are exceedingly rare in non-obfuscated code.

6.8 Conclusion

Our study contradicts the widespread belief that complex constructs severely limit the usefulness of binary-based research. In contrast, we show that modern disassemblers achieve close to 100% instruction disassembly accuracy for compiler-generated binaries, and that constructs like inline data and overlapping code are very rare. Errors in areas where disassembly is truly lacking, such as function start recovery, are not discussed nearly as often in the literature. By analyzing discrepancies between disassembler capabilities and the literature, our work provides a foundation for guiding future research.

Chapter 7

Compiler-Agnostic Function Detection in Binaries

We propose *Nucleus*, a novel function detection algorithm for binaries. In contrast to existing approaches, *Nucleus* is compiler-agnostic, and does not require any learning phase or signature information. Instead of scanning for signatures, *Nucleus* detects functions at the Control Flow Graph-level, making it inherently suitable for difficult cases such as non-contiguous or multi-entry functions. We evaluate *Nucleus* on a diverse set of 476 C and C++ binaries, compiled with `gcc`, `clang` and Visual Studio for x86 and x64, at optimization levels O0–O3. We achieve consistently good performance, with a mean F-score of 0.95.

7.1 Introduction

Function detection is a binary analysis technique that categorizes the code within a binary into functions approximating the original (source-level) functions. It is a key building block in areas like binary instrumentation [31; 114], binary-level vulnerability search [86; 140], and binary protection schemes, including Control-Flow Integrity [23; 54; 144; 197]. Moreover, accurate function detection is crucial for human reverse engineers, who rely on such compartmentalization to aid their reasoning about complex binary code.

We have shown in Chapter 6 that while modern disassemblers and binary analysis platforms achieve high accuracy in terms of instruction recovery, their function detection capabilities are still lacking [22]. For instance, for stripped x64 ELF binaries generated with the common `gcc` compiler, our results show that the prominent IDA Pro disassembler misidentifies 25% to 40% (depending on optimization level) of functions *on average*, and up to 75% in the worst case. Moreover, up to 20% of the reported functions are false positives. Other disassemblers, such as Dyninst [31] and BAP [44], deliver comparable or worse performance.

The predominant approach to the function detection problem is to use a signature database to scan binaries for known function prologues and epilogues. This approach is used even in state-of-the-art work like ByteWeight, which uses machine learning to automatically generate signatures [26; 164]. While signature-based function detection can achieve reasonable accuracy for unoptimized binaries, its performance declines steeply for highly optimized binaries, where standard function prologues are often missing altogether. Moreover, signature databases require constant maintenance, to support new compilers and compiler versions.

This chapter proposes a new signature-less approach to function detection for stripped binaries, based on *structural Control Flow Graph analysis*. We provide an open-source implementation of our approach, called *Nucleus*.[1] Rather than scanning binaries for signatures, *Nucleus* is centered around an Interprocedural Control Flow Graph (ICFG), which it constructs by disassembling a binary and analyzing its control flow. *Nucleus* identifies functions in the ICFG by analyzing the flows between basic blocks, based on our observation that intraprocedural control flow tends to use different types and patterns of control flow instructions than interprocedural control flow. We show that this property holds across different compilers and optimization levels, allowing *Nucleus* to identify functions in a completely compiler-agnostic way, without any compiler-specific signatures or heuristics. *Nucleus* also inherently supports difficult cases like non-contiguous and multi-entry functions. *Nucleus* can export its results directly to the popular IDA Pro disassembler, making it easy to use in real-world scenarios.

We evaluate *Nucleus* on a diverse set of 476 binaries, which includes binaries compiled with `gcc`, `clang` and Visual Studio for both Linux (ELF) and Windows (PE). Our evaluation covers both C and C++ code, compiled for x86 (32-bit) and x64 (64-bit), at optimization levels ranging from O0 to O3. *Nucleus* achieves mean precision and recall rates of 0.96 and 0.94, respectively; consistently outperforming IDA Pro and Dyninst, and matching the reported accuracy of state-of-the-art machine learning-based work [26; 164].

Further, our evaluation reveals a significant discrepancy between the accuracy reported for these machine learning approaches (specifically ByteWeight [26]), and the results they deliver in our tests. Upon closer analysis, we find a large overlap between the training set and test set used to evaluate *all* top-tier work on machine learning for function detection, including ByteWeight [26; 164]. We show that this leads to a large bias in the evaluations for these papers, underlining the need for future work to reassess the viability of machine learning for function detection.

Contributions Our contributions in this chapter are as follows.

- We introduce *Nucleus*, a novel compiler-agnostic function detection engine, and show that it achieves high accuracy for all major compilers and platforms,

[1] Source available at https://www.vusec.net/projects/function-detection.

without requiring any of the compiler-specific signatures used by current state-of-the-art algorithms. *Nucleus* is open source, and is easy to use in real-world environments due to its ability to integrate with IDA Pro, the *de facto* industry-standard disassembler.

- In contrast to prior work, *Nucleus* can support new compilers without any training or maintenance. Moreover, *Nucleus* provides inherent support for difficult cases, such as non-contiguous and multi-entry functions, without assuming anything about the memory or instruction layout of functions.

- We find a strong bias in the evaluations of top-tier work on machine learning-based function detection, demonstrating that these techniques need to be re-assessed before the accuracy reported in their evaluations can be assumed to generalize to other data sets.

7.2 Background

This section provides a brief introduction to function detection. We discuss the definition and scope of the function detection problem, as well as challenging cases which need to be handled.

7.2.1 Definition of Function Detection

Function detection comprises two main problems: *function start detection*, and *function boundary detection*. In function start detection, the aim is to find all addresses in a binary that correspond to a function entry point, while function boundary detection attempts to find both the first and last address of each function. Our definitions of these are analogous to the definitions by Bao et al. [26].

We use these definitions to compare *Nucleus* to existing approaches in our evaluation (Section 7.5). However, *Nucleus* is not limited to detecting only function start and end addresses; as discussed in Section 7.3, *Nucleus* assigns *all* basic blocks to their containing functions.

Function start detection Given a binary P compiled from a set of source-level functions $F := \{f_1, f_2, \ldots, f_m\}$, identify a set of addresses $S := \{s_1, s_2, \ldots, s_n\}$ in P such that s_i points to the machine instruction corresponding to the first line (*entry point*) of some $f_j \in F$. Note that for stripped binaries, F is *not* known to the function detector. Given a set of ground truth start addresses S_{gt}, we define the set of true positives as $TP := S \cap S_{gt}$, false positives as $FP := S \setminus S_{gt}$ and false negatives as $FN := S_{gt} \setminus S$.

Function boundary detection Given the same binary P compiled from functions in F, identify a set of address pairs $B := \{(s_1, e_1), (s_2, e_2), \ldots, (s_n, e_n)\}$ in P

such that s_i is the function start address of $f_j \in F$ and e_i is the last address in P corresponding to a line from f_j. Given again a set of ground truth function boundaries B_{gt}, we define the set of true positives as $TP := B \cap B_{gt}$, false positives as $FP := \{(s,e) \mid (s,e) \in B \wedge s \notin S_{gt}\}$, and false negatives as $FN := B_{gt} \setminus B$. Note that this implies that for TP, both the function start and end address must be correct; if either is incorrect this counts for FN.

7.2.2 Scope of Function Detection

For binaries with symbolic information, function detection is trivial—the symbol table specifies the set of functions, along with their names, start addresses, and sizes. Unfortunately, many binaries in practice are stripped of this information. This makes function detection far more challenging—source-level functions have no real meaning at the binary level, and their boundaries are frequently blurred by compiler optimizations. *Nucleus*, like other work on function detection [26; 31; 83; 164], focuses on stripped binaries.

Though challenging, function detection in stripped binaries is important in virtually all forms of binary reverse engineering. Human reverse engineers often deal with stripped binaries, especially in malware analysis or security auditing of untrusted binaries [83; 151]. Decompilers attempt to facilitate human reverse engineering by deriving a high-level code representation from binaries, also operating at the function level [160; 191].

Automated reverse engineering and binary instrumentation systems also rely on accurate function detection for stripped binaries, such as legacy binaries or binaries for embedded systems (which are often stripped to save memory). For instance, Control Flow Integrity mechanisms often reason about security at the function level [23; 144; 197]. Moreover, automated bug detection systems [86; 140] and binary-level reoptimizers also commonly reason at the function level [108].

7.2.3 Signature-Based Approaches

The predominant strategy for function detection is based on signatures. This strategy is used in all well-known approaches, including IDA Pro [83], Dyninst [31] and machine learning approaches like ByteWeight [26; 164].

Typically, signature-based function detection algorithms start with a pass over the disassembled binary to locate trivial functions that are directly addressed by a `call` instruction. To locate the remaining functions (such as indirectly called or tailcalled functions), these approaches scan for well-known signatures that indicate function prologues and epilogues. For instance, a typical pattern that many x86 compilers emit for unoptimized functions starts with the prologue `push ebp; mov ebp,esp`, and ends with the epilogue `leave; ret`. In practice, many patterns are used, depending on the platform, compiler, and optimization level. Indeed, optimized functions may not have well-known function prologues or epilogues at all.

This wide variety of function patterns and calling conventions is a major problem for the scalability of signature-based function detection. Signature databases need to account for all these possibilities, and need constant maintenance to account for new platforms, compilers and compiler versions. Recent work by Bao et al. [26] and Shin et al. [164] has focused on automating the process of learning new function signatures. However, these approaches still require signatures tuned for specific compilers and an expensive learning phase for every configuration change. The scalability problems are especially apparent for open-source projects like GNU `gcc` and `llvm`/`clang`, which release new major versions roughly every six months, and minor versions with even higher frequency.[2,3]

7.2.4 Challenging Cases

We distinguish several constructs which are challenging for function detection. Typically, these result from compiler optimizations.

We consider the following cases (introduced in Chapter 2.7): (1) Non-contiguous functions, (2) Multi-entry functions, (3) Padding code and inline data, (4) Unreachable code, (5) Tail calls, and (6) Alternative prologues/epilogues. *Nucleus* naturally handles most of these cases, though tail calls require a degree of dedicated handling (as we discuss in Section 7.5.3).

Sections 7.3–7.7 provide real-world examples of complex cases, and discuss how *Nucleus* handles them. We also provide a detailed discussion of cases which are not handled by *Nucleus* in Section 7.5.3.

7.3 Nucleus Overview

This section provides a high-level overview of our function detection algorithm. Implementation details are provided in Section 7.4. The main steps of our algorithm are illustrated in Figure 7.1.

Though our approach is conceptually simple, we show in Section 7.5 that it is able to detect both function starts and function boundaries with very high accuracy. Moreover, our evaluation shows consistently good results across multiple instruction sets, compilers and platforms, without requiring any compiler-specific heuristics. Additionally, in contrast to signature-based approaches, our analysis yields the complete set of basic blocks belonging to each function, rather than only a start and end address.

