Techniques For Efficient Binary-Level Coverage Analysis

Techniken für eine effiziente Analyse der Codeabdeckung auf binärer Ebene

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurswissenschaften (Dr.-Ing.)
genehmigte Dissertation

von

M.Sc. Mohamed Ammar Ben Khadra

Geb. in Damaskus, Syrien

D 386

Dekan	:	Prof. Dr. rer. nat. Marco Rahm
Vorsitzender der Prüfungskommission	:	Prof. Dr. Gerhard Fohler
Gutachter	:	Prof. Dr.-Ing. habil. Wolfgang Kunz Univ.-Prof. Dr. Daniel Große
Tag der Einreichung	:	13.01.2021
Tag der Disputation	:	21.05.2021

Dedicated to Rasha. My wife, closest friend, and life companion …

Declaration of Authorship

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Mohamed Ammar Ben Khadra
21 May 2021

Acknowledgements

I have several people to thank for making my Ph.D. journey both successful and enjoyable. The following is a incomplete attempt at this.

I am deeply grateful to my advisors Prof. Dominik Stoffel and Prof. Wolfgang Kunz for the opportunity to conduct this research. I want to thank them for the freedom they have given me to work on a variety of problems and for nudging me to keep this work on track. I would like to especially thank Wolfgang for his patience and continued support. Additionally, I am most thankful to Prof. Daniel Große for accepting to be part of the reviewer committee.

My time in Kaiserslautern would not have been the same without the camaraderie among the members of the Electronic Design Automation group including, in no particular order, Thomas Fehmel, Paulius Morkūnas, Stian Sørensen, Christian Bartsch, Anna Duque Antón, Tobias Ludwig, Michael Schwarz, and Johannes Müller. Thank you guys for the cheerful times. Also, I would like to thank our administrative staff, namely, Carmen Vicente-Fess, Matthias Legrom and Andreas Christmann for always being their when I needed help.

I would also like to thank Prof. Sanjit A. Seshia and Dr. Susmit Jha for accepting to invite me to the Dagstuhl Seminar 17351: "Machine Learning and Formal Methods." Several interesting talks and thoughtful discussions have helped crystallize my ideas around sliced microexecution, the jump table analysis technique presented in this work.

Finally, I am grateful to my parents; your constant prayers have indeed helped! Also, I would not have endured this long journey without the enthusiasm and encouragement of my wife, Rasha. She deserves a lot of credit for this work. Besides, I edited the majority of this work at home during the uncertain times of the COVID-19 pandemic. So I also have to thank our two kids for being a joy to stay with under lockdown.

This dissertation was typeset using a template adapted from the LaTeX template authored by Krishna Kumar and others which can be found here: `https://github.com/kks32/phd-thesis-template/`.

Abstract

Code coverage analysis plays an important role in the software testing process. More recently, the remarkable effectiveness of coverage feedback has triggered a broad interest in feedback-guided fuzzing. In this work, we discuss static instrumentation techniques for binary-level coverage analysis without compiler support. We show that the proposed techniques are precise, efficient, and transparent significantly beyond the state of the art.

We implement these techniques into two tools, namely, Spedi and bcov. Both tools are open source and publicly available. Spedi shows that the disassembly and function identification of stripped binaries can be highly accurate without resort to any external information. We build on these important results in bcov where we statically instrument x86-64 ELF binaries to track code coverage. However, improving efficiency and scaling to large real-world software required an orchestrated effort combining several techniques.

First, we bring a well-known probe pruning technique, for the first time, to binary-level instrumentation and effectively leverage its notion of *superblocks* to reduce overhead. Second, we introduce sliced microexecution, a robust technique for jump table analysis which improves CFG precision and enables us to instrument jump table entries. Additionally, smaller instructions in x86-64 pose a challenge for inserting detours. To address this challenge, we aggressively exploit padding bytes. Also, we introduce a greedy scheme to systematically host detours in neighboring basic blocks.

We evaluate bcov on a corpus of 95 binaries compiled from eight popular and well-tested packages like FFmpeg and LLVM. Two instrumentation policies, with different edge-level precision, are used to patch all functions in this corpus - over 1.6 million functions. Our precise policy has average performance and memory overheads of 14% and 22%, respectively. Instrumented binaries do not introduce any test regressions. The reported coverage is highly accurate with an average F-score of 99.86%. Finally, our jump table analysis is comparable to that of IDA Pro on gcc binaries and outperforms it on clang binaries.

Our work demonstrates that static instrumentation can offer unique advantages in comparison to established methods like compiler instrumentation and dynamic binary instrumentation. It also opens the door for many interesting applications of static instrumentation, which can go well beyond coverage analysis.

Table of contents

List of figures ix

List of tables xi

1 Introduction 1

1.1 Overview 1
1.2 The Importance of Code Coverage Analysis 3
1.3 Code Coverage in Test Generation 6
1.4 Proposed Coverage Analysis Workflow 8
1.5 A Comparison of Coverage Analysis Tools 10
1.6 Contributions and Outline 12
1.7 Publication List 14

2 Speculative Disassembly 16

2.1 Overview 16
2.2 The Disassembly Problem 17
2.3 Proposed Method 20
  2.3.1 Recovery of maximal blocks 21
  2.3.2 Overlap conflict analysis 24
  2.3.3 CFG conflict analysis 26
2.4 Resilience to Obfuscation 27
2.5 Experiments 30
2.6 Discussion and Related Work 32

3 Function Identification 33

3.1 Overview 33
3.2 Symbol-based Identification 35
  3.2.1 Function layout 35
  3.2.2 Function entries 36

3.2.3 Function exits 38
3.3 CFG-based Identification 38
3.3.1 Function layout approximation 39
3.3.2 CFG traversal 40
3.4 Experiments 42
3.4.1 Function definitions in CFI records 43
3.4.2 CFG-based function identification 44
3.5 Discussion and Related Work 46

4 Jump Table Analysis 48
4.1 Overview 48
4.2 Sliced Microexecution 49
4.3 Experiments 53
4.3.1 Comparison with the state of the art 53
4.3.2 Distribution of jump tables 55
4.4 Discussion and Related Work 55

5 Probe Optimizations 58
5.1 Overview 58
5.2 Probe Pruning 58
5.3 Probe Pruning Alternatives 62
5.4 Optimized Probe Selection 63
5.5 Experiments 66
5.5.1 Probe count distribution 66
5.5.2 Probe type distribution 68
5.6 Discussion and Related Work 68

6 Static Instrumentation 71
6.1 Overview 71
6.2 Detour Hosting 71
6.3 ELF Patching 73
6.4 Experiments 77
6.4.1 Detour hosting effectiveness 77
6.4.2 Patching off-the-shelf binaries 79
6.5 Discussion and Related Work 81
6.5.1 Detour hosting 81
6.5.2 ELF patching 82

7 Tool Evaluation 84
7.1 Tool Implementation . . . 84
7.1.1 Implementation of Spedi . . . 84
7.1.2 Implementation of bcov . . . 85
7.2 Experimental Setup . . . 86
7.2.1 Experimental Setup of Spedi . . . 86
7.2.2 Experimental Setup of bcov . . . 86
7.3 Evaluation Results . . . 87
7.3.1 Scalability and transparency . . . 88
7.3.2 Instrumentation overhead . . . 88
7.3.3 Comparison with DBI tools . . . 90
7.3.4 Accuracy of the reported coverage . . . 91
7.4 Discussion and Limitations . . . 92

8 Conclusion and Outlook 94

9 Zusammenfassung 97
9.1 Zusammenfassung der Techniken . . . 98
9.1.1 Spekulative Disassemblierung . . . 98
9.1.2 Funktionsidentifikation . . . 98
9.1.3 Analyse der Sprungtabellen . . . 99
9.1.4 Optimierung der Probes . . . 99
9.1.5 Statische Instrumentierung . . . 99
9.2 Ausblick . . . 100

References 101

List of figures

1.1 General workflow of the modern software development life cycle 5
1.2 Our coverage analysis workflow 9
1.3 bcov patching example 10
1.4 Outline of our contributions 13

2.1 Describing linear sweep disassembly and its issues 19
2.2 Instruction classification used in recursive-descent disassembly 20
2.3 Disassembly of the function `_start` 20
2.4 Workflow of proposed speculative disassembly method 21
2.5 Overlap conflict example taken from du utility 27
2.6 CFG conflict example depicting three conflicting BBs 28
2.7 Sporious CTI example 29

3.1 Example of incorrect function identification 34
3.2 A script that dumps the function definition of a non-stripped binary 36
3.3 Example of `setjmp` usage 37
3.4 CFG-based function identification phases 41
3.5 Evaluating the completeness of CFI records 43
3.6 Evaluating CFG-based function identification 46

4.1 Example of a switch statement 49
4.2 Example of a jump table 51
4.3 Computed goto usage example 53
4.4 Normalized jump table analysis results in comparison to IDA Pro 54
4.5 Distribution of jump tables across our dataset 56

5.1 Deriving a superblock dominator graph from a CFG 60
5.2 Comparison between basic block pruning techniques 63
5.3 Distribution of probe count across our dataset 67

5.4 Distribution of probe types across our dataset 69

6.1 Detour hosting example 72
6.2 Typical layout of an ELF file 74
6.3 ELF header example 75
6.4 ELF patching types based on the location of segment header table 77
6.5 Evaluation of detour hosting 78
6.6 Distribution of the applicability of patching techniques 80

7.1 Architecture of Spedi tool 85
7.2 Comparing the code size of our subjects to objdump 88
7.3 Overhead of the any-node instrumentation policy 89
7.4 Comparing the performance overhead of bcov to DBI tools 90
7.5 Example of a may-return call 93

List of tables

1.1 A comparison with representative coverage analysis tools 11

2.1 Description of the benchmarks used to evaluate Spedi 30
2.2 Instruction recovery results of speculative disassembly 31

3.1 CTI comparison between x86-64 and ARMv7 38
3.2 CFG-based function identification results in comparison to IDA Pro. 45

4.1 Hypotheses tested, or falsified, to analyze a jump table 50
4.2 Expected behavior of a control-bounded jump table 52

5.1 BB type classification used in probe selection 64
5.2 Examples of patching different basic block types 65

7.1 Selected subjects used to evaluate bcov 87
7.2 Evaluating the accuracy of bcov based on drcov traces 92

Chapter 1

Introduction

1.1 Overview

The past few decades have witnessed tremendous progress in all aspects of computing and communication. The fast pace of change has profoundly affected our lives economically, socially, and maybe even politically. It led to what many describe as a revolution - an information revolution. The exponential progress of Moore's law means that tiny chipsets are now capable of running complex software that can be shipped in every "thing." In turn, ubiquitous communication has enabled such "things" to be highly interconnected in the massive network we call the Internet.

Indeed, modern technology has brought many opportunities as well as challenges. For practitioners, enabling the former is of equal importance as addressing the latter. In this context, the challenges of building systems that are functionally correct and secure are of a particular significance, as software becomes more complex, and the Internet expands relentlessly into the physical world. A software bug that flickers the screen during gameplay can be annoying. However, a similar bug can be far more consequential as it moves into the physical world, for example, by affecting the safety of a nuclear plant.

Recent years provide several examples of the potential real-world impact of software errors. The infamous 2003 blackout in North America was caused, in part, by a software bug in the alarm management system. The result is that a vast area covering the northeastern US and Ontario in Canada had to stay without electricity. The blackout lasted between 2 hours and 4 days, depending on location. The losses are estimated to range between $7 and $10 billion [134]. Fast forward to December 2015, we find another power blackout, albeit on a smaller scale in Ukraine. However, this blackout was caused by a deliberate cyberattack conducted by, presumably, state-sponsored actors. While the attack started with social engineering techniques exploiting human errors, it remains a sobering reminder of the potential role software bugs can play in the future of cyberwarfare.

Despite the importance of building software that is free of (critical) bugs, this still seems to be more of a Utopian dream than a practical goal. The following is our attempt at distilling the key challenges facing such an endeavor:

- **Complexity**. The number of bugs/defects in a piece of software is proportional to its complexity. Software complexity can be measured simply by the number of lines of code (LoC). Also, more sophisticated metrics are proposed in the literature, like McCabe's cyclomatic complexity [89] and Halstead's metrics [60]. However, regardless of the metric we use, modern software remains astonishingly complex, which renders the mere understanding of a software's functionality, let alone its validation, a monumental task. For example, the Linux kernel v1.0 started in 1991 with around 10 KLoC. Less than 20 years later, it now stands at about 28.5 MLoC in kernel v5.8 [54].

- **Rapid change**. Software is a fast-moving target. On average, ten commits of source code are merged into the Linux kernel every hour [54]. The merging of thousands of commits per day is a common practice in large organizations like Google [91] and Facebook [61]. This need for speed and adaptability has pushed agile methods to the forefront of software development, and created new engineering roles, like DevOps, to enable it. However, such pace puts a lot of pressure on the testing and validation processes to maintain software quality without significant slow downs.

- **Market dynamics**. Software engineering is driven by dynamics that are different from other engineering disciplines. First, while users may not tolerate bugs in safety-critical systems, they do tolerate some bugs in hardware and a lot more in software. Subsequently, it is perfectly acceptable for software vendors to regularly push bug-fixing updates. Second, users usually prefer software offering more features over one with better quality, especially since it is difficult for them to assess the latter. Third, software is often sold "as is" where vendors offer limited liability for the losses their software causes. Given these dynamics, compromising quality by taking on some form of technical debt might be unavoidable to stay competitive in the market [36, 2]. The challenge, however, is the long-term management of such technical debt.

- **Uncertainty and non-determinism**. Software engineering is founded on the principles that computing is deterministic with fully predictable behavior. We assume that an assignment like "x := 3" will set the value of x to 3, despite knowing that a hardware glitch can affect the result at any time. We invest in system reliability to maintain this illusion of certainty and determinism. This illusion has served us well through the years but it has become increasingly costly to maintain, as we build systems that deal with uncertainty in various ways, like machine learning and cyber-physical systems. This calls for uncertainty to be treated as a first-class concern in software testing [45, 48].

- **Abstraction leakage**. Modularity and hierarchical design are fundamental to managing the complexity of modern systems. In an ideal world, we would assume that such abstractions will hide the details of the subsystems they encapsulate. Unfortunately, for the implementation to be efficient, it is often infeasible to maintain perfect encapsulation, which may allow determined adversaries to leak information about the internal state of an encapsulated subsystem. This leakage can have serious security implications if the leaked information is sensitive. The recent microarchitectural side-channel attacks, as embodied in various variants of Meltdown [87] and Spectre [78], demonstrate the shaky ground upon which rest several security mechanisms that we take for granted like process isolation.

The importance of building correct and secure software was recognized early by the research community, which responded with many techniques spanning the whole spectrum of correctness guarantees from mathematically proving that programs are correct to effective debugging and automatic program repair. In the following section, we briefly overview these techniques and discuss the important role coverage analysis plays in the context of the software development life cycle.

1.2 The Importance of Code Coverage Analysis

Code coverage analysis is commonly used throughout the software testing process [3]. Structural coverage metrics such as statement and branch coverage can inspire confidence in a program under test (PUT), or at least identify untested code [67, 53]. Additionally, coverage analysis has demonstrated its usefulness in test suite reduction [130], fault localization [100], and detection of compiler bugs [83]. To understand why coverage analysis remains indispensable for software quality assurance despite its shortcomings [67], we need to briefly look at alternative methods.

To claim that a program is correct, we must put forward a convincing argument of its correctness. Ideally, in the form of a mathematical proof. Such a view was popular since the early days of computer science and continued to thrive well into the eighties. Edsger W. Dijkstra clearly stated that[1]: “The only effective way to raise the confidence level of a

---

[1] Quote from Edsger W. Dijkstra’s The Humble Programmer (1972).

program significantly is to give a convincing proof of its correctness." The logical foundation for rigorously reasoning about programs, and subsequently proofing their properties, was laid in the landmark paper of C.A.R. Hoare [65].