7.3.1 ICFG Generation

We start by generating the Interprocedural Control Flow Graph (ICFG) around which the rest of our analysis is centered (step ① in Figure 7.1). The ICFG for a binary

[2] https://gcc.gnu.org/releases.html
[3] http://llvm.org/releases/

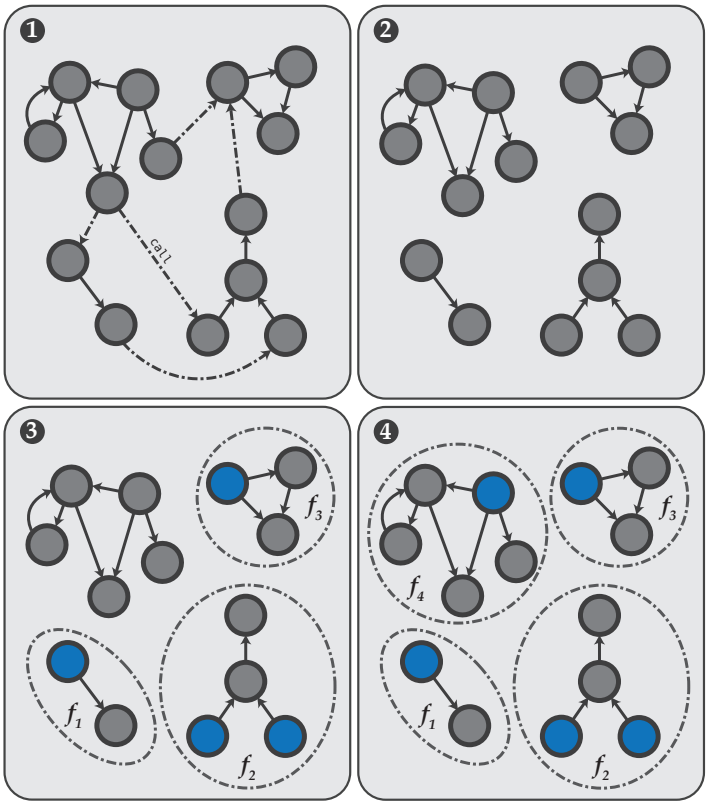


Figure 7.1: Overview of our function detection algorithm. ① Disassemble binary and generate ICFG. ② Hide edges $e \in E_{call}$. ③ Locate directly called entry points (shaded blue) and expand functions by following control flow (ignoring direction). ④ Find remaining functions through connected components analysis and detect entry points through intraprocedural control flow analysis.

B is a digraph $G = (V, E)$, where V is the set of all basic blocks in B, and E is the set of control flow edges $E \subseteq V \times V$ between basic blocks. E includes both intraprocedural and interprocedural edges (such as `call` edges). In contrast, the traditional definition of a (non-interprocedural) Control Flow Graph (CFG) is a function-level data structure that contains only the basic blocks and edges within a particular function. We operate on the ICFG because it can be generated without *a priori* knowledge of function boundaries. We generate the ICFG by disassembling the target binary, dividing it into basic blocks, and analyzing its control flow (see Section 7.4 for details).

To improve the accuracy of our analysis, we perform some preprocessing on the ICFG. Specifically, we use switch detection to resolve intraprocedural indirect jumps, and we use a combination of semantic analysis and reachability analysis to identify padding blocks and inline data. More details on our preprocessing algorithm are given in Section 7.4.

7.3.2 Connected Components Analysis

Next, we perform a weakly connected components analysis on the ICFG, temporarily excluding the `call` edges $e \in E_{call}$ (step ②). A weakly connected component is a subgraph of the ICFG, in which every two vertices are connected by an undirected path (i.e., ignoring the direction of control flow), and no vertex is connected to a vertex outside the component. By excluding `call` edges, the analysis finds all graph components that consist of basic blocks connected through only intraprocedural edges (some corner cases do exist; these are discussed in Section 7.5.3). In other words, we use the connected components analysis to find clusters of basic blocks belonging to the same function.

Note that this approach assumes nothing about the memory layout of functions, providing natural support for non-contiguous functions. Moreover, it does not require any kind of function prologue or epilogue detection, making our approach completely compiler agnostic. Our connected components analysis consists of several phases, as follows.

7.3.2.1 Directly Called Functions

First, we make a pass over the instructions of all basic blocks in the ICFG, scanning for direct `call` instructions. This allows us to detect the directly called function entry blocks, which we each expand into a complete function by following the edges from the entry block until a complete component is formed (step ③). This phase detects all components corresponding to directly called functions.

7.3.2.2 Unreachable/Indirectly Called Functions

Next, we find indirectly called or unreachable functions by iterating over all basic blocks in the ICFG, looking for BBs that are not yet part of a function (step ④). We expand each such block into a function using the aforementioned connected components analysis. Subsequently, we detect the function entry points by scanning the function for BBs that are not reached by any intraprocedural edge. (In practice, there may be loopback edges to an entry block; we describe our approach to dealing with these in Section 7.4.) We perform the same entry point analysis for the directly called functions, to detect possible multi-entry functions. If no suitable entry point can be found through other methods, our analysis assumes the function is entered at its lowest address (the default assumption in signature-based approaches).

7.4 Implementation

We implemented an open-source version of our function detection algorithm, called *Nucleus*, in 3278 C++ SLOC. The ICFG construction and function detection code consists of under 850 SLOC, while the remaining lines are attributed to our binary loader, disassembler, and utility code. We implemented a custom disassembly pass

```
mov   eax,eax
xchg  eax,eax
lea   eax,[eax + 0x0]
lea   eax,[eax + eiz*1 + 0x0]
```

Listing 7.1: Effective `nop` instructions emitted by `gcc` v5.1.1 on x86. Here, `eax` can be replaced by any general purpose register.

using Capstone v3.0.4 [147]. *Nucleus* provides the option to generate an IDA Python script that imports our function detection results directly into IDA Pro.

7.4.1 Disassembly and ICFG Generation

To find indirectly called and unreachable functions, we use a linear disassembly approach, coupled with an analysis to detect padding code and inline data. Recent work has shown that linear disassembly, even with only simple detection of padding or data, can reliably achieve high code coverage with few disassembly errors [22]. After disassembly completes, *Nucleus* constructs the ICFG by breaking the code into BBs, and creating the edges associated with each control flow instruction.

We then analyze each BB to see if it consists of do-nothing instructions used for padding. Simply checking for `nop` instructions is not enough, because not all compilers emit standard `nop` instructions. We therefore also check for instructions which move a source operand into a destination operand without modifying it. Listing 7.1 shows examples of this, used for padding by `gcc` v5.1.1 on x86.

Moreover, we use reachability analysis to determine if a `nop` block is part of a function (reachable via some control flow edge), or is padding (not reachable). We detect inline data by looking for BBs that contain invalid or privileged instructions. These blocks, and any BBs that can reach them via a jump or fallthrough edge, are marked as suspected data.

7.4.2 Switch Detection

Compilers typically implement switch statements as an indirect jump that selects its target from a *jump table* of code pointers, depending on which case should be executed. To correctly attribute all switch/case blocks to their associated function, we need to resolve these intraprocedural indirect jumps. *Nucleus* therefore implements a switch detection pass that performs a backward sweep starting from every indirect jump, looking for the instruction where the jump's target register is loaded. If this load instruction references a jump table, we scan this table for valid code pointers, adding these as targets of the indirect jump. More sophisticated switch detection is explored in related work [83; 106; 161], but is outside the scope of this work.

7.4.3 Function and Entry Point Detection

After ICFG generation is complete, we execute our connected components-based function detection algorithm as described in Section 7.3. As noted in Section 7.3,

we implement several ways of detecting function entry points (in order of priority): (1) by following direct `call` edges, (2) using intraprocedural control flow analysis, and (3) by assuming the function's lowest address as the entry point (as a last resort).

The intraprocedural control flow analysis detects function entry points by looking for basic blocks that are not reached by any intraprocedural edge. However, we must also deal with entry blocks which *do* have incoming loopback edges. Such entry blocks can be identified in two ways: (1) Loopback edges typically target not the start of an entry block, but jump to an offset within it (skipping past the function prologue). Because *Nucleus* tracks the destination offset of each edge, we can identify these cases. (2) Alternatively, we use intraprocedural loop detection to determine that the entry block is reached only via a loopback edge (while the source of the loopback edge is also reached by other inbound control flow edges).

7.5 Evaluation

In this section, we evaluate four key aspects of *Nucleus*. (1) How accurate is our function detection compared to existing work? (2) Does *Nucleus* achieve more stable cross-compiler/cross-architecture results than other approaches? (3) Which cases are handled well by *Nucleus*, and which cause false positives or false negatives? (4) How does the runtime performance of *Nucleus* compare to other approaches? We first describe our test setup, and then address these questions.

7.5.1 Test Setup

We evaluate *Nucleus* on a test suite consisting of 476 C and C++ binaries for x86 and x64—the most commonly targeted platforms in binary analysis research. Our test suite contains both Linux (ELF) and Windows (PE) binaries, compiled at optimization levels O0–O3. The ELF binaries are compiled with the popular `gcc` v5.1.1 and `clang` v3.7.0 compilers, while the PE binaries are compiled with Visual Studio 2015—these are the most recent versions at the time of our experiments. All of the binaries are stripped of any symbolic information.

Our test suite contains the SPEC CPU2006 C and C++ benchmarks, as well as the popular server applications `nginx` v1.8.0, `lighttpd` v1.4.39, `opensshd` v7.1p2, `vsftpd` v3.0.3 and `exim` v4.86. We choose this test suite for the following reasons: (1) It contains a diverse range of realistic C and C++ binaries, ranging from very small to large; (2) By testing with C and C++, as well x86 and x64 binaries, we cover a wide range of both stack-based and register-based calling conventions; (3) The tested binaries contain a wide variety of challenging cases—for instance, `perlbench` contains many indirect function calls; (4) SPEC CPU2006 compiles on both Linux and Windows, allowing a fair comparison between `gcc`, `clang`, and Visual Studio.

We obtain ground truth on function starts and function boundaries by compiling the ELF binaries with full symbolic and DWARF v3 information, and the PE binaries with full PDB (Program Database) files. After parsing the required function

information from these sources, we strip the binaries of all symbolic information before using them in our experiments.

We conduct our experiments on an Intel Core i5 4300U machine with 8GB of RAM, running Ubuntu 15.04. We compile our `gcc` and `clang` test cases on this same machine. The Visual Studio binaries are compiled on an Intel Core i7 3770 machine with 8GB of RAM, running Windows 10.

We compare *Nucleus* with *IDA Pro v6.7*, *Dyninst v9.1.0* [31], and *BAP v0.9.9* [44], which uses *ByteWeight v0.9.9* [26] to obtain function start information. We choose these tools because they are capable of delivering both function start and function boundary information, are widely used, and are also used as a reference in the evaluations of related work [26]. Moreover, in Section 7.6, we provide a more detailed comparison with the results yielded by state-of-the-art machine learning-based approaches, including ByteWeight [26] and Shin et al. [164].