However, it became increasingly evident to the research community that proofs cannot play in computer science the same role they play in mathematics, as the complexity of the specification and constant modifications will render the maintenance of such proofs a very costly endeavor [38]. For example, Klein et al. [77] estimate that the formal proof of the seL4 microkernel, which consists of about 9 KLoC of C code only, required 8 person-years. This estimation assumes that the task is conducted by team of experts in formal methods. Additionally, it is acknowledged that a small modification in the source code can invalidate a large part of the proof. That is, the effort of re-verifying the system after a small change is highly disproportionate. It is generally difficult to justify such investments except in safety-critical systems or maybe small subsystems of critical importance, such as the floating-point unit of a processor [63].

The highly influential 1982 paper of Clarke and Emerson [32] has marked the birth of model checking as an alternative method for rigorously reasoning about computing systems. Given a formal specification of a property, a model checker considers all possible states of a system to check whether the given property holds or, if it does not, provide a counter-example. This automated workflow proved to be much more practical compared to theorem proving. However, model checking comes with its own challenges, the main of which is the well-known state explosion problem, where an exponential number of states need to be considered in large systems. Several techniques were proposed to address this challenge, including symbolic model checking using BDDs [27], bounded model checking with SAT solvers [18], and property-directed reachability (PDR) analysis [23].

Model checking represents a fundamental technique for validating system *designs* ranging from cache-coherence protocols in processors [12] to fault-tolerant algorithms in massively distributed systems [95]. Additionally, model checking is widely used to verify hardware *implementations* in the form of RTL (or gate-netlist). Nowadays, all major Electronic Design Automation (EDA) tool vendors offer solutions for Formal Property Verification (FPV), where properties are typically written in SystemVerilog, an IEEE standard language for describing and verifying hardware. It is worth noting that even before FPV, formal equivalence checking, for example, between optimized and unoptimized gate-netlists, has enjoyed quick adoption by the industry. This adoption was motivated by the strong results of academic works like Kunz and Pradhan [80] and Jain et al. [70].

Unfortunately, and despite significant progress [71], the success of model checking in verifying software systems remains much less impressive compared to their hardware coun-

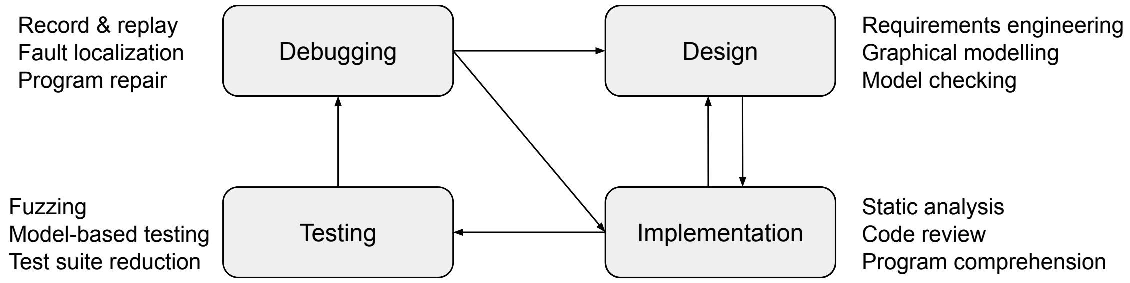


Fig. 1.1: General workflow of the modern software development life cycle. A cycle iteration can be at unit or system levels. The former may happen several times daily. Common methods for increasing software quality are depicted beside their corresponding phase.

terparts. This discrepancy can be attributed to several factors. First, software typically has an infinite state space where precise analysis is known to be undecidable in general. Second, software is modified at a much faster pace than hardware, which renders the maintenance of property specifications prohibitively expensive. Third, the market dynamics that govern software are drastically different, as discussed in the previous section.

It is evident that for software development to be economically viable, it has to embrace complexity and rapid change. This logically implies adopting an iterative process, which is commonly known as the software development life cycle (SDLC). Fig. 1.1 illustrates the main phases of this process. Note that a cycle iteration does not necessarily need to start with the design phase. In fact, developers often spend the majority of their time moving between implementation, testing, and debugging. Additionally, the design phase can benefit from the feedback gained through implementation. Moreover, iterations may be hierarchical involving a single unit (module) or the entire system.

Each phase of the SDLC represents an opportunity to apply a different set of techniques to enhance software quality. We show several examples in Fig. 1.1. It is not the goal of this section to survey the wide variety of techniques that are available in the literature. The interested reader can find a wealth of information with a simple web search. Instead, we want to emphasize that testing remains the main method by which we can enhance our confidence in the functionality of a piece of software. Testing is not perfect. It can cost a lot of resources and still miss bugs. However, it remains the most cost-effective method we have. Therefore, improvements to testing techniques and methodology are highly desirable.

Software testing revolves around three main questions: (1) is the software behavior exhibited by a test input correct? (2) can our test inputs effectively uncover bugs? (3) is the testing conducted so far sufficient? Code coverage analysis might be unable to offer much help with (1), which is also known as the test oracle problem. However, code coverage can assist in (2) by guiding automatic test generation, as we shall discuss in the next section. More importantly, coverage metrics are the main tool by which practitioners assess (3). That is, achieving a predetermined level of code coverage is typically a prerequisite to move between different testing levels, e.g., from unit to system, as well as for releasing software to customers. Additionally, particular code coverage requirements are mandated by the standards in safety-critical domains [43, 68]. For example, the aerospace guidance DO-178C requires that critical software components, whose failure can lead to a catastrophic result, must satisfy the highly stringent modified condition/decision coverage requirements (MC/DC).

We discussed in this section the important role that code coverage plays in software testing in general. Next, we elaborate on the role it plays in automatic test generation, a field that has received a lot of attention in recent years.

1.3 Code Coverage in Test Generation

Testing is an indispensable but costly factor in software development [17, 97]. Hence, it is natural to investigate techniques for test automation to reduce costs. In this respect, developers may resort to differential testing [90, 59], where they use a program P' as a testing oracle for program P that implements the same functionality. In such doing, it is possible, to a large extent, to avoid manually specifying whether the outcome of a test given to P is valid or not. Additionally, as software evolves, the size of its test suite and, subsequently, its execution time will only increase over time. Test suite reduction techniques [130] can help reduce the size of a test suite while maintaining its quality.

Among test automation techniques, automatic test generation (ATG) remains the area that received the most attention [97]. It is the process of automatically generating inputs that satisfy a given testing criterion. The problem is challenging since the generated inputs should explore the behavior of a program under test (PUT) effectively and efficiently. Additionally, developers typically have a long queue of issues they are aware of already. Hence, an ATG tool has to uncover issues that developers would deem *relevant*.

Based on the insight we have about the internal structure of the program, we can classify ATG to black-box, white-box, and grey-box testing. The former is driven by observing the inputs and outputs of a PUT only. However, in white-box testing, we analyze the internals of a PUT, e.g., paths in its control-flow graph (CFG), in an attempt to generate more effective tests. The tremendous progress in SMT solvers, like Z3 [39] and CVC [11], has brought symbolic execution [76] to the forefront of white-box test generation with several notable tools, such as KLEE [28] and SAGE [52]. On the other hand, in grey-box testing, we collect lightweight feedback from a PUT and use it to guide further input generation. In this way, it is less “blind” compared to black-box testing and, at the same time, more lightweight compared to white-box testing.

Feedback-guided fuzzing combines lightweight feedback with pseudo-random test generation. In recent years, it emerged as a successful method for automatically discovering software bugs and security vulnerabilities [105, 111, 131, 22]. Notably, the tool American Fuzzy Lop (or simply AFL) [132] has pioneered the usage of code overage as a generic and effective feedback signal. This success inspired a fuzzing "renaissance" and helped move fuzzing to industrial-scale adoption like in Google's OSS-Fuzz platform [98]. Generally, feedback-guided fuzzing consists of a simple workflow where we (1) generate a test-case, (2) run an instrumented version of the PUT to collect feedback, and (3) evaluate program feedback and schedule the next execution. This loop continues until the allocated time budget of the fuzzing campaign is reached. Typically, the goal is to maximize the number of (unique) crashes found.

To this end, balancing the trade-off between test *efficiency* and *effectiveness* is essential [21]. Ideally, fuzzers strive to reduce the time needed per fuzz cycle (efficiency) while exploring newer program behavior in each test (effectiveness). In practice, however, effective techniques for test generation (e.g., symbolic execution) and behavior monitoring (e.g., ASan) can be expensive in terms of efficiency. Consequently, recent fuzzers tend to be judicious in leveraging this effectiveness, e.g., QSYM [131] and REDQUEEN [7]. To carefully balance this trade-off, we have to develop three key policies: (1) *scheduling policy*, which selects and prioritizes seeds (test inputs) in the seed queue, (2) *test-generation policy*, which determines how to generate new test cases, and (3) *instrumentation policy*, which decides how to instrument a PUT and what feedback signals to collect.

Scheduling and test generation policies have received considerable attention. For instance, formulating the seed scheduling problem as Markov chains [22] and multi-armed bandits [128]. Additionally, the traditional boundary between mutation-based and grammar-based test generation is becoming increasingly blurry as both techniques are fused in state-of-the-art fuzzers [19, 101, 123]. Note that exploring program structure might require collecting various feedback signals beyond mere coverage such as progress in comparisons [84].

On the other hand, the implementation of instrumentation policies has been so far largely limited to dynamic binary instrumentation (DBI) and compiler instrumentation. Both options have some inherent drawbacks. DBI considerably slows down the entire PUT even if the required feedback affects only a select set of functions. In comparison, compiler instrumentation is more efficient. However, it requires access to source code, which might not be available, and changes to the build toolchain, which can be intrusive and cause conflicts with build optimizations [69].

These limitations are reflected in the instrumentation options available in AFL. AFL users typically opt for compiler instrumentation, if possible, as DBI can incur more than 5x performance overhead [132]. AFL tracks edge coverage by instrumenting every basic block with code to update a shared hash map. The size of this map is 64 kB in order to fit in an L2 cache. Despite this tuning, an average performance overhead of 36% is expected [94]. Generally, this overhead is unnecessarily incurred for all tests, even though the percentage of coverage-increasing tests quickly drops over time [94, 7]. This highlights the importance of being able to adapt the instrumentation policy dynamically. Yet another issue is that collisions in the hash map of AFL increase in large binaries, which reduces the quality of coverage feedback [47].

In this work, we show that static binary instrumentation (SBI) can provide a better alternative to DBI and compiler instrumentation. Specifically, our approach *drastically improves efficiency across key aspects of the fuzzing workflow, namely, instrumentation overhead, merging (and comparing) coverage data, and policy adaptability.* The latter property enables further overhead reduction, eventually in a fuzzing campaign, by focusing on coverage increasing inputs only [94]. Moreover, we allow fuzzers to dynamically adapt as they learn more about the program structure so that, for example, they can exclude functions that are "uninteresting" from coverage tracking. In this work, we focus on basic block coverage, which is an empirically proven feedback signal [115]. However, the proposed techniques can be extended in principle to collect various other feedback signals.

1.4 Proposed Coverage Analysis Workflow

Figure 1.2 depicts the workflow of bcov, the tool into which we implemented the majority of our work. Given a binary module as input, bcov first analyzes module-level artifacts, such as the call graph, before moving to function-level analyses to build the CFG and dominator graphs. Currently, bcov accepts binaries in the popular Executable and Link Format (ELF). Then, bcov will choose appropriate probe locations and estimate the required code and data sizes depending on the *instrumentation policy* selected by the user. Our prototype supports two instrumentation policies. The first is a *complete* coverage policy where, for *any* test input, it is possible to precisely identify covered BBs. The second one is a *heuristic* coverage policy where we probe only the leaf superblocks (SBs) in the superblock dominator graph. Running a test suite that covers *all* leaf SBs implies that we reached 100% code coverage. We refer to these policies as *any-node* and *leaf-node* policies, respectively. In our evaluation, the any-node policy probes 46% of BBs on average compared to 30% in the leaf-node policy. Average performance overheads are 14% and 8%, respectively.

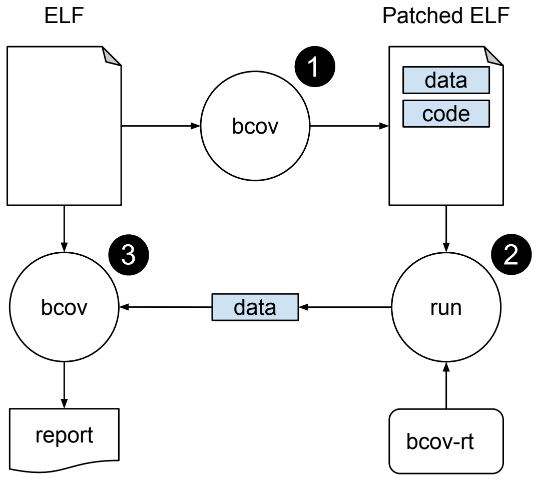


Fig. 1.2: Coverage analysis workflow of bcov. An ELF binary is patched with extra code segment (trampolines) and data segment (coverage data). Our bcov-rt library dumps the data segment at run-time. In our prototype, reporting coverage requires re-analyzing the binary.

The patching phase can start after completing the previous analysis phase. Here, bcov first extends the ELF module by allocating two loadable segments: a code segment where trampolines are written and a data segment for storing coverage data. Then, bcov iterates over all probes identified by the selected instrumentation policy. Each probe represents a single SB. Generally, patching a probe requires inserting a detour targeting its corresponding trampoline. The detour can be a pc-relative `jmp` or `call` instruction. The trampoline first updates coverage data and then restores control flow to its state in the original module as depicted in Figure 1.3.

The data segment has a simple format consisting of a small header and a byte array that is initialized to zeros. Setting a byte to one indicates that its corresponding SB is covered. It is trivial to compress this data on disk as only the LSB of each byte is used. For example, this enables storing complete coverage data of `llc` (LLVM backend) in 65 KB only. [2] Our data format also enables merging coverage data of multiple tests using a simple bitwise OR operation. Dumping coverage data requires linking against bcov-rt, our small runtime library. Alternatively, bcov-rt can be injected using the LD_PRELOAD mechanism to avoid modifying the build system. Coverage data can be dumped on process shutdown or upon receiving a user signal. The latter enables *online* coverage tracking of long-running processes. Note that the data segment starts with a magic number which allows bcov-rt to identify it.

This design makes bcov achieve three main goals, namely, transparency, performance, and flexibility. Program transparency is achieved by not modifying program stack, heap, nor any

---

[2] The binary has around 1×10^6 BBs which contain more than 4×10^6 instructions.

```
36b62: cmp  eax,0x140
36b67: sete al
36b6a: jmp  36bce
```

(a) original code

```
36b62: cmp  eax,0x140
36b67: jmp  6002b8
```

(b) patched code

```
6002b8: mov  BYTE PTR [rip+0xadd88],1
6002bf: sete al
6002c2: jmp  0x36bce
```

(c) trampoline

Fig. 1.3: bcov patching example. (a) instruction at 0x36b67 must be relocated as the size of jump at 0x36b6a is only two bytes. (b) relocated instructions are replaced with a 5 byte detour at 0x36b67. (c) coverage update happens at 0x6002b8. Control flow is then restored after executing the relocated instruction at 0x6002bf.

general-purpose register. Also, coverage update requires a single pc-relative `mov` instruction which has a modest performance overhead. Finally, bcov works directly on the binary without compiler support and largely without changes to the build system. This enables users to flexibly adapt their instrumentation policy without recompilation.

1.5 A Comparison of Coverage Analysis Tools

There is a plethora of tools dedicated to coverage analysis. They vary widely in terms of goals and features. Our approach is unique in that it tracks binary-level coverage via static instrumentation. Therefore, we motivate the need for our approach by comparing it to the approaches implemented in a representative set of popular tools.

Our discussion is based on Table 1.1. We start with source-level tools supported in gcc and clang, which are gcov and llvm-cov, respectively. Both track similar artifacts such as statement coverage. However, they differ in the performance of instrumented binaries. gcov cannot accurately track code coverage in optimized builds. In comparison, llvm-cov features a custom mapping format embedded in LLVM's intermediate representation (IR). This format allows it to cope better with compiler optimizations. Also, this format tracks various source code regions with better precision compared to gcov.

The ability of a binary-level tool, like bcov, to report source-level artifacts is limited by the binary-to-source mapping available. Off-the-shelf debug information can be used to report statement coverage - the most important artifact in practice [69, 53]. In this setting, bcov offers several advantages including: (1) detailed tracking of individual branch decisions regardless of the optimization level, (2) precise handling of non-local control flow constructs, such as `longjmp` and C++ exception handling, and (3) flexibility in instrumenting only a selected set

	Level	Coverage goal	Compiler independence	Performance overhead	Flexibility	Usability
gcov	source	complete	✗	✗	✗	✓
llvm-cov	source	complete	✗	✓	✗	✓
sancov	IR	heuristic	✗	n/a	✗	✓
Intel PT	binary	heuristic	✓	✓	✗	✗
drcov	binary	both	✓	✗	✓	✗
bcov	binary	both	✓	✓	✓	✓

Table 1.1: A comparison with representative coverage analysis tools. Compiler-dependent tools require modifying the build system and recompilation which limits flexibility. The usability of binary-level tools in the testing workflow is limited. In contrast, bcov is highly usable. It only requires replacing a binary with an instrumented version.

of functions, e.g., the ones affected by recent changes, which is important for the efficiency of continuous testing [69].

The recent fuzzing renaissance has motivated the need to improve efficiency by heuristically tracking coverage. SanitizerCoverage (sancov) [109] is a pass built into LLVM which supports collecting various types of feedback signals including basic block coverage. It is used in prominent fuzzers like LibFuzzer [85] and Honggfuzz [115]. The performance overhead of sancov is not directly measurable as the usage model varies significantly between sancov users. Also, sancov is tightly coupled with LLVM sanitizers (e.g., ASan) which add varying overhead. Extending bcov with additional feedback signals, similar to sancov, is an interesting future work.

Hardware instruction tracing mechanisms, like Intel® PT (IPT), can also be used for coverage analysis. However, IPT can dump gigabytes of compressed trace data within seconds which can be inefficient to store and post-process. In our experiments, IPT dumped 6.5 GB trace data for a libxerces test that lasted only 5 seconds. Post-processing and deduplication took more than 3 hours. In comparison, our tool can produce an accurate coverage report for the same test after processing a 53 KB dump in a few seconds. Schumilo et al. [111] propose to heuristically summarize IPT data on the fly and thus avoid storing the complete trace.

Dynamic binary instrumentation (DBI) tools can report binary-level coverage using dedicated clients (plug-ins) like drcov. DBI tools act as a process virtual machine that JIT-emits instructions to a designated code cache. This process is complex and may break binaries. Moreover, JIT optimizations add overhead to the whole program even if we are only interested in a selected part, such as a shared library. Our evaluation in Section 7.3.3 shows that bcov can provide drastic advantages in comparison to the popular DBI tools Pin [102] and DynamoRIO [24].

1.6 Contributions and Outline

In this dissertation, we propose several techniques for efficiently tracking code coverage of software at the binary level. We integrated the majority of our techniques into bcov, a tool for coverage analysis of x86-64 binaries in the ELF format. However, we separately implemented speculative disassembly and our CFG-based function identification technique in the tool Spedi, which supports ARMv7 Thumb-2 binaries only. Both tools are publicly available on the author's Github page: `https://github.com/abenkhadra`.

Fig. 1.4 depicts our contributions and fits them in the workflow of static binary analysis. Given an ELF binary, bcov uses traditional linear sweep disassembly to recover the instructions. It identifies functions using either linker symbols or Call Frame Information (CFI) records. However, this might not work in more adversarial settings, where linear sweep is not reliable, and function definitions do not exist. To address these challenges, we propose a speculative disassembly and CFG-based function techniques, both of which do not require access to external information. We discuss each of the subsequent contributions in a separate chapter. The outline of this work is as follows:

Chapter 2. bcov operates under two assumptions. First, linear sweep disassembly does not produce errors like recovering invalid instructions. Second, function definitions are available, for example, in linker symbols. These assumptions do not necessarily hold in stripped binaries. To address this challenge, we propose *speculative disassembly*, where we initially recover a superset of all possible basic blocks. Then, we refine this superset using *conflict analyses* to identify the most likely instructions. This disassembly technique is implemented into Spedi. Our experiments show that Spedi can outperform IDA Pro, the leading industry disassembler.

Chapter 3. The notion of a function is fundamental to many software analyses, including our proposed techniques, as it determines the scope of the control-flow graph (CFG). In this chapter, we discuss how to systematically identify functions in the case where linker symbols are available. Additionally, we show that Call Frame Information (CFI) records are comparable to linker symbols as a source of function definitions. This result has important implications since CFI records are often overlooked by researchers, despite being generally available in stripped binaries. Additionally, we propose a CFG-based function identification technique to address the use cases where CFI records are incomplete or even unavailable. This technique builds on the speculative disassembly technique of the previous chapter.

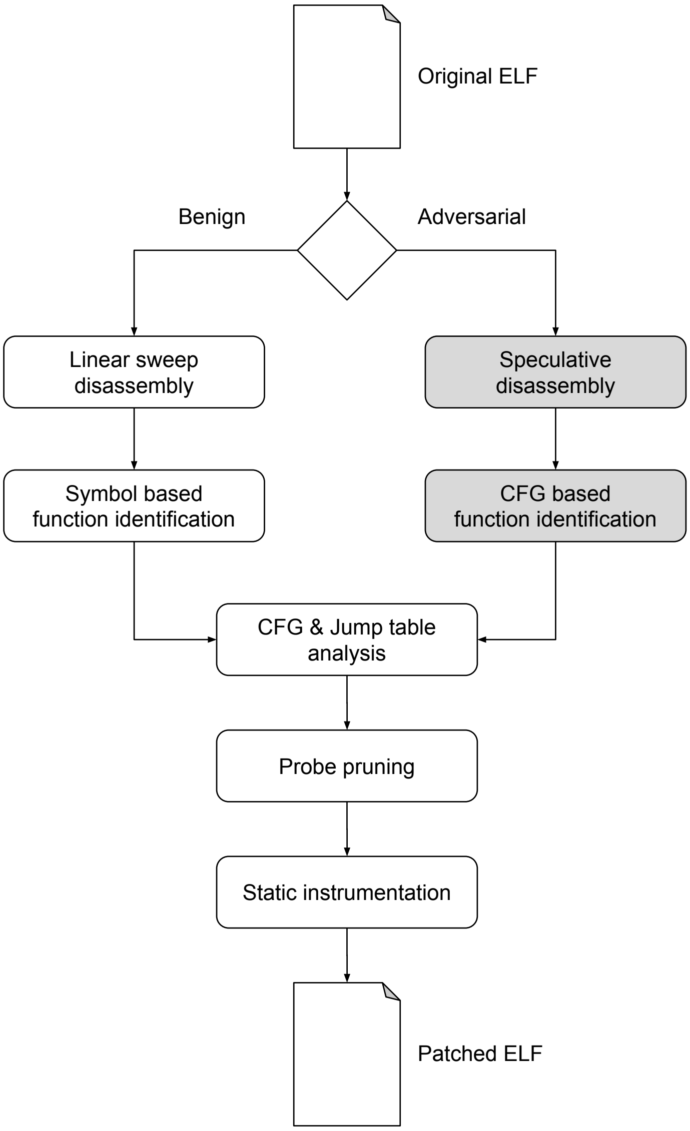


Fig. 1.4: Outline of our contributions that shows how they fit in the workflow of static binary analysis. In benign binaries, bcov relies on traditional linear sweep disassembly. However, we developed the highlighted techniques, which are implemented in Spedi, to deal with more adversarial settings where we cannot rely on external information, such as linker symbols.

Chapter 4. Imprecision in the recovered control flow graph (CFG) can cause several issues ranging from false positives in the reported coverage to crashes in the program under test (PUT) due to incorrect instrumentation. To achieve high precision, we propose *sliced microexecution*, a precise and robust technique for jump table analysis. Additionally, we use the results of this analysis to instrument jump table entries. We show thereby that data-only detours are achievable at scale. Our experiments show that bcov is comparable – or even outperforms – IDA Pro, the leading industry disassembler. Specifically, and unlike IDA Pro, we demonstrate that bcov is largely unaffected by changing compilers and optimization levels.

Chapter 5. To track code coverage, it is generally inefficient to instrument all basic blocks (BBs). To improve efficiency, we bring, for the first time to our knowledge, the probe pruning technique of Agrawal [1] to binary-level coverage analysis. Basically, we exploit the dominator relationships between BBs to group them into superblocks (SBs). Then, we arrange SBs in a superblock dominator graph. Based on this, covering a single BB in an SB implies that all of its dominators are also covered. Leveraging this dominator graph allow us to significantly reduce both the instrumentation overhead and size of coverage data. Additionally, we discuss an optimization technique to select a BB to instrument among possibly several BBs in the same SB.

Chapter 6. In a variable-length ISA like x86-64, the size of a significant percentage of BBs can be less the 5 bytes, which is too small to insert a detour without overwriting the following BB. In such a case, bcov would overwrite padding bytes, if possible. However, this can be insufficient. Leaving a small BB without instrumentation risks losing coverage information about its superblock along with all of its dominators. We overcome this problem by systematically hosting detours of small BBs in large neighboring BBs. We formulate this problem as a resource allocation problem for which we propose a greedy strategy. Additionally, we discuss in this chapter the techniques implemented in bcov to patch ELF binaries. We show that they generally apply to stripped off-the-shelf binaries.

Chapter 7. Each of the previous chapters includes experiments that evaluate their discussed techniques. In this chapter, we discuss the experimental setup that is common across all experiments. Also, we evaluate the tool, bcov, which efficiently integrates the majority of the proposed techniques. Our experiments are extensive, involving 95 real-world binaries and over 1.6 million functions. We separately applied two instrumentation policies, namely, the leaf-node and any-node policy. In the former, we instrument only the leaves of the superblock dominator graph. However, in the any-node policy, we additionally instrument *critical* superblocks. The key result is that the instrumentation is transparent by not introducing any test regressions. Also, the performance overhead is low with 8% and 14% for the leaf-node and any-node policy, respectively. We conclude by discussing further issues and future outlook.

1.7 Publication List

This dissertation is primarily based on the work published in our ESEC/FSE'20 paper [16]. However, the speculative disassembly method of Chapter 2 and CFG-based identification technique of Section 3.3 are based on our CASES'16 paper [14]. For convenience, we list these peer-reviewed papers here:

- **M. Ammar Ben Khadra**, Dominik Stoffel, and Wolfgang Kunz. 2020. Efficient Binary-Level Coverage Analysis. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE'20*. Virtual Event, USA.
- **M. Ammar Ben Khadra**, Dominik Stoffel, and Wolfgang Kunz. 2016. Speculative disassembly of binary code. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems - CASES'16*. Pittsburgh, PA, USA.

The road to discovering the techniques that are effective in practice never follows a straight path. The ideas we present in this dissertation have been indirectly influenced by the following works [13, 15], which we also conducted as part of this research:

- **M. Ammar Ben Khadra**, Dominik Stoffel, and Wolfgang Kunz. 2017. goSAT: Floating-point Satisfiability as Global Optimization. In *Proceedings of Formal Methods in Computer-Aided Design - FMCAD'17*. Vienna, Austria.
- **M. Ammar Ben Khadra**. 2017. E3Solver: decision tree unification by enumeration. preprint arXiv:1710.07021 (oct 2017).

Chapter 2

Speculative Disassembly

Overview; The Disassembly Problem; Proposed Method; Experiments; Discussion.

2.1 Overview

Starting from a given address in memory, the CPU continuously executes instructions in the familiar fetch-decode-execute cycle. This process *dynamically* skips data bytes potentially mixed with the code stream. However, our goal is *statically* analyze the behavior of machine code without necessarily executing it. To this end, we must begin with a disassembly technique to precisely recover machine instructions. The challenge, however, is that static disassembly is equivalent, in the general case, to the halting problem [66].

Fortunately, binaries encountered in practice are much less adversarial than what they could be in theory, which makes them amenable to automatic analyses. This is particularly the case for binaries generated by popular compilers. Additionally, given the need to keep the code modular and maintainable, assembly developers generally avoid writing arbitrary code unless there was a real necessity. Of course, software designed with obfuscation in mind, e.g., malware, would always prove challenging for automatic analyses.

Our coverage analysis tool, bcov, uses linear sweep disassembly to recover instructions in binaries. This method is fast and simple to implement. However, it depends on two key assumptions. First, jump table data is written to a separate data section. Second, code padding is implemented using nop instruction(s) instead of arbitrary data bytes, i.e., padding bytes can be correctly disassembled. These assumptions allow linear sweep disassembly to be reliable for x86-64 binaries [5]. This is typically the case, at least for the major compilers Visual Studio, clang, and gcc. Our experiments confirm this result for the latter two compilers.

However, our assumptions do not always hold. For example, we observed that the gcc compiler for the ARM® architecture still inlines jump table data in the code stream; A fact that is also noted by others [34]. Additionally, the rapidly increasing complexity of software forces engineers to depend on third-party libraries to accelerate product delivery. Static obfuscation techniques such as opaque predicates [93] and branch tables [86] might be employed by third-party developers to protect their intellectual property (IP). In effect, this would limit the ability to disassemble and verify such IPs. Finally, modern defenses against code *injection* attacks, like Address Space Layout Randomization (ASLR) and "write xor execute" (`W^X`), have forced attackers to rely on code *reuse* attacks such as return-oriented programming [112, 30] and its jump-oriented counterpart [20]. This raises the need for a disassembly method that can recover all existing code "gadgets" to assess the attack surface exposed to potential attackers.

These use cases call for a principled disassembly technique that is more robust than the commonly used linear sweep and recursive descent techniques. Our proposed solution is a speculative disassembly technique which consists of two phases. In the first phase, we recover *all* possible Basic Blocks (BBs) available in the binary. BBs that share the same control-transfer instruction (CTI) are grouped into one Maximal Block (MB). In the second phase, MBs are refined using *conflict* analyses in order to identify the most *likely* instructions.

Our proposed technique has several nice properties. First, it does not need to start from a given entry address, which makes it suitable for analyzing firmware blobs. Second, it is resilient to static obfuscation techniques, a useful property for auditing third-party IPs. Third, unlike a similar method proposed by Kruegel et al. [79], our method does not depend on the availability of function definitions, which makes it more practical.

In the next section, we will shed more light on the disassembly problem. Then, we discuss our speculative disassembly method in more detail.

2.2 The Disassembly Problem

The goal of disassembly is to precisely recover instructions from a given binary code. Disassembly is the fundamental first step for any static binary analysis (SBA), including the code coverage analysis considered in this work. There is a wide spectrum of SBA applications considered in the literature. For example, we can find formal verification [117, 110], static instrumentation [4, 133], equivalence checking [37, 122], and security analyses [8, 35]. Additionally, SBA can be essential even when the source code is available. Some analyses can be carried out only at the binary level, such as static timing analysis [126]. Finally, compilers might introduce bugs [83, 129] or unexpected side effects [8]. Such defects can only be detected by reasoning about the actual machine instructions instead of the source code.