7.5.2 Function Detection Results

We report our experimental results using the F-score metric, and the related notions of precision and recall. The F-score is widely used (also in related work [26; 164]) because it provides a combined metric of the true positive, false positive and false negative rates of a system. Precision is defined as $p = |TP| \,/\, (|TP| + |FP|)$, while recall is defined as $r = |TP| \,/\, (|TP| + |FN|)$. For us, $p = 1.0$ means that all reported functions are true positives (no false positives), while $r = 1.0$ means that there are no false negatives. The F-score is the harmonic mean of precision and recall: $F = 2 \cdot p \cdot r / (p + r)$. The range of the F-score is again $[0.0, 1.0]$, with $F = 1.0$ denoting perfect accuracy (no false positives or false negatives).

In our Linux-based testing environment, Dyninst was unable to process PE binaries. We therefore report only ELF results for Dyninst. As our server applications are Linux-specific, we test them only for `gcc` and `clang`.

7.5.2.1 Function Starts

We begin by discussing results for function start detection; results for function boundary detection are discussed in Section 7.5.2.2. Figure 7.2 shows the F-scores achieved by *Nucleus* and the other approaches per platform (x86 versus x64), compiler, and optimization level, differentiating between the C and C++ tests. For each case, the graph shows the geometric mean result achieved for SPEC CPU2006. Additionally, Table 7.1 shows the decomposition of the F-scores into precision and recall rates, for both the SPEC and server tests. For space reasons, Table 7.1 shows average scores taken over the geometric means for all optimization levels.

Figure 7.2 shows that *Nucleus* achieves accurate results across all compilers, platforms and optimization levels. We achieve an overall average F-score for SPEC of 0.96, ranging between 0.92 (O3) and a perfect F-score of 1.00 (O0) for the C tests, and between 0.87 and 0.99 for C++. As shown in Table 7.1, *Nucleus* consistently

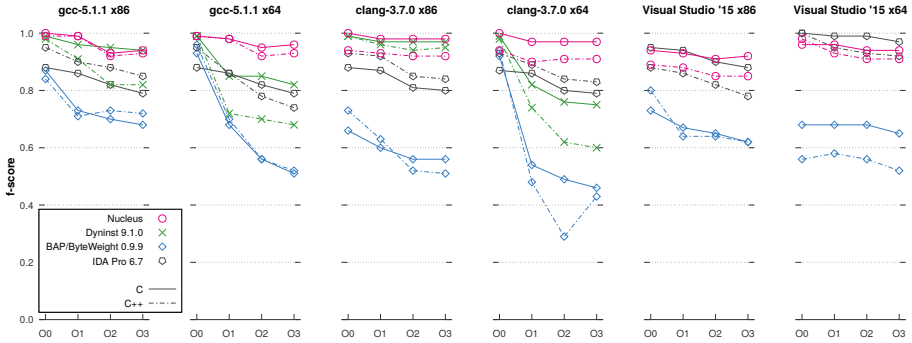


Figure 7.2: F-scores for *function start* detection (geometric mean for SPEC CPU2006).

outperforms all other disassemblers, especially in terms of recall, with the exception of Visual Studio on x64. Although *Nucleus* delivers accurate results for Visual Studio x64, IDA Pro is the most accurate for this compiler, with an average F-score of 0.97. This is due to the fact that for x64, Visual Studio 2015 uses only one calling convention [124], making IDA Pro's signature-based approach extremely effective. In contrast, other compilers use a variety of calling conventions.

As can be seen in Figure 7.2, *Nucleus* is more tolerant of varying compilers and optimization levels than other approaches, since *Nucleus* is compiler-agnostic. *Nucleus* shows stable accuracy across compilers and architectures, and the decrease in accuracy for high optimization levels is far less significant than for all other tested tools. In Section 7.5.3, we provide a detailed discussion of the more challenging cases which occur in optimized binaries.

The standard deviations in F-score for *Nucleus* are limited to the range $[0.01, 0.04]$ for C, and $[0.00, 0.07]$ for C++. In contrast, the ranges for IDA Pro are $[0.00, 0.13]$ for C (deviations below 0.11 only for Visual Studio), and $[0.00, 0.19]$ for C++. Dyninst ranges from $[0.01, 0.14]$ for C to $[0.01, 0.16]$ for C++, while BAP/ByteWeight ranges from $[0.04, 0.16]$ to $[0.02, 0.26]$, respectively. Again, this shows that *Nucleus* provides accurate results more consistently than signature-based approaches.

The results shown for ByteWeight are based on the ByteWeight version shipped with BAP v0.9.9, which we refer to as *BAP/ByteWeight v0.9.9* (BAP/BW for short). BAP uses ByteWeight to detect function starts, which it uses as entry points for disassembly. We tested BAP/Byteweight with the default ELF/PE signatures that are included with it. In our tests, this yielded significantly lower accuracy than the other approaches, with an overall mean F-score of only 0.65, which is 0.32 points lower than reported in the Byteweight paper [26]. Results for the server tests (which do not include C++ binaries) are more accurate, at a mean F-score of 0.75, but this is still 0.22 points lower than expected. Note that for BAP/ByteWeight, we excluded `xalancbmk` at O3 from the tests because of scalability issues (see Section 7.5.4).

To investigate this discrepancy more closely, we requested the trained version of ByteWeight used in the original paper from the authors. Unfortunately, the authors

	gcc x86	gcc x64	clang x86	clang x64	VS x86	VS x64
IDA Pro 6.7	0.98/0.78	0.97/0.74	0.98/0.78	0.98/0.77	0.84/0.93	1.00/0.94
BAP/BW 0.9.9	0.68/0.83	0.70/0.66	0.52/0.71	0.73/0.49	0.63/0.74	0.69/0.56
Dyninst 9.1.0	0.93/0.91	0.96/0.74	0.98/0.95	0.88/0.72	—	—
Nucleus	0.98/0.96	0.98/0.96	0.96/0.97	0.96/0.95	0.86/0.96	0.95/0.94
$\Delta_{Nucleus}$	+0.00/+0.05	+0.01/+0.22	−0.02/+0.02	−0.02/+0.18	+0.02/+0.03	−0.05/+0.00

(a) SPEC CPU2006 (all binaries, optimization levels O0–O3)

	gcc x86	gcc x64	clang x86	clang x64
IDA Pro 6.7	0.93/0.88	0.92/0.86	0.93/0.85	0.91/0.84
BAP/BW 0.9.9	0.71/0.91	0.78/0.86	0.57/0.84	0.79/0.65
Dyninst 9.1.0	0.91/0.96	0.92/0.85	0.93/0.97	0.87/0.85
Nucleus	0.98/0.98	0.98/0.97	0.99/0.99	0.99/0.96
$\Delta_{Nucleus}$	+0.05/+0.02	+0.06/+0.11	+0.06/+0.02	+0.08/+0.11

(b) Servers (C only, tested at per-server default optimization ranging from O0–O2)

Table 7.1: Precision/recall for *function start* detection (average geometric mean). $\Delta_{Nucleus}$ shows the improvement in *Nucleus* over other approaches.

replied that only an untrained version is still available. Given the uncertainties in attempting to exactly reproduce the training used in the ByteWeight paper, we instead performed a detailed analysis of the difference between our test suite, and the tests used in the ByteWeight paper. This analysis, which we discuss in Section 7.6, shows a significant overlap between the training set and test set used in the original ByteWeight evaluation. We show that this overlap causes a significant bias in evaluation results, which we believe is responsible for the accuracy discrepancy we observe. This is worrying, because the ByteWeight test suite has since been used to evaluate *all* top-tier work on function detection through machine learning.

7.5.2.2 Function Boundaries

Figure 7.3 and Table 7.2 show our results for function boundary detection. Recall from Section 7.2 that in contrast to function start detection, which only finds the first address of each function, function boundary detection involves finding both the first and the last address. As discussed in Section 7.3, *Nucleus* finds not only function boundaries, but all basic blocks belonging to each function. Nevertheless, for comparability with the results of other approaches, we measured our results for *Nucleus* by taking the lowest and highest address found for each function.

As before, Figure 7.3 graphs the F-scores achieved for SPEC CPU2006 in various configurations, while Table 7.2 decomposes these F-scores into precision and recall, and additionally shows results for our server tests. Again, *Nucleus* consistently outperforms other approaches, with the exception of IDA Pro on Visual Studio x64. (As discussed in Section 7.5.2.1, this is because Visual Studio uses only one calling convention on x64.) *Nucleus* achieves an overall mean F-score of 0.90 for the SPEC CPU2006 tests, while IDA Pro (the best performing alternative) yields a mean F-score of only 0.84, even including its extremely good results for Visual

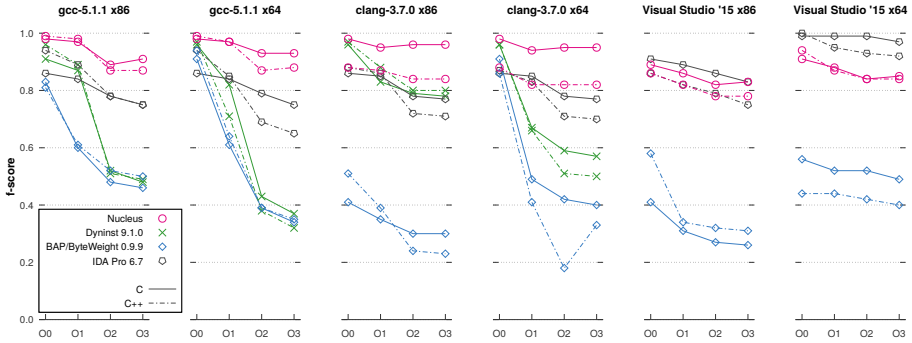


Figure 7.3: F-scores for *function boundary* detection (geometric mean for SPEC CPU2006).

Studio x64. For our server tests (which do not include C++ code), *Nucleus* achieves an even higher overall mean F-score of 0.97.

In addition, the standard deviations in F-score for *Nucleus* are lower than those for other approaches, meaning that *Nucleus* provides more predictable accuracy. *Nucleus* achieves an average standard deviation of only 0.02 for C, and 0.04 for C++. In contrast, IDA Pro, the best performing other approach, has an average standard deviation of 0.10 for C, and 0.11 for C++. Moreover, Figure 7.3 shows that *Nucleus* again achieves more stable results across compilers and architectures than other approaches, while better retaining its accuracy for highly optimized binaries.

7.5.3 Analysis of Results

In Section 7.2.4, we discussed challenging constructs for function detection. As shown in Section 7.5.2, *Nucleus* achieves significantly more accurate results than other approaches. To gain a better understanding of the errors which do occur in *Nucleus*, and the tradeoffs compared to other approaches, we select and manually analyze a random sample of 100 false positives and false negatives from our experiments. The sample includes all compilers and platforms we tested, and covers both our function start and function boundary detection experiments.

7.5.3.1 False Positives

As discussed in Section 7.3, *Nucleus* uses connected components analysis of the ICFG to detect functions without assuming anything about their memory layout, and without requiring any prologue/epilogue signatures. This has several benefits: it allows *Nucleus* to be compiler-agnostic, detect non-contiguous functions, and find unreachable or indirectly called functions which are missed by signature-based approaches. The tradeoff is that *Nucleus* requires switch analysis and address-taken analysis to correctly handle intraprocedural indirect jumps.