Formally, we can describe the problem as follows: given a buffer of bytes B characterized by (st, sz) where st is the start address, and sz is its byte size. The goal is to recover the set I

of valid instructions. Typically, we would also have E which is the set of entry point addresses such that $\forall e \in E, st \leq e < st + sz$ holds. Executables usually have only one entry address. However, the set E would include all exposed functions in a shared library. An instruction is considered valid if it can possibly be reached when execution starts at an entry $e \in E$. Note that the reachability from e is a sufficient but not a necessary condition. Some code of interest in B might be unreachable. For example, it can be an artifact generated by the compiler or simply code bloat that remained after software refactoring.

The fundamental challenge in disassembly is identifying data inlined in the code steam, as data bytes may be decoded into spurious instructions. Additionally, variable-size ISAs like x86-64 allow for multiple interpretations of the same byte sequence. The interpretation depends on where we start disassembling. Similarly, some variants of RISC-like ISAs, e.g., Thumb[1] and TriCore, also allow the instruction size to be variable to increase code density. This variability means that spurious instructions may lead to missing valid ones.

The traditional algorithms of disassembly are *linear-sweep* and *recursive-descent*. In the former, disassembly starts at st and then follows the byte stream until the end of B. That is, for each disassembled instruction t_i, the disassembly of t_{i+1} starts at address $start(t_{i+1})$ such that,

$$start(t_{i+1}) = start(t_i) + size(t_i)$$

Linear sweep is fast and simple. However, it may produce a significant number of disassembly errors. In Fig. 2.1, for example, consider the byte sequence starting at `0xb992` where each assembly instruction is shown next to its start address. Linear sweep incorrectly disassembles the data bytes at `0xb998` and `0xb99a`. Upon reaching `0xb99c`, it disassembles the data to the instruction `rsb.w`, which has a size of 4 bytes. Consequently, disassembly would continue at `0xb9a0`, skipping the valid instruction at address `0xb99e`.

Inlined data is not reachable at run time. Hence, it does not affect the execution of the processor. Recursive descent disassembly tries to mimic this behavior by starting from e = `0xb992` and then following the CFG. CFG traversal is based on the classification of instructions depicted in Fig. 2.2. The key difference to linear sweep disassembly lays in the treatment of control transfer instructions (CTI). Basically, upon encountering a direct CTI, the target address will be added to E, a queue of disassembly addresses. In Fig. 2.1, for example, starting from e = `0xb992`, the target address `0xb99e` will be queued as the instruction happens to be also a conditional CTI. However, upon reaching the indirect CTI at `0xb996`, the algorithm

---

[1] We use the term Thumb to refer to Thumb-2 ISA, which is the variable size extension to Thumb-1 introduced in ARMv7.

```
b992: 1b b1 | cbz    r3, b99e        | cbz    r3, b99e
b994: 01 99 | ldr    r1, [sp, #4]    | ldr    r1, [sp, #4]
b996: 10 47 | bx     r2              | bx     r2
b998: 48 b0 | .data                  | add    sp, #0x120
b99a: 02 00 | .data                  | movs   r2, r0
b99c: cb eb | .data                  | rsb.w  fp, fp, r1
b99e: 01 0b | lsrs   r1, r0, #0xc    |
b9a0: 39 46 | mov    r1, r7          | mov    r1, r7
b9a2: 04 22 | movs   r2, #4          | movs   r2, #4
b9a4: af e7 | b      b906            | b      b906
```

Fig. 2.1: Comparing correct disassembly (left) to linear sweep disassembly (right) for a sample ARM Thumb code. Disassembling data bytes leads to recovering spurious instructions at addresses `0xb998`, `0xb99a`, and `0xb99c`. The size of latter instruction is 4 bytes which causes the instruction at `0xb99e` to be missed.

would stop following that path since it can not know the possible run time values of `r2`. Then, the algorithm would pop `0xb99e` from the queue and follow it to continue the disassembly.

Recursive descent disassembly is correct as long as the encountered direct CTIs are sufficient to explore the binary code. However, a significant part of the code may be reachable only through indirect CTIs (ICTIs), which leads to low code coverage. In practice, disassemblers in the industry, e.g., IDA Pro [64], combine both methods, linear sweep and recursive descent, with custom heuristics to achieve high code coverage.

For example, simply identifying the entry of the function `main` requires using a heuristic. ELF executables start execution at the function `_start`, which in turn indirectly calls `main` through the function `__libc_start_main`. The address of `main` will be set as the first parameter to function `__libc_start_main` in the register `rdi`. This makes identifying the address of `main` relatively easy in position-dependent executables, as depicted in Fig. 2.3. Unfortunately, or maybe fortunately from a security perspective, executables in modern Linux distributions, including Android, are position-independent by default. Therefore, identifying the run-time value of `rdi`, which holds the `main` entry, becomes significantly more difficult.

Symbol information, e.g., linker symbols, can assist in correct disassembly where we add symbol addresses to E, the queue of entry addresses. However, symbols might be unavailable. And if available, they might be unreliable. Therefore, disassemblers in safety-critical applications, like the work of Theiling [118], rely on pattern matching of known compiler idioms. This assumes that the compiler used to produce the binary, or maybe even the exact compiler version, is known a priori. Ideally, a disassembly technique must be generic and

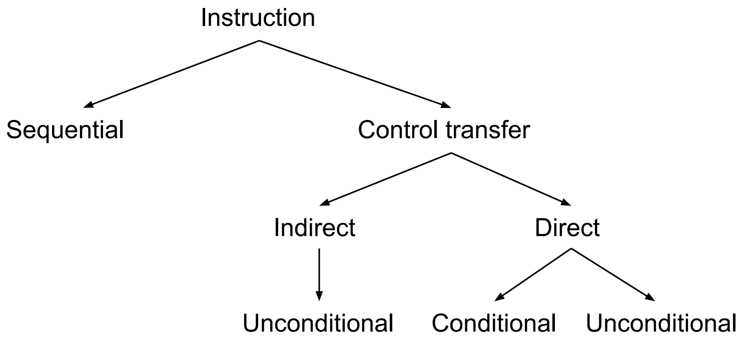


Fig. 2.2: Instruction classification used in recursive-descent disassembly. The targets of direct CTIs are collected in a queue of entry points. Typically, conditional instructions do not have an indirect form. Disassembly does not continue after indirect CTIs.

```
00000000004056d0 <_start>:
4056d0:  xor   ebp,ebp
4056d2:  mov   r9,rdx
4056d5:  pop   rsi
4056d6:  mov   rdx,rsp
4056d9:  and   rsp,0xfffffffffffffff0
4056dd:  push  rax
4056de:  push  rsp
4056df:  mov   r8,0x4eae60
4056e6:  mov   rcx,0x4eadf0
4056ed:  mov   rdi,0x403eb0   ; sets the address of main
4056f4:  call  403340 <__libc_start_main@plt>
4056f9:  hlt
```

Fig. 2.3: Disassembly of the function `_start` taken from an x86-64 executable compiled with gcc-7.4. Luckily, this particular binary is position dependent. The address of the function `main` is a constant value `0x403eb0` stored in the register `rdi`.

does not demand such assumptions about the binary. This represents the goal of our proposed disassembly technique.

2.3 Proposed Method

The workflow of our method is depicted in Fig. 2.4. It accepts an ELF binary as input. Then, it recovers the set of all potential basic blocks in the code. A basic block (BB) is a sequence of instructions that ends with a CTI. A Maximal Block (MB) is a set of all overlapping BBs such that each BB has a unique entry address but shares the same CTI with the other BBs. A CFG connecting MBs is then built based on encountered direct CTIs. Then, this CFG is used in the overlap and CFG conflict analyses to identify valid instructions.

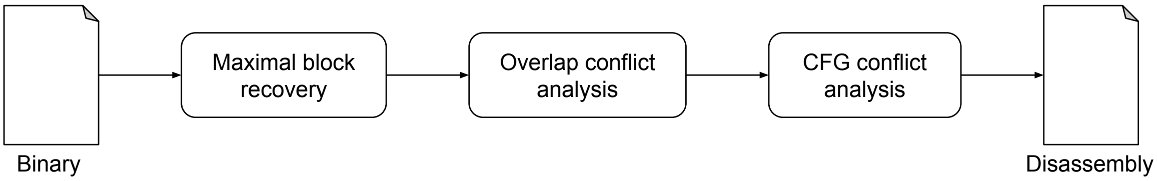


Fig. 2.4: Workflow of proposed speculative disassembly method. The method consists of two phases (1) speculatively recovering all BBs which are grouped in Maximal Blocks, and (2) conflict analyses used for BB refinement.

Our proposed disassembly method provides *provably* complete code coverage. Also, our experiments show that the proposed conflict analyses correctly identify valid instructions in practice. This method leverages the following key insights:

- Data bytes can only exist after CTIs and before the beginning of valid BBs. That is, they cannot exist, by definition, within the instruction sequence of a valid BB.
- More importantly, valid BBs are more connected to the CFG compared to spurious BBs generated by speculative disassembly. This gives them more “weight” that we leverage to resolve conflicts with spurious BBs.

We discuss this method in more detail in the remaining part of this section.

2.3.1 Recovery of maximal blocks

In this step, our goal is to determine all potential BBs available in a binary, i.e., to recover a superset of valid BBs. To this end, we start disassembly at the buffer start st, and continue until the end of B. This is similar to linear sweep except that we speculatively attempt to disassemble every possible instruction. In Thumb ISA, this would be every 2 bytes. In comparison, every byte in x86 may represent the start of an instruction. Unlike recursive-descent disassembly, we do not queue any addresses in E. Instead, we append every disassembled instruction to one or more BBs, if applicable. Otherwise, we would create a new BB for that instruction. BBs that meet at the same CTI are grouped in the same Maximal Block (MB). An MB is a convenient container for all BBs that exhibit the same control-flow behavior.

Going back to Fig. 2.1. We group the BBs starting at the addresses `0xb99c` and `0xb99e` in single MB, which ends with the CTI at `0xb9a4`. This means that we have two BBs sharing the same MB. They start with different instructions but share the remaining instructions starting from `0xb9a0`. We continue our discussion after introducing some necessary notation. For an instruction t its end address is,

$$end(t) = start(t) + size(t)$$

Note that $start(t)$ is unique for each t. Also, for a basic block bb we have,

$$end(bb) = start(bb) + size(bb)$$

where $start(bb) = start(t_0)$ such that t_0 is the first instruction in bb. Similarly, $end(bb) = end(t_i)$ where t_i is the last instruction in bb. Also, $size(bb)$ is the total size of a BB, which is sum of the size of its instructions. We say that t can append bb iff $end(bb) = start(t)$. For a maximal block mb, the sets T_{mb} and BB_{mb} represent the sets of instructions and basic blocks it contains, respectively. Also, $end(mb) = end(t_{cti})$ where t_{cti} is its unique CTI. We say that a maximal block mb_{i+1} can append mb_i iff:

$$\exists bb \in BB_{mb_{i+1}}, end(mb_i) = start(bb)$$

Maximal blocks are recovered using Algorithm 1. The discussion focuses on the Thumb ISA as it is the one supported by `Spedi`, our tool prototype. Basically, we decode every possible instruction t (line #4), where the step size is two bytes for Thumb ISA (line #14). Then, we check if t is acceptable. An instruction is acceptable if (1) its bytes are disassemble to valid ISA instruction, and (2) it does not violate ISA rules. For example, there are certain restrictions on the usage of register `pc` in ARM ISA manuals that, if violated, can make such instruction unexecutable, i.e., it causes the cpu to raise an exception. BBs containing such instructions can be safely discarded at this early stage.

Then, we check whether t represents a CTI (#line 6). If it does not, we search the current MB looking for possible BBs that can be appended by t (line #11). If none is found, a new BB will be created containing only t. However, if t is a CTI, then it would be appended to a BB like any other instruction. However, an MB has to be constructed now that at least one BB, which contains the CTI instruction, is complete.

Basically, BBs contained in the current MB are classified into two main categories (1) complete BBs which can be appended by the current CTI, or (2) incomplete BBs which hold a sequence of instructions but without a CTI. Note that a complete BB may contain only one CTI. Incomplete BBs are further classified into two sets, (1) invalid BBs, and (2) overlap BBs. Overlap BBs are kept since they can potentially be appended by an instruction in the next MB. For a BB to be regarded as an overlap BB it should satisfy the following:

$$end(mb) - end(bb) < S_{max}$$

where S_{max} is the maximum size of an instruction in the current ISA. It is 4 bytes in Thumb, and 15 bytes in x86 ISA. We keep all overlap BBs for the next MB (#line 9) and discard invalid BBs. After all, a valid BB must end with a CTI.

Algorithm 1 has linear complexity in terms of buffer size $O(|B|)$. Classification of BBs to complete, invalid, or overlap (line #9) is linear in $O(|BB_{mb}|)$. However, in our current implementation, mapping instructions to basic blocks is $O(|BB_{mb}| * |T_{mb}|)$.

Theorem 1 *The set of basic blocks recovered in Algorithm 1 is a superset of valid basic blocks existing in buffer B.*

Proof. Algorithm 1 advances with a step size the is minimal according to current ISA (line #14). Therefore, it disassembles every possible instruction. Let bb be a valid basic block. Then, bb can either consist of a single t_{cti}. In such a case, it is recovered directly in a maximal block. Otherwise, bb consists of T_{bb}, a sequence of instructions, that ends with a t_{cti} where $\forall t_i, t_{i+1} \in T_{bb}, end(t_i) = start(t_{i+1})$. Algorithm 1 recovers all instruction sequences (line #11) by (1) appending each disassembled instruction t to an existing sequence, or (2) constructing a new sequence T_{bb} containing only t. However, upon encountering a CTI instruction, then either (1) the CTI appends the sequence T_{bb}, which means that the current mb is now complete (line #8), or (2) it is a spurious t_{cti}, which means that T_{bb} can be retained for the next maximal block (line #9). In both cases, each mb would contain a superset of valid BBs that end with the same CTI. □

Having recovered Γ_{mb}, the list of MBs, we build the CFG Γ_{cfg} using direct CTIs only, as shown in Algorithm 2. Basically, it is a single pass over Γ_{mb} where an edge is added between mb and mb_r, if the latter is reachable from the former using a direct CTI (line #5). Another edge should be added in conditional CTI (line #8), if mb_i can append mb. Finding a remote target (lines #4) returns a result only if the target MB contains an instruction with the designated start address, otherwise, the procedure fails. This is similar for the procedure of finding an immediate target (line #7). Failing to find a target means that the current MB, and all of its direct predecessors, should be discarded since they do not target a valid BB. This is based on the fact that Algorithm 1 guarantees the recovery of all valid BBs. However, in the analysis of dynamically linked executables, we allow an MB to target an external ELF section, e.g., `.plt` section, but we make sure that the targeted section is at least executable. After all, valid control flow can not target a section holding data, e.g., the section `.rodata`.

Algorithm 2 is linear in $|\Gamma_{mb}|$. Finding an immediate target (line #7) is typically $O(1)$ as it is the immediate successor of an mb in $|\Gamma_{mb}|$. The most costly operation is the binary search required to find a remote MB (line #4). In this search, we take advantage of the ordering found in $|\Gamma_{mb}|$ which can be stated as:

Algorithm 1: Maximal block recovery

```
   Input  : Byte buffer to be disassembled $B$
   Output : List of maximal blocks $\Gamma_{mb}$
1  $addr$ := $st$
2  $mb$ := create_maximal_block()
3  while addr < end(B) do
4      inst := decode_inst_at($addr$)
5      if acceptable(inst) then
6          if is_cti(inst) then
7              append($mb$, $inst$)
8              add_to_result($\Gamma_{mb}$, $mb$)
9              $mb$ := get_overlaping_basic_blocks($mb$)
10         else
11             $mb$:= create_or_append($mb$, inst)
12         end
13     end
14     $addr$ := $addr$ + 2
15 end
```

Theorem 2 *Let mb_i and mb_j be maximal blocks in Γ_{mb} such that $i < j$ then $end(mb_i) \leq end(mb_j)$.*

Proof. Consider how MBs are recovered in Algorithm 1. Specifically, consider how new MBs are added to $|\Gamma_{mb}|$ (line #8). Each MB ends with a single CTI where $end(mb) = end(t_{cti})$. Given that no two CTIs can start at the same address, then we have $start(t^i_{cti}) < start(t^j_{cti})$ because instructions are disassembled in ascending order starting from a_{st}. This leads to $end(t^i_{cti}) \leq end(t^j_{cti})$ where the last 2 bytes of t^j_{cti} may overlap with t^i_{cti}. □

2.3.2 Overlap conflict analysis

In Algorithm 1, the current maximal block (mb) may retain overlap BBs (line #9). An overlap BB might be appended with a CTI in the next MB to form a complete BB. In this case, we say that both MBs overlap. Formally, for mb_i and mb_j to overlap where $i < j$ the following holds,

$$start(mb_j) < end(mb_i)$$

This situation creates a conflict that should be resolved because valid instructions cannot overlap, in general. To resolve such conflict, we can either (1) invalidate mb_i altogether, or (2)

Algorithm 2: Build direct CFG connecting MBs

```
Input : List of maximal blocks $\Gamma_{mb}$
Output: Direct CFG $\Gamma_{cfg}$
foreach $mb$ in $\Gamma_{mb}$ do
    $inst$ := get_cti($mb$)
    if is_direct_cti($inst$) then
        $mb_r$ = find_remote_target($inst$)
        add_edge($\Gamma_{cfg}$, mb, $mb_r$)
        if is_conditional_cti($inst$) then
            $mb_i$ = find_immediate_target($inst$)
            add_edge($\Gamma_{cfg}$, mb, $mb_i$)
        end
    end
end
```

shrink the size of mb_j to fit mb_i. Shrinking mb_j means increasing its start address such that $end(mb_i) \leq start(mb_j)$. To resolve an overlap, we take into account the following heuristics:

- Alignment. If mb_j can be appended to some mb_k, with $k < i$, then it is more reasonable to invalidate mb_i than to shrink mb_j.
- Connectedness. If mb_i is "more connected" to the CFG than mb_j, then it is more reasonable to shrink mb_j than to invalidate mb_i.

To leverage both heuristics, we rely on the "weight" of an MB. In our implementation, the weight of a particular MB is the sum of its instruction count and the instruction count of its immediate predecessors in the CFG. Note that it is possible to additionally use the weight of CFG successors, but we found our heuristic to be adequate in practice.

Algorithm 3 implements our overlap analysis. For each mb, we look ahead to check if it overlaps with its successors (line #2). Should an overlap mb_o exist, we first try to shrink its size (line #4). If shrinking is inapplicable, we have to invalidate either mb or mb_o based on their weight. Note that mb_o is shrinkable w.r.t. mb iff,

$$\exists t \in T^{o}_{mb} \; : \; end(mb) \leq start(t)$$

In other words, an instruction t that starts after mb ends should exist in mb_o. If mb_o is shrinkable, then we set its start address to the first t that satisfies the above relation. Shrinking mb_o effectively invalidates some instructions at its beginning, which is not always applicable. For example, another mb_k in Γ_{cfg} could be targeting one of those would-be invalidated

Algorithm 3: Overlap conflict resolution

```
Input  : CFG $\Gamma_{cfg}$
Output : Modified CFG $\Gamma_{cfg}$
foreach $mb$ in $\Gamma_{cfg}$ do
    if has_overlap($mb$) then
        $mb_o$ = get_overlap_mb($mb$)
        if is_shrinkable($mb_o$) then
            shrink_if_applicable($mb_o$)
        else
            invalidate_either($mb$, $mb_o$)
        end
    end
end
```

instructions. This leads to a conflict between mb and mb_k. Again, such conflict is resolved using their weights.

We simplify our discussion of overlap analysis with the help of the example depicted in Fig. 2.5. It is taken from `du`, one of the utilities found in GNU's Coreutils, and shows the instructions of 4 maximal blocks. We annotate each instruction with the index of its corresponding maximal block. The example has two overlap conflicts in total. The first is between mb_0 and mb_1 where,

$$end(mb_0) = \texttt{0x2cb12} < start(mb_1) = \texttt{0x2cb0e}$$

The second conflict is between mb_2 and mb_3 where,

$$end(mb_2) = \texttt{0x2cb1c} < start(mb_3) = \texttt{0x2cb1a}$$

Resolving the first conflict is done by shrinking mb_1 by invalidating the instruction at `0x2cb0e`. As for the second conflict, it is not possible to shrink mb_3 because it consists of a single instruction. Consequently, mb_3 will be invalidated.

2.3.3 CFG conflict analysis

The goal of overlap conflict analysis is to ensure that each MB occupies an area that does not overlap with other MBs. After resolving such *inter-MB* conflicts, we discuss here how to resolve *intra-MB* conflicts between different BBs. Basically, each BB starts at a unique start address within its MB. Therefore, only one BB might be a valid BB, while others are spurious.

```
2cb08 : c2 f1 | rsb.w_0 r2, r2, 1f |                          | rsb.w_0 r2, r2, 1f
2cb0a : 1f 02 |                    |                          |
2cb0c : 20 fa | lsr.w_0 r0, r0, r2 |                          | lsr.w_0 r0, r0, r2
2cb0e : 02 f0 |                    | and_1  r7, r2, f0000000  |
2cb10 : 70 47 | bx_0    lr         |                          | bx_0    lr
2cb12 : 08 b1 |                    | cbz_1  r0, 2cb18         | cbz_1   r0, 2cb18
2cb14 : 4f f0 | mov.w_2 r0, #-1    |                          | mov.w_2 r0, #-1
2cb16 : ff 30 |                    |                          |
2cb18 : 00 f0 | b.w_2   2cf0c      |                          | b.w_2   2cf0c
2cb1a : f8 b9 |                    | cbnz_3 r0, 2cb5c         |
2cb1c : 00 29 | cmp_4   r0, #0     |                          | cmp_4   r0, #0
2cb1e : f8 d0 | beq_4   2cb12      |                          | beq_4   2cb12
```

Fig. 2.5: Overlap conflict example taken from `du` utility. Each instruction is subscripted with the number of its corresponding maximal block. The disassembly after resolving overlap conflicts is shown on the right.

Consider the MB depicted in Fig. 2.6, which ends with a conditional CTI at `0x9df8`. It consists of 3 BBs starting at addresses `0x9dec`, `0x9df0`, and `0x9df4`, respectively. Each instruction is annotated with the BB(s) it belongs to. For example, bb_0 and bb_1 meet at the instruction `0x9df2`, and henceforth share all the following instructions.

Let us assume that overlap analysis sets the start address to `0x9dec`. However, as far as overlap analysis is concerned, any address greater or equal to `0x9dec` would be acceptable. For example, choosing to start at `0x9df0` means that instructions at `0x9dec`, `0x9dee`, and `0x9df4` have to be invalidated since they do not belong to the valid BB. Invalidating an instruction would also mean invalidating any MBs that could be targeting it in Γ_{cfg}. Such a conflict is resolved by weighing each BB and choosing the BB with the highest weight. The weight of a BB is calculated similarly to the case of MB. It is the sum of its instruction count together with the instruction count of its direct predecessors. In our example, BB_0 starting at `0x9dec` has the largest number of instructions. Additionally, basic blocks BB_1 and BB_2 have no direct predecessors. Therefore, BB_0 represents the most reasonable choice to resolve the conflict.

2.4 Resilience to Obfuscation

Obfuscation is any transformation applied to a program to make it harder to reverse-engineer. Obfuscation can be applied at source-code level (or similar) to make algorithms difficult to reverse engineer. Additionally, it can also be applied to the binary code to make it difficult to

```
9dec : 10 b5 | push_0         {r4, lr}  |
9dee : 41 f2 | mov.w_0        r4, 1268  |
9df0 : 68 24 |                          | movs_1  r4, 68
9df2 : c0 f2 | movt_{0,1}     r4,#3     |
9df4 : 03 04 |                          | lsls_2  r3, [r4]
9df6 : 23 78 | ldrb_{0,1,2}   r3, [r4]  |
9df8 : 1b b9 | cbnz_{0,1,2}   r3, 9e02  |
```

Fig. 2.6: CFG conflict example depicting three conflicting BBs in the same MB. Each instruction is annotated with basic block(s) it belongs to.

disassemble. Our focus is on the latter techniques that are applied statically, i.e., without run-time code generation or modification. Static obfuscation techniques try to violate assumptions that normally hold in benign binaries. Specifically, the key violated assumptions are: (1) control flow returns to the instruction that immediately follows a call, and (2) either branch of a conditional CTI can be taken.

Linn et al. [86] implemented an effective obfuscator that violates assumption #1 by replacing direct calls with *branch subroutines*. A branch subroutine would indirectly call the original function at run-time, which complicates the recovery of the call graph. Moser et al. [93] explored violating assumption #2 by introducing *opaque constants* that are provably difficult to statically analyze. Such constants can be used to build an opaque *predicate*, which is a conditional CTI that *always* branches to only one of its targets. In this way, junk bytes can be inserted, for example, at the address of the fallthrough target to confuse static disassemblers.

Kruegel et al. [79] used speculative disassembly to build an effective x86 disassembler tailored towards the obfuscator of Linn et al. [86]. This disassembler can also handle opaque predicates to a significant extent. Our speculative disassembly method is inspired by the work of Kruegel et al. [79]. We share the basic ideas of disassembling all possible instructions and refinement through conflict analyses. Hence, we provide comparable resilience to obfuscation. However, there are several differences between both methods, which we discuss in the following:

Data structures. In the work of Kruegel et al., each instruction belongs to a single *fragment*, a sequence of instructions that does not necessarily end with a CTI. In comparison, in our method, each instruction belongs to a single Maximal Block (MB). Because of the self-repairing property of x86 disassembly [108], a fragment would normally contain 1–3 instructions. This leads to producing many fragments and makes conflict analysis less efficient.

Function identification. Due to the inefficiencies in data structures, they demand function entries (or their approximation) to be first identified using an external method. Then,

```
237ae: ite      le
237b0: movle    r7, r0
237b2: bgt.w    229f8
237b6: .invalid
```

Fig. 2.7: Spurious `ite` produced by speculatively disassembly leads to a spurious conditional CTI at `0x237b2` followed by invalid instruction. Technique of Kruegel et al. would invalidated the whole fragment.

they apply their speculative disassembly technique to each function individually. Our method does not impose such restriction. Instead, we recover functions after completing instruction recovery, as discussed later in Section 3.3.

Statistical analysis. Kreugel et al. speculatively disassemble gaps between functions in a separate step. Here, they use bigram statistical learning in order to weight instructions. Basically, every pair of instructions that appears more frequently together in their learning corpus, will have more weight. This weight is used to calculate the total weight of a fragment, and resolve conflicts. In comparison, we uniformly disassemble the whole code in one pass, and we do not use statistical analyses.

Conflict analysis. They resolve conflicts in one step by assigning a weight to each fragment and selecting fragments with the highest weight. A fragment's weight is the number of its direct predecessors but sometimes the successors maybe used as well. In comparison, we do conflict analysis in two steps. First, we resolve conflicts among MBs in overlap analysis. Then, we resolve conflicts among BBs in the same MB in CFG conflict analysis. This provides better efficiency as our data structures are more coarse granular. It also provides flexibility in tackling subtle cases.

For example, consider the MB depicted in Fig. 2.7, which is taken from the GNU utility `du`. The CTI at `0x237b2` is conditional only because it happens to be the else case for the invalid `ite` instruction at `0x237ae`. Our overlap analysis can efficiently invalidate the `ite` instruction. On the other hand, the method of Kruegel et al. would invalidate the whole fragment (and its predecessors) since it conditionally branches at `0x237b2` into an invalid instruction.

To conclude, obfuscation and de-obfuscation are in a constant arms race. In speculative disassembly, we guarantee to recover all static BBs. Effective refinement of these BBs depends on how far an obfuscator can go. In principle, eliminating most direct CTIs is possible. This would adversely affect the conflict analyses used in our method, as well as those of Kruegel et al. However, this would also severely increase the performance overhead since indirect branches are costly. Additionally, it will still not be strong enough to deter more sophisticated dynamic analyses.

	# of Inst.	# of CTI	# of ICTI
`nnet`	13749	1942	413 (21.27%)
`sha`	14800	1843	413 (22.41%)
`parser`	17059	2916	561 (19.24%)
`loops`	23110	2759	492 (17.83%)
`zip`	26789	4057	848 (20.90%)
`expr`	30423	6728	813 (12.08%)
`ls`	30460	6640	963 (14.50%)
`csplit`	31342	6761	787 (11.64%)
`ptx`	34301	7634	913 (11.96%)
`du`	48809	10940	1380 (12.61%)

Table 2.1: Description of the benchmarks used to evaluate Spedi. Selected 5 largest binaries in `coreutils` and `coremark-pro` benchmarks. We also show instruction statistics. Indirect CTIs represent a significant fraction of total CTIs which complicates recursive-descent disassembly.

2.5 Experiments

We implemented our speculative disassembly method in the tool Spedi. The implementation details of this tool are highlighted in Section 7.1.1. The experimental setup is discussed in Section 7.2.1. An overview of our benchmark binaries is given in Table 2.1. We show the number of valid instructions in addition to indirect CTIs and the total number of CTIs. Note that ICTIs constitute a significant percentage of all CTIs. This indicates that pure recursive descent disassembly might not provide adequate coverage.

We identified valid instructions based on ARM code-mapping symbols. These symbols are available in the symbol table to classify code regions. The symbols are `a`, `t`, and `d`, which indicate are ARM, Thumb, and Data regions respectively. To collect the ground truth, we built a designated disassembly mode into Spedi to read these symbols. This ground truth is then validated against `objdump`, which can also use code mapping symbols, if available. The metric we use to evaluate instruction recovery is the valid instruction ratio (VIR) metric. It is the ratio of valid instructions, which are recovered by Spedi, compared to total number of valid instructions in the binary.

Our results are shown in Table 2.2. We compare VIR results by providing `objdump`, IDA, and Spedi with stripped binaries, i.e., without symbol information. `objdump` uses linear sweep disassembly in the absence of linker symbols. It produces a high number of disassembly

	objdump	IDA	Spedi	Time (ms)
`nnet`	49.37%	97.30%	99.94%	51
`sha`	50.13%	74.53%	99.95%	101
`parser`	50.00%	97.72%	99.95%	56
`loops`	49.98%	98.31%	99.96%	79
`zip`	50.08%	97.87%	99.97%	82
`expr`	50.47%	98.86%	99.97%	89
`ls`	54.00%	97.77%	99.97%	91
`csplit`	50.42%	98.78%	99.97%	94
`ptx`	50.47%	98.94%	99.97%	102
`du`	79.13%	98.18%	99.98%	146

Table 2.2: Instruction recovery results comparing objdump, IDA Pro, and Spedi. We also show the average time needed by Spedi to process the binary.

errors where it consistently achieves a VIR of about 51%. This renders any SBA based on it to be effectively useless. These poor results can be attributed to how `objdump` behaves upon encountering a disassembly error. It skips the default size of an ARM instruction (4 bytes), which causes disassembly errors to cascade. Instead, skipping only 2 bytes would have significantly improved the results of `objdump`. In comparison, IDA was in a better position by achieving about 98% on average. For unclear reasons, `sha` was the exception with 74.53%. Inspecting error causes, we noticed that IDA was skipping valid instructions even in sequence of correctly disassembled BBs. Move instructions, e.g., `movt` and `movs`, were particularly affected by such skipping.