All of the false positives we analyzed, for ELF as well as PE binaries, are caused by inaccuracies in resolving intraprocedural indirect edges. For C binaries, this is

	gcc x86	gcc x64	clang x86	clang x64	VS x86	VS x64
IDA Pro 6.7	0.97/0.71	0.97/0.68	0.98/0.68	0.97/0.68	0.83/0.85	1.00/0.94
BAP/BW 0.9.9	0.60/0.60	0.63/0.53	0.34/0.34	0.68/0.41	0.40/0.32	0.61/0.40
Dyninst 9.1.0	0.89/0.60	0.91/0.51	0.98/0.75	0.85/0.57	—	—
Nucleus	0.97/0.89	0.97/0.90	0.95/0.88	0.94/0.86	0.85/0.84	0.94/0.85
$\Delta_{Nucleus}$	+0.00/+0.18	+0.00/+0.22	−0.03/+0.13	−0.03/+0.18	+0.02/−0.01	−0.06/−0.09

(a) SPEC CPU2006 (all binaries, optimization levels O0–O3)

	gcc x86	gcc x64	clang x86	clang x64
IDA Pro 6.7	0.93/0.83	0.92/0.81	0.93/0.83	0.92/0.82
BAP/BW 0.9.9	0.67/0.75	0.75/0.74	0.42/0.47	0.77/0.52
Dyninst 9.1.0	0.91/0.79	0.92/0.70	0.93/0.85	0.85/0.74
Nucleus	0.98/0.96	0.98/0.94	0.99/0.97	0.99/0.93
$\Delta_{Nucleus}$	+0.05/+0.13	+0.06/+0.13	+0.06/+0.12	+0.07/+0.11

(b) Servers (C only, tested at per-server default optimization ranging from O0–O2)

Table 7.2: Precision/recall for *function boundary* detection (average geometric mean). $\Delta_{Nucleus}$ shows the improvement in *Nucleus* over other approaches.

due to unresolved switch edges, which result in isolated case blocks. When *Nucleus* finds an isolated basic block, it flags this block as a possible indirectly called function entry, thereby producing a false positive. In C++ binaries, false positives are caused by both unresolved switch edges, and unresolved exception handling edges (again leading to isolated exception handling blocks). These results show that more sophisticated switch detection and exception handling detection, explored in related work [83; 106; 167; 196], could reduce the false positive rate in *Nucleus*.

7.5.3.2 False negatives

False negatives in *Nucleus*, for both function start and function boundary detection, are caused almost exclusively by tailcalls. In the random sample we analyzed, tailcalls are responsible for 96% of false negatives. An example of a tailcall causing a false negative is shown in Listing 7.2.

In a tailcall, a function (`0x5daf10` in Listing 7.2) ends with a `jmp` to another function (`0x5dab70`). This is an optimization frequently used by compilers—instead of inserting a `call` instruction at the end of a function, the compiler instead uses a `jmp` to remove the need for two subsequent `ret` instructions. Recall from Section 7.3 that *Nucleus* starts by looking for functions that are directly called, and then expands these by following control flow edges. Function `0x5dab70` is never reached by a direct call, and is therefore not found in this first phase. However, `0x5daf10` *is* called directly. When expanding function `0x5daf10`, *Nucleus* follows the tailcall edge to `0x5dab70`, merging the two functions and producing a false negative.

This produces false negatives *only* if the tailcalled function (the tailcallee) is never called directly. If it is, then it is found in the first analysis phase, and the problem does not occur. We have also seen cases where the tailcallee is called di-

```
00000000005daf10 <rli_size_so_far>:
  5daf10: 48 8b 47 08     mov rax,[rdi+0x08]
  5daf14: 48 8b 77 18     mov rsi,[rdi+0x18]
  5daf18: 48 89 c7        mov rdi,rax
  5daf1b: e9 50 fc ff ff  jmp 5dab70 <bit_from_pos>
```

Listing 7.2: False negative due to tailcall.

```
   44a36b:  mov   edi,0x628882
   44a370:  mov   esi,0x213
   44a375:  mov   edx,0x62888e
   44a37a:  call  47ce90 <fancy_abort>
   44a37f:  nop

 000000000044a380 <cfg_layout_initialize>:
   44a380:  push  rax
   44a381:  mov   edi,0x20
   44a386:  call  444970 <alloc_aux_for_blocks>
```

Listing 7.3: False negative due to fallthrough from non-returning `call`.

rectly (by another function than the tailcalling function), but the tailcaller is never called directly. To prevent function merging in these cases, *Nucleus* does not expand functions along inbound edges to directly called entry points (i.e., for directly called basic blocks the connected components analysis is directed rather than undirected).

Tailcalls are the main cause of false negatives in both function start and function boundary detection. For function boundary detection, a single tailcall can cause *two* false negatives: (1) a wrong start address for the tailcallee, and (2) a wrong end address for the tailcaller.

In some cases, merged functions are closely related, and the function performing the tailcall is merely a stub that sets up a parameter and performs the tailcall. *Nucleus* classifies such cases as multi-entry functions. Arguably, this could be considered correct. However, we count such cases as false negatives because the symbolic information specifies the merged functions as separate.

In most cases, the tailcaller and tailcallee are in distinct memory ranges. As such, extending *Nucleus* with the assumption that functions are contiguous in memory could remove these false negatives on platforms where this assumption holds. In this work, we chose not to add this assumption to *Nucleus*, as our aim is to show that *Nucleus* achieves accurate function detection *without* such assumptions.

As *Nucleus* does not currently implement detection for non-returning functions, it must assume that a `call` to such a function can return normally. This can cause false negatives, if a `call` to a non-returning function directly precedes another function that is itself never called. We show an example in Listing 7.3. Here, there is a `call` instruction at address `0x44a37a` that targets a non-returning function (`fancy_abort`). Directly after the `call` is the start of another function, at `0x44a380`. Since this function is never called directly, it is merged into the preceding function through the fallthrough edge from the `call`. This case is responsible for 4% of false negatives in our analysis, and again occurs only if the function after the non-returning `call` is itself never called.

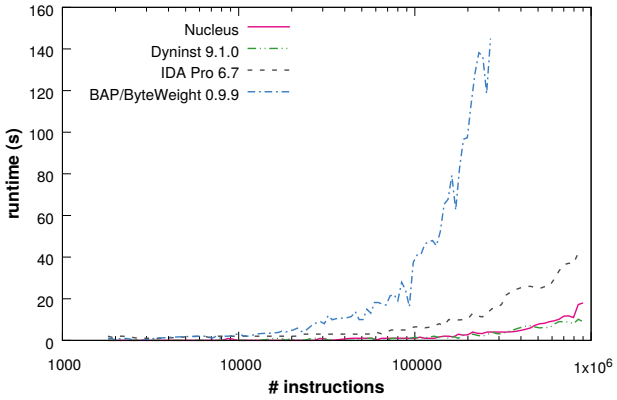


Figure 7.4: Runtime performance for function boundary detection. The x-axis (number of instructions) is logarithmic.

7.5.4 Runtime Performance

Figure 7.4 compares the runtime performance for *Nucleus* to other approaches. The measured runtimes include not only the function detection phases, but also the disassembly and (I)CFG analysis phases of all compared tools. *Nucleus* is among the fastest of the compared approaches, providing runtime performance comparable with Dyninst, and completing all of its analysis in under 20 seconds even for binaries with code sections in the order of 1×10^6 instructions. Both *Nucleus* and Dyninst scale roughly linearly. IDA Pro performs a more extensive analysis phase for each binary, and therefore requires 45 seconds to process the largest binary. Note that for BAP/ByteWeight, we were forced to exclude the largest binary (`xalancbmk` at O3) from our tests due to scalability issues, which can be observed from the steep increase in runtime for BAP/ByteWeight as binary size increases.

7.6 Analysis of Machine Learning in Function Detection

Several recent papers have investigated the use of machine learning techniques to automatically learn signatures for function recognition. Bao et al. use a machine learning system (ByteWeight) to construct a weighted prefix tree of known code sequences that delineate functions [26], while Shin et al. train Neural Networks to recognize functions [164]. Both of these papers report extremely accurate results.

Unfortunately, Shin et al. have not released an open-source version of their system, preventing us from directly comparing it with *Nucleus*. ByteWeight is available open-source, and is used for function detection in recent BAP versions [44]. The results for our tests with this version of ByteWeight (which we refer to as BAP/ByteWeight) are reported in Section 7.5.2. As described there, we were unable to reproduce the performance reported by Bao et al. for this ByteWeight version.

	Start			Boundary		
	p	*r*	*F*	*p*	*r*	*F*
ByteWeight [26]§	0.97	0.97	0.97	0.93	0.93	0.93
Neural Nets [164]§	0.99	0.99	0.99	0.97	0.94	0.95
Nucleus†	0.96	0.94	0.95	0.96	0.88	0.92

Table 7.3: Precision/recall/F-scores for function start and boundary detection (average scores for the test suite of Bao et al. [26]). † *Nucleus* results are for `gcc`, `clang` and Visual Studio. § Bao et al. [26] and Shin et al. [164] results are for `gcc`, `icc` and Visual Studio.

For instance, our BAP/ByteWeight tests with `gcc` on `x64` produced a mean F-score for function start detection of only 0.65, which is 0.32 points lower than the result presented in the original ByteWeight paper.

As mentioned in Section 7.5.2.1, we also requested the trained ByteWeight version tested in the original paper by Bao et al., in order to run it on our own test suite. Unfortunately, the authors were unable to provide us with this version of ByteWeight, as they did not retain the trained ByteWeight version used for their tests. We therefore provide an additional comparison of *Nucleus* against the results as presented by Bao et al. and Shin et al. in their respective papers (Section 7.6.1). Subsequently, we provide an in-depth analysis of the reasons for the diminished performance we observed in BAP/ByteWeight (Section 7.6.2). This analysis reveals inadvertent methodological errors in the evaluations of both Bao et al. and Shin et al., which cause a strong bias in the test suite they used for evaluation. This bias provides a likely explanation for the observed performance discrepancy.

7.6.1 Function Detection Performance

Table 7.3 compares the function detection results achieved in *Nucleus* to those presented by Bao et al. and Shin et al. Both machine learning papers use the same test suite, which consists of `coreutils`, `binutils` and `findutils`, and a number of Windows applications (see Section 7.6.2). We repeated our experiments with *Nucleus* for this same test suite, the only difference being that *Nucleus* is evaluated on `gcc`, `clang`, and Visual Studio, while both Bao et al. and Shin et al. used a compiler suite consisting of `gcc`, `icc` and Visual Studio.

The table shows that *Nucleus* achieves F-scores comparable to those presented by Bao et al. and Shin et al. For function start detection, the precision, recall and F-scores for the different approaches are within 0.05 points from each other. The function boundary detection scores are also comparable—*Nucleus* achieves higher precision than ByteWeight (fewer false positives), though with slightly lower recall (more false negatives). The overall F-scores are within 0.03 points from each other.

Though *Nucleus* performs well on the test suite used by Bao et al. and Shin et al., we opted to use our own test suite for our main evaluation. The reasons for this choice are explained in Section 7.6.2, which provides an in-depth analysis of the differences between our test suite and the tests done by Bao et al. and Shin et al.

7.6.2 Evaluation Methodology

As mentioned in Section 7.6, we observed a large discrepancy in the results achieved by ByteWeight on our own SPEC-based test suite compared to the results reported in the original ByteWeight paper [26]. The mean F-score was 0.32 points lower than expected, and this observation persisted across `gcc`, `clang` and Visual Studio. It also persisted across different versions of `gcc`, ranging from version 4.7 (used in the original ByteWeight evaluation) to version 5.1.1.

Upon closer inspection of the test suite used by Bao et al. to evaluate ByteWeight, we found that it contains many binaries with large amounts of common functions. In the remainder of this section, we show that this leads to a large bias in the results reported by Bao et al., due to a significant overlap between training set and test set. This is problematic, because ByteWeight is a machine learning approach, and thus the validity of its evaluation relies on a strong separation between training and test binaries. Moreover, the exact same problem occurs in the Neural Network-based approach by Shin et al., as they used the same evaluation test suite as Bao et al. for their own evaluation.

7.6.2.1 Test Suite for ELF Binaries

Bao et al. build their ELF test suite (for `gcc` and `icc`) from three popular open-source binary suites: `coreutils`, `binutils` and `findutils`. These contain 106, 16 and 7 binaries, respectively. Though all of these binary suites contain large amounts of shared code, we focus here on `coreutils`, as it comprises the majority of the Linux test suite. We perform our analysis on the binaries as compiled and used by Bao et al., which they make available online.[4] We focus our discussion here on the binaries compiled at optimization level O0, but we verified that the same effects occur at all optimization levels up to O3.

The `coreutils` binaries, as compiled by Bao et al. with `gcc` at O0, contain 1839 unique functions, distributed over 106 binaries (excluding PLT stubs and commonly named functions like `main`). There are 102 functions which occur in at least 90% of these binaries—mostly utility functions such as `xmalloc` and `quotearg`. We took a random subset of 50 such functions, comparing 2 randomly selected binaries for each function. In each case, the function body was shared *verbatim* between binaries, the only difference being in code addresses (which are normalized by ByteWeight).

Moreover, 87 functions occur in *all* binaries.[5] Since the average `coreutils` binary has 160 functions, this means that for the average binary, if selected for the test set, 54% of its functions are *guaranteed* to occur in the training set. The three binaries with the most functions are `mv`, `ginstall` and `vdir` (388, 358 and 355 functions, respectively). Thus, even these binaries share nearly 25% of their functions with *all* other `coreutils` binaries. The largest degree of overlap is found in

[4] http://security.ece.cmu.edu/byteweight/

[5] Except `make-prime-list`, which shares less code than other `coreutils` binaries

`true` and `false`; 94% of their functions are guaranteed to occur in the training set. In contrast, the average binary in our own SPEC-based test suite contains less than 1% of such shared functions (the only cases being bootstrap functions like `_start`).

The average `coreutils` binary shares 94% of its functions with *at least* one other binary in the test suite. This is because many `coreutils` binaries are extremely simple, often having only a `main` and `usage` function in addition to the shared utility functions.

Both Bao et al. and Shin et al. use 10-fold cross validation in their evaluations. This means that the set of binaries B is divided into two sets B_E and B_T, such that $B_E \cup B_T = B$. B_T consists of 90% of the binaries, and is used for training the system. The trained system is then evaluated on B_E, which contains the remaining 10% of binaries. This is done 10 times, each binary occurring in B_E exactly once.

To determine the precise probability of overlap between binaries in B_E and B_T, let b_f and c_f be two binaries that share function f. b_f has an $11/106$ chance of being chosen for B_E. Supposing that $b_f \in B_E$, c_f will be in B_T with probability $95/105 \approx 0.91$. Given that the average `coreutils` binary shares 94% of its functions with at least one other binary, for the average binary in B_E *at least* a fraction $0.94 \times 0.91 \approx 0.86$ (86%) of its functions are expected to occur in B_T.

7.6.2.2 Test Suite for PE Binaries

A similar situation occurs in the PE test suite used by both Bao et al. and Shin et al. for testing Visual Studio. We simply report the number of related binaries in the PE suite, rather than repeating the argument made for the ELF tests.

The PE test suite contains a total of 17 applications, from 7 open-source projects: `putty`, `7zip`, `vim`, `libsodium`, `libetpan`, `HID API`, and `pbc`. Out of these, 7 applications belong to the `putty` project: `pageant`, `plink`, `pscp`, `psftp`, `putty`, `puttygen` and `puttytel`. All of these share a common code base. Related applications are also found for the `7zip` project (3 related), `vim` (2 related) and `libetpan` (2 related). Overall, only 3 of the applications in the PE test suite do not have a relative that also occurs in the test suite.

In summary, it is clear that both the ELF and PE test suites used by Bao et al. and Shin et al. cause a strong bias in their evaluation results, preventing us from directly comparing these results to *Nucleus*. We believe this bias is the most likely explanation for the drop in accuracy when testing ByteWeight on our own test suite. Given this bias, the results presented by Bao et al. and Shin et al. cannot currently be assumed to generalize. Thus, further research in this area is needed to reassess the viability of machine learning for function detection.

7.7 Discussion

Chapter 6 has shown that in existing approaches, function detection is among the most compiler-specific and error-prone stages of the binary analysis process. To

the best of our knowledge, *Nucleus* is the first approach which shows that accurate function detection can be achieved in a completely compiler-agnostic way, with significantly fewer false positives and false negatives than existing work. This enables function detection for binaries compiled with new or unknown compilers, and eliminates the need for maintaining signature databases.

We show in Section 7.6 that existing work, which aims to reduce maintenance costs through machine learning [26; 164], suffers from a significant evaluation bias due to overlapping training and test sets. In principle, it should be possible for these approaches to match the accuracy of other signature-based approaches, such as IDA Pro. Unfortunately, the question of whether or not they can exceed this accuracy remains to be answered in future work. While machine learning approaches do succeed in reducing manual maintenance, *Nucleus* eliminates maintenance completely, while achieving higher accuracy than any of the other approaches we tested.

To demonstrate the generality of *Nucleus*, in this work we have limited our assumptions on function structure to a minimum. We assume only that intraprocedural control transfers follow a different general pattern than interprocedural control flow. In contrast, existing work, including machine learning approaches, inherently relies on compiler-specific function prologue and epilogue patterns [26; 83; 164], which are not always present at high optimization levels.

Section 7.5.3 shows that most false negatives in *Nucleus* (resulting from tailcalls) can be eliminated if it can be assumed that functions are laid out contiguously in memory. Although we opted not to make this assumption, the open source version of *Nucleus* contains a command-line option to enable this assumption when it is known that functions are contiguous. We stress that this feature is strictly optional; it is not required by *Nucleus*, and is disabled in all tests in this chapter.

Since the vast majority of false positives result from unresolved indirect intraprocedural flows (Section 7.5.3), *Nucleus* benefits from advances made in switch detection and reverse engineering of exception handling constructs [83; 106; 167; 196].

Though *Nucleus* does not explicitly target malware or obfuscated binaries, its lack of assumptions on low-level code structure enable *Nucleus* to handle some common types of obfuscations more accurately than signature-based work. For instance, *Nucleus* is agnostic to instruction-level polymorphism, and to obfuscators that intentionally rewrite function prologues and epilogues to non-standard variants [154]. Additionally, *Nucleus* provides inherent support for finding indirectly called functions, making it immune to obfuscations that obscure function calls by using branching functions, or transforming direct calls to indirect calls [110; 118]. We chose not to evaluate these aspects of *Nucleus*, due to the large variety of obfuscation techniques used in practice and the lack of ground truth for malware samples. Instead, we defer proper handling of obfuscated malware to related work on deobfuscation, which can be used in unison with *Nucleus* [110; 154].

For compilers which emit highly predictable function patterns, our results show that traditional signature-based approaches perform extremely well (Section 7.5.2.1). In our tests, this was only the case for Visual Studio on x64, where IDA Pro took full

advantage of the predictability in calling convention. However, as our results also show, the majority of compilers use a variety of calling conventions and function patterns, causing a decline in the accuracy of signature-based approaches. In all these cases, *Nucleus* provides significantly higher accuracy.

7.8 Related Work

Previous work has used machine learning to automatically generate signatures, eliminating the need for manual maintenance [26; 164]. However, these approaches still require an expensive learning phase for every new compiler version, and cannot handle unknown compilers. In contrast, *Nucleus* is a completely compiler-agnostic and zero-maintenance approach.

Signature-based function detection is used in all major disassemblers [31; 44; 83; 106; 165]. Several papers have found that function detection is significantly more inaccurate than other primitives such as instruction or CFG recovery [22; 125]. At the same time, function detection is widely used in binary analysis, ranging from binary-level Control Flow Integrity [23; 126; 144; 196; 197; 199] to automatic vulnerability detection [86; 140], binary instrumentation [31; 114], and manual binary analysis [160; 191]. Thus, our results for *Nucleus* facilitate work in a large range of binary analysis applications.

Our approach to disassembly is based on linear disassembly with error correction. Similar approaches have been explored in the context of high-coverage Control Flow Integrity [199], deobfuscation [110], and binary instrumentation [114; 198]. In all these cases, linear disassembly results have proven extremely accurate, a finding confirmed in recent work on disassembly accuracy [22].

7.9 Conclusion

This chapter has shown that compiler-agnostic function detection can achieve high accuracy. We have shown that *Nucleus*, our function detection approach, provides significantly more accurate results than existing approaches in terms of both function start and function boundary detection, without making any compiler-specific assumptions. *Nucleus* provides inherent handling of complex cases such as non-contiguous and multi-entry functions, and functions with unknown prologues or epilogues, which are not handled in current signature-based work. Moreover, we have found a significant bias in the evaluations of existing approaches that aim to reduce maintenance costs for function signature databases through machine learning, showing the need for future work to reassess the viability of these approaches. In addition to achieving more accurate results than existing work, *Nucleus* is zero-maintenance, supporting new or unknown compilers without any additional effort. We provide *Nucleus* open-source, including the option to transfer results to IDA Pro, making it straightforward to use *Nucleus* in real-world environments.

Discussion

Chapter 6 shows that many of the complex constructs introduced in Chapter 2 are far less prevalent in modern compiler-generated binaries than previously thought [31; 125; 135]. For instance, modern ELF binaries contain no inline data, and even on Visual Studio binaries which do contain a degree of inline data, the impact on disassembly accuracy is limited. In practice, for most primitives, the tested disassemblers achieve sufficient precision to allow them to serve as a basis for binary-level security work. Combined with the mechanisms discussed in Part I of this thesis for dealing with disassembly errors, strong safety guarantees can be provided, with only limited performance and analysis overhead.

Ironically, potential errors in function detection are discussed less in the literature than problems for any other kind of primitive (Chapter 6.5), even though function detection is the least reliable primitive of all. At the same time, Chapter 6.4 shows that the lack of precision in function detection has strong implications in many popular areas of binary-based security research, including Control-Flow Integrity. This clearly demonstrates the need for improved function detection, as we pursue in our work in Chapter 7.

As discussed in Chapter 7, our novel function detection approach retrieves a significantly larger fraction of the functions (more true positives) than any of the other tested disassemblers, including IDA Pro. Moreover, it achieves this without increasing (indeed, often even reducing) the false positive rate. These results directly translate to improved security guarantees and efficiency in virtually all of the discussed areas of binary analysis-based research.