On the other hand, Spedi demonstrates consistent results for all benchmarks. Actually, our main source of disassembly errors lays in the procedure `call_weak_fn`, which consists of 7 ARM instructions. Unfortunately, our prototype still does not support mixed ARM/Thumb disassembly. Note that achieving a VIR of 100 % does not mean that the disassembly method is perfect, as some data bytes can still be decoded as instructions. However, Spedi can already identify most data bytes generated for switch tables and PC-relative load instructions.

As for the scalability of Spedi, we show the average execution time of 10 runs. Our tool did scale gracefully with increased program size. The only exception was the `sha` benchmark with a relatively high run-time of 101ms. But this particular benchmark has very large MBs with thousands of instructions. This makes building BBs and attaching instructions to them expensive in our current prototype. Finally, note that the time required by Spedi cannot be directly compared to that of `objdump` or IDA. These tools implement very different functionality. For example, `objdump` uses a simple linear sweep, while IDA performs more analyses to build a disassembly database.

2.6 Discussion and Related Work

The disassembly problem was tackled by several works, either independently or as part of a proposed SBA technique. We will try in this section to cover some prominent works that do not rely on symbol information. Zhang et al. [133] used iterative linear sweep disassembly using `objdump` to implement their binary rewriter. Anand et al. [4] used recursive descent disassembly and speculative disassembled code areas that are unreachable by direct CTIs. Such areas would be translated to LLVM IR to be later refined, or simply rewritten back to the binary. Wartell et al. [125] used a speculative disassembly, but their instruction recovery method depends on statistical machine learning. Hence, it is more complex than our approach and prone to overfitting, similar to other machine learning techniques. The closest work to ours is that of Kruegel et al. [79], which we previously discussed in Section 2.4.

Kinder et al. [75] approached CFG recovery from an abstract interpretation perspective. They use recursive-descent disassembly until reaching an indirect CTI. Targets of indirect CTIs are safely overapproximated using a custom abstract domain. Their analysis is implemented on top of an Intermediate Representation (IR). In the case of encountering a loop, a significant part of the CFG needs to be reanalyzed, which is computationally expensive. Harris et al. [62] used recursive-descent disassembly combined with practical heuristics for function identification and CFG recovery. Theiling [118] used recursive-descent disassembly in addition to pattern-matching heuristics to detect common compiler idioms. The author identifies functions using a top-down approach, which applies only to statically-linked programs.

Compared to other works, our approach is more lightweight by not relying on an Intermediate Representation (IR) and expensive analyses. Also, leveraging the CFG allows for better robustness compared to approaches that use machine learning or custom heuristics. Moreover, our speculative disassembly method lends itself easily to parallelization. It allows each thread to independently process a memory region before merging results with the other threads. Lastly, for a fairer comparison with other works, it is interesting to extend our method to x86. We leave such an extension as future work.

Chapter 3

Function Identification

Overview; Symbol-based Identification; CFG-based Identification; Experiments; Discussion.

3.1 Overview

The notion of functions is important to our approach as it determines the scope of the CFG and, consequently, the correctness of dominance relationships between BBs. Functions are well-defined constructs in the source code. However, compiler optimizations, such as function inlining and splitting[1], significantly change the layout of corresponding binary-level functions. Fortunately, these optimizations are not of concern to us as long as *well-formed* function definitions are given to our coverage analysis tool, bcov.

Formally, a function is defined by the tuple $F = (st, sz, EN)$, where st and sz are its start address and size in bytes respectively. The set EN represents the basic block *entry* points where control flow enters the function. Similarly, we are also interested in EX, which is the set of exit points leaving a function. However, the majority of exit points are either tail calls or return to callers. Thus, we can automatically infer EX given the function definition F.

We say that the definition of function F is well-formed if (1) it does not overlap with other function definitions, i.e., the function occupies a dedicated code region, and (2) all of its basic blocks are reachable *only* through its entry points. For a set of function definitions, it is straightforward to check that it satisfies condition #1. However, checking the satisfaction of condition #2 is more complicated because of the existence of indirect CTIs. We might assume that EN comprises all entry points, only to find out later, at run time, that an indirect CTI is targeting an unidentified function entry. The fact is that the precise analysis of indirect CTIs, similar to other problems in static analysis, is an undecidable problem in general [81].

---

[1] Splitting is an optimization that can improve function inlining by dividing a function into multiple parts, making them easier to inline.

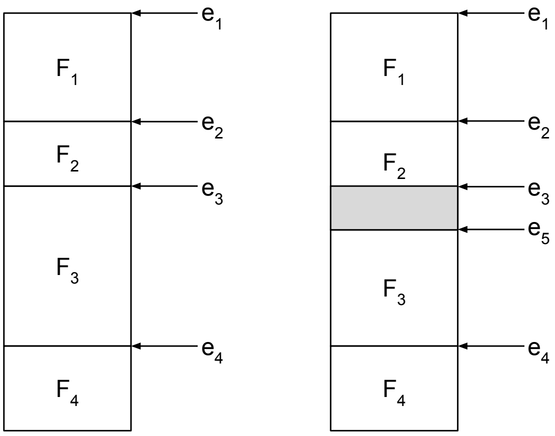


Fig. 3.1: Example of incorrect function identification. (left) correct function layout (right) an identification technique incorrectly assigned the region between e_3 and e_5 to F_2 instead of $F3$. In this case, e_3 is now an internal BB entry in F_2.

It is important to note that determining EN has a direct effect on the probe pruning techniques that we later discuss in Chapter 5. In the worst-case scenario, it is possible to add the entries of all BBs to EN. This would render probe pruning ineffective, as we have to instrument all BBs. In other words, over-approximation of EN increases the instrumentation overhead. On the other hand, under-approximation of EN may lead to more serious consequences, including crashes in the binary. For example, for a BB that is reachable only from another function, missing its entry from EN might lead us to incorrectly assume that it represents padding bytes. In effect, the misidentified BB might be overwritten, which leads to executing code with unintended consequences.

We discuss the issues associated with imprecise function identification based on Fig. 3.1. In a code region C, assume that the correct function layout consists of functions $F_1 \ldots F_4$ where, for the sake of simplicity, each function has single entry located at the function start. A function identification technique I_c applied on C was able to detect four functions. However, I_c misidentified the size of F_2 and, since F_2 must not overlap with F_3, slightly shifted the start of $F3$ to the new entry e_5. Starting from this function layout, I_c has to add e_3 to the set of entries of F_2, i.e., $EN_2 = \{e_2, e_3\}$. Compared to the actual functions entries, this solution comes at the expense of additional instrumentation overhead at e_5. Note that e_5 is presumably dominated by e_3. Therefore, in the correct function layout, the additional instrumentation probe at e_5 would have been pruned.

So what if I_c did not detect e_3? That is, the entry set of F_2 now contains $EN_2 = \{e_2\}$. Hence, we will not insert a probe in e_3. In this case, a call from F_4 targeting e_3 might go undetected by our coverage analysis tool. Even worse, the BB at e_3 might be incorrectly considered as padding since it is not targeted by any BB inside F_2. Recall that e_3 should have been assigned by

I_c to function F_3. Building on this incorrect assumption, our tool might incorrectly overwrite e_3 to exploit its area, which may lead to a program crash.

Our previous discussion shows how imprecision in function identification can have dramatic consequences on the correctness of instrumentation and the precision of the reported coverage. Identifying functions with high precision depends on the setting in which we conduct binary analysis. In the usual setting, we assume that linker symbols, or at least Call Frame Information (CFI) records, do exist and are trustworthy. Section 3.2 discusses how we leverage this information. However, in the case where this assumption does not hold, we can still resort to the CFG based identification technique of Section 3.3, which does not require access to external information.

3.2 Symbol-based Identification

We discuss how to identify the function layout under the assumption that linker symbols, or at least CFI records, are available. We collectively refer to this information as function symbols. Then, we build on that to identify function entries and exits.

3.2.1 Function layout

The process of compiling a C/C++ file goes through four main phases; First, the file needs to be pre-processed where macros and include headers are expanded to produce a compilation unit. Second, the compiler converts this compilation unit to an assembly file. Then, comes the assembler, which converts the latter file to a relocatable object file. Finally, the linker will takeover linking multiple object files into an ELF binary, be it an executable or shared library.

To merge multiple code and data sections, the linker relies on symbol tables embedded in the object files. These symbol tables must define, among other things, the layout of functions in the code section. Typically, these symbols remain in the final binary and are sufficient for identifying functions. Fig. 3.2 depicts a simple bash script that uses the popular `readelf` utility to dump the static functions available in a binary. Note that a binary's debug build provides more detailed information, including variable definitions. However, producing a debug build requires supplying additional compiler options like `-g`. Modifying the compiler options imposes a limitation that we like to avoid. Therefore, and by design, our techniques do not rely on debug information.

Coverage analysis is usually an internal activity within the organization that builds the software. Therefore, keeping linker symbols should not be difficult. However, the vast majority of commercial off-the-shelf (COTS) software is distributed in the form of stripped binaries

```
readelf -s binary | grep -P '\sFUNC\s' | \
gawk '{
  address = strtonum("0x"$2)
  size = strtonum($3)
  func_name = $8
  if (address != 0 && size != 0)
    print address,size,func_name
}'
```

Fig. 3.2: A script that dumps the function definition of a non-stripped binary. It skips dynamically linked functions, which are not of interest to us for the purpose of instrumentation.

to save space, or maybe to complicate reverse engineering. Moreover, some software builds instruct the linker to immediately strip symbols. We can achieve this in `gcc`, for example, by supplying the option `-s`. To accommodate these use cases, we turn to another source of information for identifying function layouts, namely, Call Frame Information (CFI) records.

CFI records are generally found in the `.eh_frame` section, which is part of the loadable image of an ELF binary, i.e., this section cannot be stripped. The `.eh_frame` section is similar in format to the `.debug_frame` section. However, the latter is part of debug information and is available only in debug builds. CFI records store the data necessary for stack unwinding. Hence, they must be available to enable C++ exception handling. However, CFI records are useful for several other purposes, such as crash reporting and performance profiling. Note that CFI records might not contain all the function definitions that are included in linker symbols. To save space, developers might decide to exclude CFI records of leaf functions. Nevertheless, our experiments show that CFI records provide a very large subset of the functions contained in symbol information.

3.2.2 Function entries

The main entry of a function is trivially defined by its start address. Other functions can either *call* or *tail-call* targeting only the main entry. We have empirically validated this assumption in our binary dataset (described in Section 7.2.2). That is, we have not found any instance where a (direct) call targets a basic block in another function. [2] Actually, the existence of such artifacts indicates that the given function definitions might not be precise.

However, non-local control transfer mechanisms, such as `longjmp` and exception handling, violate this assumption by branching indirectly to an internal BB. We consider the targets of non-local control transfer mechanisms to be *auxiliary* function entries. Such entries are not dominated by, or even unreachable from, the main function entry. Auxiliary entries of

---

[2] We found a few functions that have *intra-procedural* calls. They were found only in `libopencv_core`.

```
1 PL_warnhook = PERL_WARNHOOK_FATAL;
2 PL_diehook  = NULL;
3 JMPENV_PUSH(ret);
4
5 /* Effective $^W=1.  */
6 if ( ! (PL_dowarn & G_WARN_ALL_MASK))
7   PL_dowarn |= G_WARN_ON;
```

(a) source code

```
420089: mov QWORD PTR [rip+0x2be0cc],0x6d7680 # 6de160 <PL_warnhook>
420094: mov QWORD PTR [rip+0x2b7c71],0x0 # 6d7d10 <PL_diehook>
42009f: mov QWORD PTR [rsp+0x90],rax
4200a7: call 41fe20 <__sigsetjmp@plt>
4200ac: mov ebp,eax
4200ae: mov DWORD PTR [rsp+0x160],eax
4200b5: lea rax,[rsp+0x90]
4200bd: mov BYTE PTR [rsp+0x164],0x0
4200c5: mov QWORD PTR [rip+0x2b7bec],rax # 6d7cb8 <PL_top_env>
4200cc: mov ax,WORD PTR [rip+0x2b7dad] # 6d7e80 <PL_delaymagic>
4200d3: mov WORD PTR [rsp+0x166],ax
4200db: mov al,BYTE PTR [rip+0x2b7c9f] # 6d7d80 <PL_dowarn>
4200e1: test al,0x6
```

(b) machine code

Fig. 3.3: Example of `setjmp` usage in perl v5.28.1. **(a)** source code taken from function `S_gen_constant_list`. Call to `setjmp` happen in the macro at line #3. **(b)** corresponding assembly generated by `gcc` v7.3. Call to `setjmp` is located at `0x4200a7`. We consider the following address `0x4200ac` to be an auxiliary function entry.

`longjmp` are identified during CFG construction. They are simply the successor of each basic block that calls `setjmp`. Fig. 3.3 depicts an example of `setjmp`, as used in `perl` v5.28.1.

The identification of auxiliary entries of exception handling is more elaborate. The Itanium C++ ABI [55] specifies the exception handling standard used in modern Unix-like systems. Of interest to us in this specification is the *landing pad*, which is a code section responsible for catching, or cleaning up after, an exception. A function may have several landing pads, e.g., it catches exceptions of different types. We consider each landing pad to be an auxiliary entry. Collecting landing pad addresses requires bcov to iterate over all CFI records in the `.eh_frame` section. More specifically, bcov examines all Frame Description Entry (FDE) records looking for a pointer to a language-specific data area (LSDA). If such a pointer exists, then bcov would parse the corresponding LSDA to extract landing pad addresses. Note that LSDAs can generally be found in the `.gcc_except_table` section. We refer the reader to [56] for more details on LSDA format, and the role it plays in exception handling.

Type	**x86-64**	**ARMv7**
`call`	`call`	`bl/blx`
`jump`	`jmp`	`b/bx`
`return`	`ret`	`varies`

Table 3.1: CTI comparison between x86-64 and ARMv7. The latter allows a large number of instructions to directly modify the `pc` register which complicates the identification of returns.

3.2.3 Function exits

We analyze the CFG to identify its exit points, i.e., the basic blocks where control flow leaves a function. We consider two parameters: (1) the type of the control-transfer instruction (CTI), which can be `jmp`, `call`, or `return`, and (2) whether it is direct or indirect instruction. Table 3.1 maps the type of CTI used in this analysis to the actual instructions found in x86-64 and ARMv7 respectively. ARMv7 allows a large number of instructions to directly modify the `pc` register, which complicates the identification of returns, and control flow analyses in general. For example, a function may return by simply executing the instruction `mov pc, lr` where `lr` typically stores the return address. Alternatively, the same task can be achieved using a combination of `push lr` and `pop pc`.

A `jmp` targeting another function is a tail-call and generally also an exit point. However, the jump table analysis presented in Chapter 4 can determine that certain indirect `jmp`s are in fact intra-procedural, i.e., they are local to the function. On the other hand, a `call` typically returns, i.e, is not an exit point, except for calls to non-return functions. The non-return analysis implemented in our tool, bcov, is responsible for identifying such functions. Finally, we consider all `ret` instructions to be exit points.

3.3 CFG-based Identification

The symbol-based method discussed previously relies on assumptions that largely hold in practice. Developers will not find it difficult to keep linker symbols, or at least keep the part defining function layouts. Moreover, the experiments of Section 3.4.1 show that CFI records do exist in COTS binaries and are largely comparable to linker symbols in terms of completeness. This is expected since CFI records are required to implement essential functionality. For example, to assist in debugging by generating a stack backtrace after a program crash.

We present here a technique for function identification that relies solely on analyzing the CFG without recourse to any external information, not even to CFI records. There are several use cases for such a technique. For example, functions that do not participate in

Algorithm 4: Collect targets of direct calls

```
Input   : List of machine instructions $I$
Output  : List of direct call targets $T_{call}$
foreach $inst$ in $I$ do
    if is_call($inst$) and is_direct($inst$) then
        $t$ = get_target_address($inst$)
        add_target($T_{call}$, $t$)
    end
end
```

Algorithm 5: Collect targets of tail calls

```
Input   : List of machine instructions $I$
Input   : Sorted and de-duplicated list of call targets $T'_{call}$
Output  : List of tail-call targets $T_{tail}$
foreach $inst$ in $I$ do
    if is_jump($inst$) and is_direct($inst$) then
        $t$ = get_target_address($inst$)
        $c$ = get_current_address($inst$)
        ($e_{low}$ , $e_{high}$) = get_bounds($T'_{call}$, $c$ )
        if $t < e_{low}$ or $t > e_{high}$ then
            add_target($T_{tail}$, $t$)
        end
    end
end
```

stack unwinding might not have CFI records. Our CFG-based technique can complement the definitions found in CFI records to fill in these missing functions. Additionally, developers of C programs that do not use stack unwinding features, e.g., function `backtrace()`, might opt for omitting CFI records altogether to reduce the binary size. This can be accomplished in `gcc` by supplying the option `-fno-asynchronous-unwind-tables`.

Our proposed CFG-based technique consists of two main phases. First, we approximate the function layout by leveraging calls and tail-calls existing in the binary. Then, we improve this approximation via a CFG traversal which also helps to locate function exit points.

3.3.1 Function layout approximation

Starting from a disassembled binary, our first goal is to collect the targets of direct calls. These are the most obvious hints left behind by the compiler. The method used to disassemble the binary is irrelevant here. It can be a simple linear sweep or the more elaborate speculative disassembly method discussed in Chapter 2. This step is shown in Algorithm 4, where we collect the initial approximation of function entries in the set of targets T_{call}.

Algorithm 5 builds on that to identify tail calls and collect their targets in T_{tail}. Note that T_{call} is crucial for the identification of T_{tail}. In practice, the majority of jumps are intra-procedural, i.e., targets are within the same function. Therefore, we leverage our initial guess about function entries in T_{call} to distinguish between intra-procedural jumps and inter-procedural jumps (tail calls). Basically, we take two parameters into account to classify a particular jump instruction. They are the jump instruction's address $addr$, and its target t. We use $addr$ to determine the bounds of current function $[e_{low}, e_{high}]$ (line #5) where,

$$e_{low} = max\{e \in T_{call} : e < addr\}$$

and conversely,

$$e_{high} = min\{e \in T_{call} : e > addr\}$$

And then we check the target t against the function bounds $[e_{low}, e_{high}]$ (line #6). An out-of-bound target is definitely a tail call since it leaves the current function. On the other hand, a jump instruction with an in-bounds target is considered an intra-procedural jump, at least for this stage. Note that we used T'_{call} for the bound search in (line #5). This is simply to exploit binary search which is more efficient.

The end result of this function layout analysis phase is the set $T = T_{call} \cup T_{tail}$ which represents an overapproximation of function entries as depicted in Fig. 3.4. Next, we attempt to improve this approximation and complete function definitions. Note that we have so far recovered function entries only. That is, auxiliary entries and function exits are still missing.

3.3.2 CFG traversal

In the previous phase, we assumed that a jump with a target t that lays within the range $[e_{low}, e_{high}]$, is intra-procedural. This is usually, but not necessarily the case. In fact, multiple functions might exist in the region $[e_{low}, e_{high}]$. However, these functions might be called only indirectly and, therefore, their entries are still not discovered.

Our goal in this phase is to detect such functions and complete the function definitions. This is achieved using Algorithm 6. It starts from the approximated entries T, which we assume to be sorted and de-duplicated for better efficiency. Then, we analyze the region between current entry t and its immediate successor t'. First, we construct the basic blocks and connect them in a CFG. However, this step can be omitted if a CFG, which connects

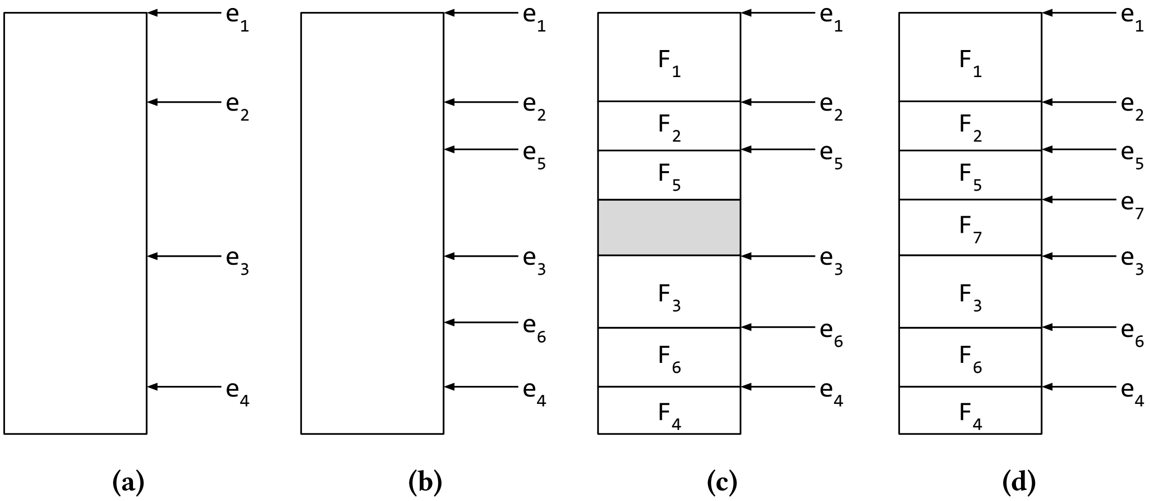


Fig. 3.4: CFG-based function identification phases. (a) collected the targets of direct calls, (b) collected the targets of tail-calls, namely, e_5 and e_6, (c) analyzing functions by CFG traversal of identified entries, (d) marked code regions are assigned to newly created functions such as F_7.

maximal blocks, was already constructed to enable speculative disassembly. Then comes CFG traversal, which is the core step of this phase.

We do a depth-first search (DFS) traversal starting from t, the function's main entry. DFS is more memory efficient compared to BFS. More importantly, it enables us to correctly analyze the exit points along the traversed program paths. For example, a function that pushes `lr` to specific location on the stack should use that same location to return to its caller. In other words, a `push` should match a `pop` on the same CFG path. Additionally, we analyze auxiliary entries. However, this analysis is limited to `setjmp/longjmp` entries. Recall that, unlike the symbol-based technique of Section 3.2, we do not assume the existence of CFI records. Accordingly, we do not assume that the binary uses standard exception handling mechanisms.

Starting only from the entry t, CFG traversal allows us to complete the function definition. This includes identifying the exit points and the byte size s. The latter can be simply calculated by $end(BB_l) - t$, where BB_l is the last basic block reachable by CFG traversal. The key assumption here is that for a function F with main entry t, all basic blocks must be reachable from the main entry. Generally, this assumption holds in C/C++ binaries generated by popular compilers like `clang` and `gcc`.

After recovering the definition of F, the following relation is expected to hold,

$$t + s + \epsilon >= t'$$

where t' is the start of the successor function, while ϵ represents the maximum size of the padding area, which may be used to align the start address of functions. Padding can improve

Algorithm 6: CFG traversal phase

```
Input  : List of machine instructions $I$
Input  : Approximated function entries $T$
Output : List of function definitions $\Delta$
Output : List of marked regions $M$
foreach $t$ in $T$ do
    $t'$ = get_next_target($T$, $t$)
    $\delta$ = build_function($I$, $t$, $t'$)
    add_function($\Delta$, $\delta$)
    $s$ = get_size($\delta$)
    if $t + s + \epsilon < t$ then
        add_marked_region($M$, $t + s$, $t'$)
    end
end
```

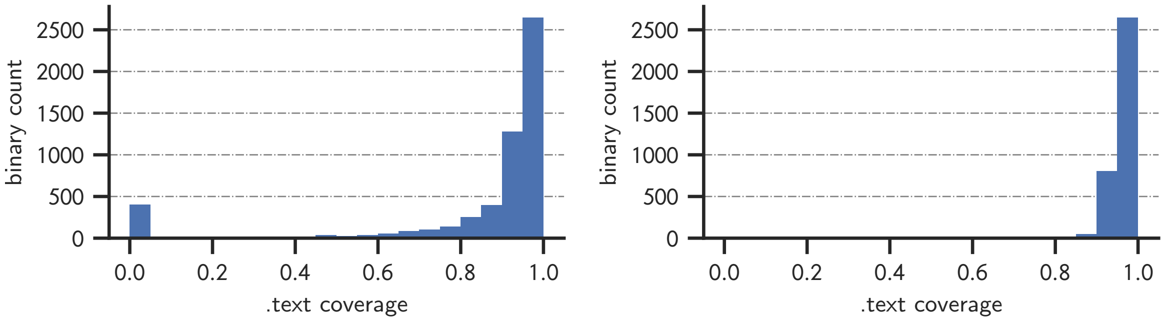
code cache locality, or maybe required to comply with ABI rules. For example, in x86-64 ABI, functions are aligned to a 16-byte boundary by default. Should the above relation not hold, then it is probably the case that there are more than one function in the region between $t + s$ and t'. This is the case Fig. 3.4, for example, where there exit two functions in the region between e_5 and e_3. Such code regions are marked for further processing. In our implementation, we simply assume that each marked region contains a single function like F_7 in our example. However, it is not difficult to imagine implementing the CFG traversal as a fix point analysis where a newly discovered region is marked in each iteration. The fix point would be reached after not discovering any new code regions.

Lastly, we consider the issue of analyzing calls to non-return functions, e.g, `abort`. During CFG traversal, we assume that each call generally returns to its caller. Hence, we continue CFG traversal after the call instruction. This might lead us to erroneously traverse into another function. We address this issue by identifying calls targeting common non-return functions like `abort` and `exit`. We identify these calls by analyzing the function addresses of the `.plt` section and mapping them to their corresponding function names.

3.4 Experiments

Identifying the main function entries is straightforward if linker symbols are available. In Section 3.2, we build on that to identify auxiliary entries as well as exit points. In this section, we focus on the following research questions:

- Do CFI records exist in COTS binaries? And if so, to what extent are they complete?



(a) All binaries **(b)** Relevant binaries

Fig. 3.5: Evaluating the completeness of CFI records in terms of coverage ratio relative to `.text` section. (a) distribution of all 5,538 binaries, (b) distribution of a subset consisting of 3,574 “relevant” binaries.

- Does our CFG-based identification technique improves upon the state-of-the-art?

We attempt to answer each question in the following discussions.

3.4.1 Function definitions in CFI records

To identify the main function entries, the symbol-based identification technique of Section 3.2 may use linker symbols or CFI records, with the latter being our only option in stripped COTS binaries. Therefore, to use CFI records as a source of function definitions, it is imperative to first evaluate their completeness.

First, we quantify to what extent CFI records are available. We conducted an experiment where we check for the existence of the `.eh_frame` section in x86-64 ELF binaries. Our subject is a typical Ubuntu 16.04 installation used by the author. Searching default executable and library paths like `/usr/bin` and `/lib`, we found a total of 6,852 binaries of which only 1.5% did not have an `.eh_frame` section. The majority of these latter binaries are related to the system packages klibc and syslinux. We repeated the same experiment on an Ubuntu 18.04 installation. We found 5,538 ELF binaries in total. This time *all* binaries did have an `.eh_frame` section. This indicates that CFI records are generally available in stripped off-the-shelf binaries.

We have established the availability of CFI records, but we still do not know the extent to which these records are complete. After all, an `eh_frame` section with only a handful of function definitions is not that useful. We measure the completeness of CFI records by the percentage of the area they cover in the `.text` section, the default code section in ELF binaries. Specifically, if the functions described in CFI records cover all the code, then there will be no need for linker symbols anymore. Also, this would mean that the function identification problem is essentially “solved” even for stripped binaries.

We stay with our latest dataset of 5,538 binaries found in an Ubuntu 18.04 installation. For each binary, we dump its CFI records using the command `readelf --debug-dump=frames`. Then, we sum the record's byte sizes. This sum will then be divided by the size of `.text` section to obtain our coverage ratio. Fig. 3.5 shows the distribution of coverage ratios across our dataset. It is noticeable here that about 400 binaries have a coverage ratio of less than 5%. Recall that the `.eh_frame` section is available in all of these binaries. The purpose of having an `.eh_frame` section with almost no CFI records is not clear though. On the other hand, we have a comfortable majority of about 4,000 binaries (72% of total) with a high coverage ratio of more than 90%.

Our dataset is comprehensive, but it is not necessary representative of the binaries we expect to analyze. In order to obtain a more representative dataset, we omitted system binaries found in the paths `/usr/lib/syslinux`, `/usr/lib/klibc`, and `/usr/lib/debug/`. We also omitted smaller binaries, which have a `.text` section size of less than 4KB. This results in a dataset consisting of 3,574 binaries that we deem to be "relevant". The coverage ratio distribution is now quite different as shown in Fig. 3.5b.

Admittedly, the relevance of a binary is a subjective criterion. Nevertheless, the overall trend is clear. CFI records are mostly complete in larger user-space binaries. Note that a coverage ratio of 90% does not necessarily mean that remaining 10% of the `.text` section represent missed functions. Padding bytes and other data inlined between functions are normally not accounted for in CFI records.

3.4.2 CFG-based function identification

The results presented in the previous section show that CFI records are usually complete. Despite that, there exist practical use cases where CFI records are either incomplete or missing altogether. Here comes the role of the CFG-based identification method, which we implemented in the tool Spedi. We highlight the implementation details of this tool in Section 7.1.1. The experimental setup is discussed in Section 7.2.1. An overview of our benchmark binaries was previously given in Table 2.1.

We measure the quality of function definitions recovered using Spedi by comparing them to those recovered using IDA Pro, the leading industry disassembler. We focus on the identified function boundaries represented by the start and end addresses. The ground truth is obtained from linker symbols and then cross-checked with IDA Pro on non-stripped binaries. This is needed to confirm that IDA Pro can correctly identify functions if linker symbols are available. Then, binaries were stripped of all symbols using the popular utility `strip` and given as input to both Spedi and IDA Pro.

	Orig.	IDA		Spedi	
		RF	PRF	RF	PRF
nnet	296	70	1	294	2
sha	296	49	1	292	3
parser	325	102	1	321	3
loops	337	79	3	332	3
zip	377	97	2	372	4
expr	172	103	2	165	6
ls	247	114	6	233	11
csplit	174	107	4	170	3
ptx	191	114	5	186	3
du	305	175	15	291	12

Table 3.2: CFG-based function identification results in comparison to IDA Pro.

Table 3.2 shows the results, which we classify into two categories: recovered functions (RF) and partially recovered functions (PRF). The former indicates that both function boundaries were precisely identified by the tool, while the latter indicates that only one boundary was identified. Fig. 3.6 visualizes this function recovery data for better readability. Across all benchmarks, Spedi precisely recovered 97.6% of the functions on average. Compare this to 37.1% recovered by IDA Pro. Based on this, our CFG-based technique demonstrates a clear advantage. However, improving our results requires facing some challenging corner cases. For example, assume that F_1 and F_2 are two adjacent functions where F_2 is tail-called only from F_1. In such a case, our technique would merge both functions into one function that starts with $F1_1$ and, accordingly, is deemed as partially recovered. The lack of calls to F_2 from other functions makes it obscure to our CFG-based technique. It suggests that complementary techniques might be needed to fill this gap.