In summary, our results provide a more stable foundation for future binary-level security research, both by improving the precision of the previously least reliable primitives, and by reducing the gap between true disassembler performance and the expectations of reviewers and researchers in the field.

Chapter 8

Conclusions

Our goals in this thesis have been twofold. (1) Building methods for using potentially inaccurate disassembly to safely build secure systems, and (2) Quantifying and improving the precision of the disassembly primitives underlying binary-based systems, to provide a better understanding of disassembly guarantees for both reviewers and researchers. In Chapter 1, we formulated four research questions around these key goals. We now recapitulate the conclusions we have reached regarding each of these research questions. We also provide a discussion of the limitations of our work, and possibilities for future research.

8.1 Results

Here, we briefly recall each of the research questions we asked in Chapter 1, summarizing our main conclusions. Our main conclusions regarding each of the research questions are as follows.

Question (1): *Given all the potential disassembly inaccuracies, how can we effectively and safely apply binary analysis to build systems for analyzing and securing legacy and proprietary binaries?*

In Part I (Chapters 3–5) of this thesis, we arrived at several strategies for achieving crash-safety and low overhead in binary-based security solutions. Specifically, it is possible to apply binary analysis in such a way that the analysis result is either an overapproximation, or an underapproximation of the ideal result. Thus, even when we cannot guarantee that the result is fully correct, we can still guarantee the absence of false negatives or false positives, respectively. We have shown that these relatively weak guarantees are sufficient to build safe and efficient binary analyses. In addition, we have considered the use of other techniques for minimizing instrumentation errors, fixing errors at runtime, and using policy-driven checks to flexibly utilize source-level information (only) when available.

To minimize the overhead induced by binary analysis and instrumentation, we have considered the use of runtime analysis, limited to executed paths (as opposed to the exponentially large number of paths to be considered statically). Moreover, we have considered techniques for reducing the necessary instrumentation code (and thus overhead) to a minimum.

Question (2): *How precise is disassembly (in the broadest sense) in practice, and how frequently should we expect inaccuracies of each possible kind?*

This question is addressed in Chapter 6. We have shown that several complex constructs are far less prevalent than previously thought, and that disassembly results in practice are quite accurate for the best performing disassemblers. The exception to this is function detection, for which we have shown that significant amounts of both false positives and false negatives are to be expected.

Question (3): *To what extent is there a consensus on disassembly precision in the binary analysis community, and where is that consensus mismatched with our findings from Question (2)?*

Like Question (2), this question is addressed in Chapter 6. We have reviewed binary analysis papers published in six major conferences over the course of three years, revealing a strong focus on complex corner cases which are unlikely to occur in practice. At the same time, truly problematic cases like function detection are hardly considered at all. Thus, we have shown (and hopefully alleviated) a considerable mismatch between expectations in the binary analysis community, and the actual performance of modern disassemblers.

Question (4): *How can we achieve more precise function detection results?*

Finally, this question is addressed in Chapter 7, where we designed and implemented a novel function detection algorithm which yields considerably better results than existing approaches. The key insight here is to focus on Control Flow Graph-based analysis, rather than relying on compiler-specific instruction-level signatures for detecting functions. By conducting our analysis at the CFG level and using connected component analysis, we are better able to locate functions without standard prologues, as well as indirectly called functions not found by traditional methods.

8.2 Limitations and Future Work

In Chapter 7, we implemented an improved function detection approach that achieves significantly more accurate results than existing approaches. Given these results, we can now achieve reasonably accurate performance for all disassembly primitives discussed in Chapter 2. In accordance with our focus in this thesis, these are all code primitives. In contrast, the precision of data structure reverse engineering still lags

far behind code reverse engineering [168]. Thus, while some research on reversing data structures has been undertaken in the past [116; 168], this remains a promising direction for future research. Improved results in this area would have an important impact on both manual reverse engineering and automated security solutions; for instance, *StackArmor* (Chapter 3) would benefit greatly from improved analysis of stack objects.

A remaining area in code reverse engineering which may benefit from future research is the matching of indirect call sites to address-taken functions. While our work in Chapter 7 has improved the detection of address-taken functions themselves, it has not addressed the problem of accurately analyzing which (indirect) call sites may call these functions. More accurate results in this area would be extremely useful to, for instance, Control-Flow Integrity solutions like *PathArmor* (Chapter 4), which rely on such information to determine CFI policies. Recent work has improved binary-level call site/function matching by using (conservative) information on function arguments [181]. However, further contributions are worth the effort, as they translate directly to (for instance) improved CFI solutions.

Finally, it is important to reiterate that our work has focused on x86/x64 disassembly. Some of our conclusions may not necessarily hold for other platforms. In particular, complex constructs such as inline data may be significantly more prevalent in embedded platforms, or in firmware, than we have found to be the case for the platforms evaluated in Chapter 6. An interesting direction for future research may thus be to conduct a study similar to Chapter 6 for embedded platforms, or even obfuscated code.

References

[1] Apache Benchmark. http://httpd.apache.org/docs/2.0/programs/ab.html.

[2] LLVM DSA - Reproducing the Result in the PLDI'07 Paper. http://lists.cs.uiuc.edu/pipermail/llvmdev/2015-May/085390.html.

[3] memslap. http://docs.libmemcached.org/bin/memslap.html.

[4] OpenSSH Portable Regression Tests. http://dtucker.net/openssh/regress.

[5] pyftpdlib. https://code.google.com/p/pyftpdlib.

[6] ROPC: ROP compiler. https://github.com/pakt/ropc.

[7] SendEmail. http://caspian.dotconf.net/menu/Software/SendEmail.

[8] SysBench. http://sysbench.sourceforge.net.

[9] System V Application Binary Interface (AMD64). http://www.x86-64.org/documentation/abi.pdf.

[10] ASLR: Leopard versus Vista. http://blog.laconicsecurity.com/2008/01/aslr-leopard-versus-vista.html, 2008.

[11] A Framework for Aviation Cybersecurity, 2013. Technical report, American Institute of Aeronautics and Astronautics.

[12] Switching from "-fstack-protector" to "-fstack-protector-strong" in Fedora 20. http://fedorahosted.org/fesco/ticket/1128, 2013.

[13] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. A Theory of Secure Control-Flow. In *Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM'05)*, 2005.

[14] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity. In *Proceedings of the 12th Conference on Computer and Communications Security (CCS'05)*, 2005.

[15] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity: Principles, Implementations, and Applications. *ACM TISSEC*, 13(1), 2009.

[16] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[17] Periklis Akritidis. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *Proceedings of the 19th USENIX Security Symposium (USENIX Sec'10)*, 2010.

[18] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P'08)*, 2008.

[19] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense against Out-of-Bounds Errors. In *Proceedings of the 18th USENIX Security Symposium (USENIX Sec'09)*, 2009.

[20] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*, 2013.

[21] Kapil Anand, Matthew Smithson, Aparna Kotha, Khaled Elwazeer, and Rajeev Barua. Decompilation to Compiler High IR in a Binary Rewriter, 2010. Technical report, University of Maryland.

[22] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25th USENIX Security Symposium (USENIX Sec'16)*, Austin, TX, USA, August 2016. USENIX.

[23] Dennis Andriesse, Victor van der Veen, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)*, Denver, CO, USA, October 2015. ACM.

[24] Michael Backes and Stefan Nürnberger. Oxymoron Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)*, 2014.

[25] Gogul Balakrishnan and Thomas Reps. WYSINWYX: What You See is Not What You eXecute. *ACM Transactions on Programming Languages and Systems*, 32(6):23:1–23:84, August 2010.

[26] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)*, 2014.

[27] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (Im)possibility of Obfuscating Programs. In *Proceedings of the 21st Annual International Cryptology Conference (Crypto'01)*, 2001.

[28] Mario Ballano Barcena and Candid Wueest. Insecurity in the Internet of Things, 2015. Technical report, Symantec. https://www.symantec.com/content/dam/symantec/docs/white-papers/insecurity-in-the-internet-of-things-en.pdf.

[29] Robert Bartels. The Rank Version of von Neumann's Ratio Test for Randomness. *Journal of the American Statistical Association*, 77(377), 1982.

[30] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, 2006.

[31] Andrew R. Bernat and Barton P. Miller. Anywhere, Any-Time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, 2011.

[32] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium (USENIX Sec'03)*, 2003.

[33] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium (USENIX Sec'05)*, 2005.

[34] Philippe Biondi and Fabrice Desclaux. Silver Needle in the Skype. In *Black Hat Europe*, 2006.

[35] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating Code-reuse Attacks with Control-flow Locking. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*, 2011.

[36] Hans-J. Boehm. Bounding Space Usage of Conservative Garbage Collectors. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[37] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)*, 2015.

[38] Jean-Marie Borello and Ludovic Mé. Code Obfuscation Techniques for Metamorphic Viruses. *J. Computer Virology and Hacking Techniques*, 2008.

[39] Erik Bosman and Herbert Bos. Framing Signals—A Return to Portable Shellcode. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, 2014.

[40] Pete Broadwell, Matt Harren, and Naveen Sastry. Scrash: A System for Generating Secure Crash Information. In *Proceedings of the 12th USENIX Security Symposium (USENIX Sec'03)*, 2003.

[41] Derek Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[42] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the Symposium on Code Generation and Optimization (CGO'03)*, 2003.

[43] Derek Bruening and Qin Zhao. Practical Memory Checking with Dr. Memory. In *Proceedings of the 9th International Symposium on Code Generation and Optimization*, 2011.

[44] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*, 2011.

[45] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC. In *Proceedings of the 15th Conference on Computer and Communications Security (CCS'08)*, 2008.

[46] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *International Journal of High Performance Computing Applications (IJHPCA)*, 14(4), 2000.

[47] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012.

[48] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Sec'15)*, 2015.

[49] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)*, 2014.

[50] Miguel Castro, Manuel Costa, and Tim Harris. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[51] Hoi Chang and Mikhail J. Atallah. Protecting Software Code by Guards. In *Proceedings of Digital Rights Management Symposium (DRM'01)*, 2001.

[52] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-Oriented Programming Without Returns. In *Proceedings of the 17th Conference on Computer and Communications Security (CCS'10)*, 2010.

[53] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, 2011.

[54] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. StackArmor: Comprehensive Protection from Stack-Based Memory Error Vulnerabilities for Binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)*, San Diego, CA, USA, February 2015. Internet Society.

[55] Yuqun Chen, Ramarathnam Venkatesan, Matthew Cary, Ruoming Pang, Saurabh Sinha, and Mariusz H. Jakubowski. Oblivious Hashing: A Stealthy Software Integrity Verification Primitive. In *Proceedings of the ACM Workshop on Information Hiding and Multimedia Security (IH'05)*, 2003.

[56] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. ROPecker: A Generic and Practical Approach for Defending Against ROP Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'14)*, 2014.

[57] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A Platform for In-vivo Multi-path Analysis of Software Systems. *ACM SIGPLAN Notices*, 46(3):265–278, 2011.

[58] Tzi-Cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, 2001.

[59] Jim Chow, Ben Pfaff, Tal Garfinkel, and Mendel Rosenblum. Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation. In *Proceedings of the 14th USENIX Security Symposium (USENIX Sec'05)*, 2005.

[60] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 1st edition, 2009.

[61] Marc L. Corliss, E. Christopher Lewis, and Amir Roth. Using DISE to Protect Return Addresses from Attack. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus*, 2004.

[62] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-end Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, 2005.

[63] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. Static Detection of Vulnerabilities in x86 Executables. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006.

[64] Crispin Cowan, Coltan Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattle, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium (USENIX Sec'98)*, 1998.

[65] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant Systems: A Secretless Framework for Security Through Diversity. In *Proceedings of the 15th USENIX Security Symposium (USENIX Sec'06)*, 2006.

[66] Anton Kuijsten Cristiano Giuffrida and Andrew S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Security Symposium (USENIX Sec'12)*, 2012.

[67] John Criswell, Nathan Dautenhahn, and Vikram Adve. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, 2014.

[68] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007.

[69] Cryptic Apps. Hopper Disassembler v3. https://www.hopperapp.com/.

[70] CVE-2008-0063. Kerberos stack-based information leak. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2008-0063, 2008.

[71] CVE-2010-0262. Microsoft Office arbitrary code execution vulnerability due to uninitialized stack variable. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-0262, 2010.

[72] CVE-2012-5976. Asterisk stack-based buffer overflow. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5976, 2013.

[73] CVE-2013-4368. Xen stack-based information leak. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-4368, 2013.

[74] Thurston HY Dang, Petros Maniatis, and David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS'15)*, 2015.

[75] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)*, 2015.

[76] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)*, 2014.

[77] B. de Sutter, B. de Bus, and K. de Bosschere. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of the 4th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00)*, 2000.

[78] Dinakar Dhurjati and Vikram Adve. Backwards-compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[79] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'06)*, 2006.

[80] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory Safety Without Runtime Checks or Garbage Collection. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems*, 2003.

[81] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory Safety Without Garbage Collection for Embedded Applications. *ACM Transactions on Embedded Computer Systems*, 2005.

[82] DWARF Debugging Information Format Committee. DWARF Debugging Information Format Version 4. http://www.dwarfstd.org/doc/DWARF4.pdf.

[83] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. 2nd edition, 2011.

[84] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket Execution: Dynamic Similarity Testing for Program Binaries and Components. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)*, 2014.

[85] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*, 2006.

[86] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'16)*, 2016.

[87] Hiroaki Etoh and Kunikazu Yoda. ProPolice – Improved Stack Smashing Attack Detection. *IPSJ SIGNotes Computer Security*, 14, 2001.

[88] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)*, Denver, CO, USA, 2015. ACM.

[89] F. Falcon and N. Riva. Dynamic Binary Instrumentation Frameworks: I Know You're There Spying On Me. In *RECON'12*, 2012.

[90] Ivan Fratric. Runtime Prevention of Return-Oriented Programming Attacks, 2012. Technical report.

[91] Nicu G. Fruja. The Correctness of the Definite Assignment Analysis in C#. In *Proceedings of the 2nd International Workshop on .NET Technologies*, 2004.

[92] Robert Gawlik and Thorsten Holz. Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*, 2014.

[93] Jonathon Giffin, Mihai Christodorescu, and Louis Kruger. Strengthening Software Self-Checksumming via Self-Modifying Code. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC'05)*, 2005.

[94] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. Practical Automated Vulnerability Monitoring Using Program State Invariants. In *Proceedings of the 43rd Conference on Dependable Systems and Networks (DSN'13)*, 2013.

[95] GNU Project. GNU Binutils. https://www.gnu.org/software/binutils/.

[96] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of Control: Overcoming Control-flow Integrity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, 2014.

[97] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)*, 2014.

[98] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)*, 2013.

[99] Xin Hu and Kang G. Shin. DUET: Integration of Dynamic and Static Analyses for Malware Clustering with Cluster Ensembles. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC'13)*, 2013.

[100] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer Manuals. http://www.intel.com/products/processor/manuals/.

[101] Intel Corporation. Pin 2.14 User Guide. https://software.intel.com/sites/landingpage/pintool/docs/67254/Pin/html/.

[102] Matthias Jacob, Mariusz H. Jakubowski, and Ramarathnam Venkatesan. Towards Integral Binary Execution: Implementing Oblivious Hashing Using Overlapped Instruction Encodings. In *Proceedings of the Conference on Multimedia and Security (MM&Sec'07)*, 2007.

[103] Jiyong Jang, Maverick Woo, and David Brumley. Towards Automatic Software Lineage Inference. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)*, 2013.

[104] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'02)*, 2002.

[105] Ye Joey. gcc patch: "merge stack alignment branch". https://gcc.gnu.org/ml/gcc-patches/2008-04/msg00349.html, 2008.

[106] Johannes Kinder. *Static Analysis of x86 Executables*. PhD thesis, Technische Universität Darmstadt, 2010.

[107] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure Execution via Program Shepherding. In *Proceedings of the 11th USENIX Security Symposium (USENIX Sec'02)*, 2002.

[108] Toshihiko Koju, Reid Copeland, Motohiro Kawahito, and Moriyoshi Ohara.

Re-constructing High-level Information for Language-specific Binary Re-optimization. In *Proceedings of the Symposium on Code Generation and Optimization (CGO'16)*, 2016.

[109] S. Krishnamoorthy, M.S. Hsiao, and L. Lingappan. Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs. In *Proceedings of the 19th IEEE Asian Test Symposium (ATS'10)*, 2010.

[110] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of the 13th USENIX Security Symposium (USENIX Sec'04)*, 2004.

[111] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer Integrity. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*, 2014.

[112] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the Third International Symposium on Code Generation and Optimization*, 2004.

[113] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-sensitive Points-to Analysis with Heap Cloning Practical for the Real World. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, 2007.

[114] Michael Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2010.

[115] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High Accuracy Attack Provenance via Binary-based Execution Partition. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[116] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the 17th Network and Distributed System Security Symposium (NDSS'10)*, 2010.

[117] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Reuse-Oriented Camouflaging Trojan: Vulnerability Detection and Attack Construction. In *Proceedings of the 40th Conference on Dependable Systems and Networks (DSN'10)*, 2010.

[118] Cullen Linn and Saumya Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*, 2003.

[119] Kangjie Lu, Siyang Xiong, and Debin Gao. RopSteg: Program Steganography with Return Oriented Programming. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY'14)*, 2014.

[120] Kangjie Lu, Dabi Zou, Weiping Wen, and Debin Gao. deRop: Removing Return-Oriented Programming from Malware. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC'11)*, 2011.

[121] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, 2005.

[122] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: Trading Address Space for Reliability and Security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[123] Matthew C. Merten and Michael S. Thiems. An Overview of the IMPACT x86 Binary Reoptimization Framework, 1998. Technical report.

[124] Microsoft Developer Network. Overview of x64 Calling Conventions, 2015. https://msdn.microsoft.com/en-us/library/ms235286.aspx.

[125] Barton P. Miller and Xiaozhu Meng. Binary Code is Not Easy, 2015. Technical report, University of Wisconsin-Madison.

[126] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. Opaque Control-Flow Integrity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[127] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proceedings of the 28th IEEE Symposium on Security and Privacy (S&P'07)*, 2007.

[128] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[129] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the Third International ACM SIGPLAN Conference on Virtual Execution Environments*, 2007.

[130] James Newsome and Dawn Song. Dynamic Taint Analysis: Automatic Detection, Analysis, and Signature Generation of Exploit Attacks on Commodity Software. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'05)*, 2005.

[131] Ben Niu and Gang Tan. Monitor Integrity Protection with Space Efficiency and Separate Compilation. In *Proceedings of the 20th Conference on Computer and Communications Security (CCS'13)*, 2013.

[132] Ben Niu and Gang Tan. Modular Control-flow Integrity. In *Proceedings of the 35th Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[133] Ben Niu and Gang Tan. RockJIT: Securing Just-In-Time Compilation Using Modular Control-Flow Integrity. In *Proceedings of the 21st Conference on Computer and Communications Security (CCS'14)*, 2014.

[134] Gene Novark and Emery D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, 2010.

[135] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. N-Version Disassembly: Differential Testing of x86 Disassemblers. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA'10, 2010.

[136] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)*, 2013.

[137] PaX Team. Overall Description of the PaX Project. http://pax.grsecurity.net/docs/pax.txt, 2008.

[138] Mathias Payer, Antonio Barresi, and Thomas R. Gross. Fine-Grained Control-Flow Integrity through Binary Hardening. In *Proceedings of the 12th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'15)*, 2015.

[139] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)*, 2014.

[140] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, 2015.

[141] Jannik Pewny, Felix Schuster, Christian Rossow, Lukas Bernhard, and Thorsten Holz. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC'14)*, 2014.

[142] Daniel Plohmann. x86 Opcode Structure and Instruction Overview. https://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/x86_opcode_structure_and_instruction_overview.pdf.

[143] Georgios Portokalidis and Angelos D. Keromytis. Fast and Practical Instruction-set Randomization for Commodity Systems. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*, 2010.

[144] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)*, San Diego, CA, USA, February 2015. Internet Society.

[145] Manish Prasad and Tzi cker Chiueh. A Binary Rewriting Defense against Stack-based Buffer Overflow Attacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'03)*, 2003.

[146] Rui Qiao, Mingwei Zhang, and R. Sekar. A Principled Approach for ROP Defense. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, 2015.

[147] Nguyen Anh Quynh. Capstone: Next-Gen Disassembly Framework. In *Black-hat USA*, 2014.

[148] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise Garbage Collection for C. In *Proceedings of the 8th International Symposium on Memory Management*, 2009.

[149] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the USENIX Windows NT Workshop (NT'97)*, 1997.

[150] Chad Rosier. Support for Dynamic Stack Realignment + VLAs for x86. http://lists.cs.uiuc.edu/pipermail/llvm-commits/Week-of-Mon-20120702/146062.html, 2014.

[151] Christian Rossow, Dennis Andriesse, Tillmann Werner, Brett Stone-Gross, Daniel Plohmann, Christian Dietrich, and Herbert Bos. P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[152] Brent G. Roth and Eugene H. Spafford. Implicit Buffer Overflow Protection Using Memory Segregation. In *Proceedings of the 6th International Conference on Availability, Reliability and Security*, 2011.

[153] Kevin A. Roundy and Barton P. Miller. Binary-Code Obfuscations in Prevalent Packer Tools. *ACM Computing Surveys*, 2012.

[154] H. Saïdi, V. Yegneswaran, and P.A Porras. Experiences in Malware Binary Deobfuscation. *Virus Bulletin*, 2010.

[155] Babak Salamat, Andreas Gal, Todd Jackson, Karthikeyan Manivannan, Gregor Wagner, and Michael Franz. Multi-variant Program Execution: Using Multi-core Systems to Defuse Buffer-Overflow Vulnerabilities. In *Proceedings of the International Conference on Complex, Intelligent and Software Intensive Systems*, 2008.

[156] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-space. In *Proceedings of the 4th European Conference on Computer Systems*, 2009.

[157] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming. In *Proceedings of the 36th Symposium on Security and Privacy (S&P'15)*, 2015.

[158] Felix Schuster, Thomas Tendyck, Jannik Pewny, Andreas Maaß, Martin Steegmanns, Moritz Contag, and Thorsten Holz. Evaluating the Effectiveness of Current Anti-ROP Defenses. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14)*, 2014.

[159] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Security Symposium (USENIX Sec'11)*, 2011.

[160] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)*, 2013.

[161] Benjamin Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*, 2002.

[162] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'12)*, 2012.

[163] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-Libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS'07)*, 2007.

[164] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium (USENIX Sec'15)*, 2015.

[165] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. 2015.

[166] Saravanan Sinnadurai, Qin Zhao, and Weng-Fai Wong. Transparent Runtime Shadow Stack: Protection Against Malicious Return Address Modifications, 2004.

[167] Igor Skochinsky. Compiler Internals: Exceptions and RTTI. In *RECON'12*, 2012.

[168] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the Network and Distributed System Symposium (NDSS'11)*, 2011.

[169] Asia Slowinska, Traian Stancescu, and Herbert Bos. Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'12)*, 2012.

[170] Matthew Smithson, K. Anand, and A. Kotha. Binary Rewriting Without Relocation Information, 2010. Technical report, University of Maryland.

[171] K.Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[172] Mary Lou Soffa, Kristen R. Walcott, and Jason Mars. Exploiting Hardware Advances for Software Testing and Debugging. In *Proceedings of the International Conference on Software Engineering (ICSE'11)*, 2011.

[173] A. Sotirov. Heap Feng Shui in JavaScript. In *Black Hat Europe*, 2007.

[174] Abhinav Srivastava and Jonathon Giffin. Automatic Discovery of Parasitic Malware. In *Proceedings of the 13th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'10)*, 2010.

[175] Mark Stanislav and Tod Beardsley. Hacking IoT: A Case Study on Baby Monitor Exposures and Vulnerabilities, 2015. Technical report, Rapid7.

[176] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *Proceedings of the 22nd Conference on Computer and Communications Security (CCS'15)*, 2015.

[177] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC and LLVM. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)*, 2014.

[178] Mustafa M. Tikir, Michael Laurenzano, Laura Carrington, and Allan Snavely. PMaC Binary Instrumentation Library for PowerPC/AIX. In *Workshop on Binary Instrumentation and Applications*, 2006.

[179] Trail of Bits. A Preview of McSema, 2014. Technical report. http://blog.trailofbits.com/2014/06/23/a-preview-of-mcsema/.

[180] United States Department of Defense. DoD Software Protection Initiative. In *Systems and Software Technology Conference (SSTC'06)*, 2006. Description at http://sstc-online.org/2006/index.cfm?fs=exhlist&Letter=D.

[181] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A Tough call: Mitigating Advanced Code-Reuse Attacks At The Binary Level. In *Proceedings of the 37th Symposium on Security and Privacy (S&P'16)*, May 2016.

[182] Vendicator. Stack Shield: A "Stack Smashing" Technique Protection Tool for Linux. http://www.angelfire.com/sk/stackshield/info.html, 2001.

[183] Sebastian Vogl, Robert Gawlik, Behrad Garmany, Thomas Kittel, Jonas Pfoh, Claudia Eckert, and Thorsten Holz. Dynamic Hooks: Hiding Control Flow Changes within Non-Control Data. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Sec'14)*, 2014.

[184] David Wagner and Drew Dean. Intrusion Detection via Static Analysis. In *Proceedings of the 22nd IEEE Symposium on Security and Privacy (S&P'01)*, 2001.

[185] Minghua Wang, Heng Yin, Abhishek Vasisht Bhaskar, Purui Su, and Dengguo Feng. Binary Code Continent: Finer-Grained Control Flow Integrity for Stripped Binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, 2015.

[186] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable Disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Sec'15)*, 2015.

[187] Zhi Wang and Xuxian Jiang. HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P'10)*, 2010.

[188] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani M. Thuraisingham. Differentiating Code from Data in x86 Binaries. In *Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, 2011.

[189] G. Wurster, P. van Oorschot, and A. Somayaji. A Generic Attack on

Checksumming-Based Software Tamper Resistance. In *Proceedings of the 26th Symposium on Security and Privacy (S&P'05)*, 2005.

[190] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting Violation of Control Flow Integrity Using Performance Counters. In *Proceedings of the 42nd Conference on Dependable Systems and Networks (DSN'12)*, 2012.

[191] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[192] Heng Yin, Zhenkai Liang, and Dawn Song. HookFinder: Identifying and Understanding Malware Hooking Behaviors. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'08)*, 2008.

[193] Yves Younan, Pieter Philippaerts, Lorenzo Cavallaro, R. Sekar, Frank Piessens, and Wouter Joosen. PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, 2010.

[194] Yves Younan, D. Pozza, Frank Piessens, and Wouter Joosen. Extended Protection Against Stack Smashing Attacks Without Performance Loss. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, 2006.

[195] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A Retargetable Framework for Low-Level Inlined-Reference Monitors. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)*, 2013.

[196] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the Network and Distributed System Security Symposium (NDSS'15)*, 2015.

[197] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P'13)*, 2013.

[198] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R. Sekar. A Platform for Secure Static Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'14)*, 2014.

[199] Mingwei Zhang and R. Sekar. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Sec'13)*, 2013.

[200] Mingwei Zhang and R. Sekar. Control Flow and Code Integrity for COTS binaries. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, 2015.

[201] Yang Zhang, Xiaoshan Sun, Yi Deng, Liang Cheng, Shuke Zeng, Yu Fu, and Dengguo Feng. Improving Accuracy of Static Integer Overflow Detection in Binary. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)*, 2015.

Contributions to Papers

Some of the chapters included in this thesis are based on papers which are the result of collaborative work. The below details my involvement in each of these works.

Chapter 3: StackArmor: Comprehensive Protection from Stack-Based Memory Error Vulnerabilities for Binaries *StackArmor* is joint work with Xi Chen, who is the first author and main developer of the final version. My contribution consists of implementing a full early version of our approach using PEBIL [114], upon which the final version is also based. In addition, I implemented several optimizations for PEBIL and a custom static code injection tool (`elfinject`) used in *StackArmor* to inject instrumentation code. I also drafted the first version of our paper.

Chapter 4: Practical Context-Sensitive Control-Flow Integrity I share the first authorship on *PathArmor* with Victor van der Veen. As described in Chapter 4, *PathArmor* consists of two main components: (1) A kernel module which handles LBR management and hooks the necessary system calls and functions, and (2) A verification module, which verifies the validity of LBR paths based on the Control-Flow Graph. Victor is the main author of the kernel module, while I am the main author of the verification module. We contributed equally to the evaluation and paper.

Chapter 5: Parallax: Implicit Binary Code Integrity Verification Using Return-Oriented Programming I performed all work in designing and implementing the *Parallax* system, as well as writing the paper describing it.

Chapter 6: An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries I performed the majority of the work in designing and executing our experiments, and all work in writing the paper. Xi Chen ran the Windows-based experiments, while Victor van der Veen assisted with reading and categorizing papers for our literature review.

Chapter 7: Compiler-Agnostic Function Detection in Binaries I performed all work in designing and implementing the *Nucleus* system, as well as writing the paper describing it.

Summary

Disassembly is the process of identifying code in binary programs, and translating it into a form fit for human analysis or further processing. It is a crucial step in virtually all forms of binary analysis, including malware analysis and security techniques for binaries. The need for securing legacy and proprietary binaries becomes ever more pressing as attackers develop new exploitation techniques which threaten unhardened binaries. Often, recompilation is not an option, as source code (or even symbols) may not be available; this leaves us with no choice but to use binary-level techniques. Unfortunately, disassembly is an undecidable problem, meaning that any system that operates on non-trivial binaries is bound to run into incomplete or erroneous disassembly.

This thesis explores methods for safely implementing binary-level hardening techniques using imperfect disassembly as a basis. We develop novel defenses against several forms of advanced attacks, including stack-based attacks, control-flow hijacking attacks, and tampering attacks on a hostile host. These defenses implement several strategies which allow us to make a balanced tradeoff between the level of security, overhead, and crash-safety in protected binaries.

Moreover, we perform an in-depth analysis of the disassembly process itself on the x86/x86-64 platform, highlighting the most error-prone cases and uncovering discrepancies between disassembly accuracy in practice, and widespread expectations on disassembly accuracy in the literature. In doing so, we provide a more stable foundation for future disassembly-based research, clarifying where problems are most likely to occur and special measures for ensuring correctness are thus the most necessary. Based on our analysis, we also implement improved methods for function detection; the most error-prone disassembly primitive at the time of writing.

Samenvatting

Disassembly is het identificeren van machinecode in binaire programma's, en het vertalen hiervan in een vorm die geschikt is voor analyse door mensen, of verdere verwerking door een computer. Het is een cruciale stap in vrijwel alle vormen van binaire analyse, waaronder malware-analyse en beveiligingstechnieken voor binaire programma's. Het belang van beveiliging voor bestaande binaire programma's groeit naarmate nieuwe exploitatietechnieken voor onbeschermde programma's worden ontwikkeld. Het is vaak geen optie om een programma opnieuw te compileren, omdat de broncode (of zelfs symbolische informatie) in veel gevallen niet meer beschikbaar is; in zulke gevallen is er dus geen andere keus dan het direct toepassen van binaire beveiligingstechnieken. Helaas is disassembly een onbeslisbaar probleem, waardoor ieder systeem dat niet-triviale binaire programma's verwerkt met zekerheid te maken krijgt met incomplete of foutieve disassembly.

In dit proefschrift onderzoeken we methodes voor het veilig implementeren van binaire beveiligingstechnieken met imperfecte disassembly als basis. We ontwikkelen nieuwe verdedigingsmechanismes tegen meerdere geavanceerde aanvallen, waaronder aanvallen op de stack, zogenaamde "control-flow hijacking" aanvallen, en aanvallen waarbij de integriteit van het programma zelf bedreigd wordt. Onze verdedigingsmechanismes implementeren een aantal strategieën die het mogelijk maken een gebalanceerde afweging te maken tussen het gewenste beveiligingsniveau, de resulterende vertraging van het programma, en de mate van waarschijnlijkheid van crashes door disassemblyfouten.

Daarnaast bevat dit proefschrift een gedetailleerd onderzoek naar het disassemblyproces zelf op het x86/x86-64-platform, waarbij aandacht wordt besteed aan de meest foutgevoelige gevallen, en waarbij we afwijkingen constateren tussen de betrouwbaarheid van disassembly in de praktijk, en de verwachtingen betreffende deze betrouwbaarheid in de wetenschappelijke literatuur. Hiermee ontwikkelen we een stabielere basis voor toekomstig onderzoek, door te verduidelijken welke problemen het meest waarschijnlijk zijn, en dus speciale aandacht behoeven. Gebaseerd op onze analyse implementeren we ook een verbeterde methode voor functiedetectie, hetgeen op het moment van schrijven de meest onbetrouwbare component van het disassemblyproces is.