Speculative disassembly introduces a peculiar type of errors. Specifically, these errors are located at the entry of functions that are (1) preceded by padding bytes, and (2) called only indirectly. In such a case, the CFG conflict analysis of Section 2.3.3 cannot select a specific basic block among, potentially, several ones in a maximal block. Therefore, we default to selecting the largest possible basic block, a choice that may lead to identify spurious instructions introduced by padding bytes. However, the locality of such errors makes them of less significance.

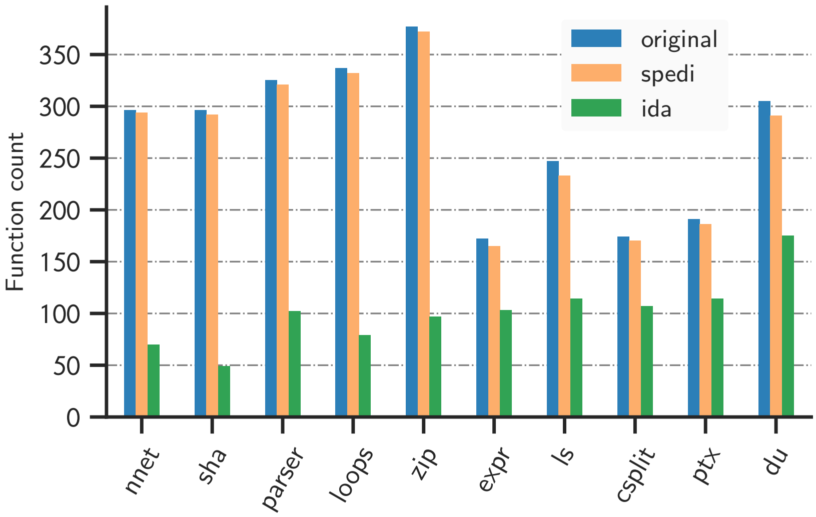


Fig. 3.6: Evaluating CFG-based function identification. We show the results of precisely recovered functions in comparison to the ground truth.

3.5 Discussion and Related Work

Generally, there are two models of binary-level functions considered in the literature. They are the *contiguous model* and the *chunk model*. In the former, a function occupies a single contiguous code region. In contrast, the chunk model is more general by allowing a function to span multiple chunks. Each chunk in turn is a contiguous code region. This model might be motivated by a need to maintain the mapping between a source-level function and its potentially multiple parts in the binary. As a result, a single chunk might be shared between several functions. This model is adopted in tools like Dyninst [92] and `rev.ng` [41].

However, this work adopts the contiguous function model. It is simple; yet, we found it to be consistent with the symbol-based definitions in our large dataset. Moreover, it can be augmented with additional analyses to identify function entries and exits. This provides enough flexibility to handle special situations that might arise in practice, for example, using `ret` to implement indirect calls in Retpoline [120]. Additionally, link-time and post-link optimizations are becoming widespread and more aggressive in changing a function layouts. Consequently, we do not believe that the generality of the chunk model justifies the complexity it requires to use it in practice.

Identifying functions using liker symbols is straightforward. Of course, this assumes that such symbols are trustworthy. Therefore, we looked at CFI records as an alternative source of function definitions. We have shown that they are largely complete in larger binaries, as demonstrated by our experiments in Section 3.4.1 on a large dataset of stripped off-the-shelf binaries. This is an important result since CFI records are often overlooked in the literature.

Additionally, this result shows that the coverage analysis techniques considered in this work can support off-the-shelf binaries without resort to linker symbols.

For the use cases where symbol-based techniques are inadequate, we proposed a CFG-based technique that does not rely on external information. This is, it recovers function definitions by analyzing only the binary. We first presented this technique in our CASES'16 paper [14], where we demonstrated its effectiveness on ARM Thumb binaries. In the following year, two research works independently proposed similar CFG-based techniques and successfully applied them to x86 binaries. They are the works of Andriesse et al. [6] and Qiao et al.[104]. Moreover, the effectiveness and practicality of the CFG-based technique has motivated the developers of Binary Ninja, a commercial binary analysis tool, to adopt it [103].

Before these proposals, the research community has largely relied on pattern matching heuristics, and more recently, machine learning techniques. Functions prologues and epilogues often contain instructions for stack setup and tear down, respectively. Manually developing heuristics for detecting such patterns is not difficult. However, as we noted previously, compiler optimizations are only getting more aggressive. Hence, functions do not necessarily have to start with, or even have, instructions for stack setup. As a result, pattern matching heuristics are generally brittle and costly to maintain.

Pattern matching can be automated using machine learning, a research theme that has been considered by several works. The beginning was with Rosenblum et al. [108], who leveraged logistic regression. Also notable are the works of Bao et al. in ByteWeight [10] and Chul et al. [113] who, for the first time, leveraged neural networks. These proposals attempt to recognize function boundaries by learning byte-level features. With features at such a low level, coupled with the aggressiveness of modern compiler optimizations, it becomes challenging for machine-learning techniques to generalize beyond the training set [6].

Chapter 4

Jump Table Analysis

Overview; Sliced Microexecution; Experiments; Discussion and Related Work.

4.1 Overview

Recovering the targets of indirect control transfer instructions is desirable in several applications, such as reverse engineering and control flow integrity. However, this problem is undecidable in general, which means that we can only aspire to obtain approximate solutions. That is, either an over-approximation or under-approximation of the actual set of targets. Nevertheless, the `switch` statement in C and C++ is commonly implemented as an indirect `jmp` that uses a lookup-table in a manner amenable to precise analysis.

The C/C++ language standards mandate that; if the value of the controlling expression of a `switch` does not match any `case` label, then the control flow should be transferred to the `default` label if one exists. Otherwise, no statement within the `switch` body should be executed. Compilers satisfy this requirement by ensuring that the value of the controlling expression is *bounded*. Also, this bound must be *intra-procedural* since `switch` statements cannot span beyond the scope of the current function. The key insight that we leverage is that combining boundedness and function locality enables the analysis of jump tables to be precise and distinguish them from other constructs of indirect control transfer.

Compilers enjoy a lot of flexibility in implementing `switch` statements. For example, a `switch` statement that has few `case` labels, like the one in Fig. 4.1, will probably be compiled to a series of conditional CTIs. A jump table can be *control-bounded* by checking the value of the controlling expression against a bound condition. Alternatively, should the expected values be dense, e.g., many values below 16, the compiler might prefer a *data-bounded* jump table by using a bitwise `and` with `0xf`. Additionally, compilers are free to divide a `switch` with many `case` labels into multiple jump tables. Our goal in this analysis is to recover precise

```
switch (ilen) {
  case 4:
    *codestr++ = PACKOPARG(EXTENDED_ARG, (oparg >> 24) & 0
       xff);
  /* fall through */
  case 3:
    *codestr++ = PACKOPARG(EXTENDED_ARG, (oparg >> 16) & 0
       xff);
  /* fall through */
  case 2:
    *codestr++ = PACKOPARG(EXTENDED_ARG, (oparg >> 8) & 0xff
       );
  /* fall through */
  case 1:
    *codestr++ = PACKOPARG(opcode, oparg & 0xff);
  break;
  default:
    Py_UNREACHABLE();
}
```

Fig. 4.1: Example of a switch statement taken from function `write_op_arg` in `cpython` v3.7.3. The variable `ilen` represents the controlling expression. This particular example has only four cases, which is usually compiled to a series of conditional CTIs, instead of a single indirect `jmp`.

information about each jump table. This includes its entry type, control-flow targets, and the total number of entries.

The analysis of such jump tables offers a number of benefits. First, it increases CFG precision, which improves the efficiency of probe pruning. Second, it enables jump table instrumentation where we modify jump table entries instead of inserting detours. Third, it allows us to avoid disassembly errors. The latter issue is relevant to architectures such as ARM, where compilers usually inline jump table data in the code section. Fortunately, in x86-64, such data typically reside in a separate read-only section, which enables correct disassembly using a simple linear sweep [5].

4.2 Sliced Microexecution

We propose sliced microexecution, a novel method for jump table analysis that combines classical backward program slicing with microexecution [51]. The latter refers to the ability to emulate any code fragment without manual inputs. Basically, for each indirect `jmp` in a function, bcov attempts to test the sequence of hypotheses depicted in Table 4.1. If they are invalid, then bcov aborts the analysis and considers the `jmp` in question to be a tail call. Otherwise, bcov proceeds with the actual recovery depending on the type of jump table, which can generally either be control-bounded or data-bounded.

	Hypothesis	Action
(1)	Depends on constant base address?	if yes test (2) else abort
(2)	Is constrained by a bound condition?	if yes test (3) else assume (4)
(3)	Bound condition dominates jump table?	if yes do recovery else assume (4)
(4)	Assume jump table is data-bounded	do recovery and try to falsify

Table 4.1: Hypotheses tested, or falsified, to analyze a jump table. Backward slicing answers (1)-(3). Microexecution is used to falsify assumptions and recover the jump table.

We discuss this method based on the example shown in Figure 4.2. First, bcov has to test hypothesis (1) by backward slicing from `0x9f719` until it reaches instruction at `0x9f712`, which has a memory dependency. This dependency stores the base address in `r15`. So is this base address constant? Backward slicing for `r15` shows that it is, in fact, a constant. Note that jump tables should depend on a single variable used as the index. The base address is a constant determined at compile-time.

We move now to test hypothesis (2). To this end, bcov spawns a *condition slicer* upon encountering each conditional `jmp` like the instruction at `0x9f707`. This slicer is used to check whether the variable influencing the bound condition is also the jump table index. This happens to be the case in our example at `0x9f6f0`, where the value in `r12b` influences both the condition at `0x9f707` and the jump table index. Now that a bound condition is found, we need to test it against hypothesis (3).

A jump table might be preceded by multiple conditional comparisons that depend on the index. We apply heuristics in order to quickly discard the ones that cannot represent a bound condition, e.g., comparisons with zero. However, there can still be more than one candidate. Here, we leverage the fact that a bound condition should dominate the jump table. Otherwise, a path in the CFG would exist where the index value remains unbounded. We check for dominance during the backward CFG traversal needed for slicing. Basically, it should not be possible to bypass the bound condition.

Backward slicing produces a slice (code fragment), which captures the essential instructions affecting the jump table. This slice represents a univariate block-box function with the index as its input variable. Modifying the index should trigger behavioral changes, especially in the observed jump address at the output. Assuming that this slice represents a jump table, we reason about its behavior using microexecution. Also, we try to validate our assumption by widely varying the index.

```
9f6a1: lea    r15,[rip+0xe69e4] ; set base
.
9f6f0: movzx  eax,r12b    ; index is r12b
9f6f4: cmp    r12b,0x5b   ; bound comparison
9f6f8: mov    QWORD PTR [rsp+0x8],rax
9f6fd: mov    rax,QWORD PTR [rbx]
9f700: mov    r13,QWORD PTR [rax+0x10]
9f704: mov    ecx,r13d
9f707: ja     9f880   ; jump to default case
9f70d: mov    rax,QWORD PTR [rsp+0x8]
9f712: movsxd rax,DWORD PTR [r15+rax*4]
9f716: add    rax,r15
9f719: jmp    rax     ; jump to matching case
```

Fig. 4.2: Example of a jump table taken from perl v5.28 compiled with gcc v7.3. Highlighted instructions are not part of the backward slice. The jump table base is set relatively far at `0x9f6a1`.

Before starting microexecution, bcov needs to first load the code and data segments of the binary. We implemented a custom ELF loader to support this functionality. Then, bcov initializes a valid memory environment for the given code slice. For example, it allocates memory for the pointer `[rsp+0x8]` and assigns a valid address to `rsp`. It is now possible to start "fuzzing" the index. However, the expected behavior of the slice depends on the type of jump table.

In control-bounded jump tables, a change in behavior must be observed between the intervals $[0, b)$ and $(b, +\infty)$, where b is the bound constant. This constant is located in the first instruction that sets the flags before the bound condition. In our example, this is the instruction at `0x9f6f4` where b equals `0x5b` (or 91). bcov tests 24 index values in total, 8 of which are sampled from $[0, b]$ including 0, $b - 1$, and b. The remaining 16 values increase exponentially, in power of 2, starting from $b + 1$. We found this scheme to give us high confidence in the analysis results.

The jump table is expected to target an instruction inside the current function for most inputs in $[0, b)$. In contrast, the jump table should not be reachable for all inputs in $(b, +\infty)$. That is, the bound condition should redirect control flow to the default case. Table 4.2 shows the expected behavior of our running example for four different index values. Should the behavior of the code slice not match what we expect from a control-bounded jump table, then we abort and assume it to be data bounded. Note that we are not strict about the behavior for input b since the bound condition might check for equality.

Assuming that a given indirect `jmp` represents a data-bounded jump table, we need effective techniques to (1) stop backward slicing, (2) validate our assumption, and (3) explore the bound limits. Note that compilers might use more than one bitwise instruction to bound

Address	Check	`r12b=0`	`r12b=45`	`r12b=91`	`r12b=128`
`0x9f712`	Accessed a read-only memory segment?	yes	yes	maybe	n/a
`0x9f719`	Reached `jmp` instruction?	yes	yes	maybe	no
`0x9f719`	Target address within function?	yes	yes	yes	yes

Table 4.2: Expected behavior of jump table example for 4 different index values (`r12b`). Behavioral checks are apply to specific instruction addresses. The bound constant is 91. Jump tables are typically stored in a read-only segment to maintain run-time control-flow integrity. For all index values greater than 91, we expect the bound condition to redirect control flow to the default case, which is also expected to be within the function.

the index. Moreover, developers might prefer computed gotos over `switch` statements. [1] In this case, they need to assume responsibility for checking index bounds. To cope with this implementation diversity, bcov continues backward slicing as long as the current slice depends on only one variable. For example, assume that `rax` holds the index. Then, `rax` is used as a base register to read from memory. This means that the current slice would depend on `rax` in addition to the accessed memory variable. We would stop backward slicing before this increase in dependencies.

Having identified the relevant program slice, we validate our assumption and explore the bound limits of the jump table. To this end, bcov executes the slice 24 times, each time increasing the index exponentially while setting the least significant bits to one. This allows us to explore the bound limits in the common case of a bitwise `and` with a bitmask like `0xf`. We also try other bit patterns to better penetrate possible combinations of bitwise instructions. Our key insight is that we should not have full control over the jump target. In other words, supplying an arbitrary index value should reflect in a *constrained* jump table target. Additionally, similar to the case of control-bounded jump tables, the jump targets must be located inside the current function. Should the slice withstand our diverse tests, then we can be highly confident that it represents a jump table we could falsify this assumption.

Our experiments show that sliced microexecution is precise and robust against various compiler optimizations. It even allowed bcov to recover the jump tables of the core loop of the Python interpreter, which are located in function `_PyEval_EvalFrameDefault`. Note that these jump tables are compiled from complex computed gotos. However, recall that in computed gotos, it is the developer's responsibility to ensure the boundedness of the index. Fig. 4.3 shows a computed goto example where the index is not intra-procedurally bounded. Actually, the bounds are enforced in the callers of function `mlp_filter_channel_x86`. In

---

[1] Computed gotos is a gcc extension to C which is also supported in clang. See: `https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html`

```
static void mlp_filter_channel_x86(
                int32_t *state, const int32_t *coeff,
                int firorder, int iirorder,
                unsigned int filter_shift, int32_t mask,
                int blocksize, int32_t *sample_buffer)
{
  const void *firjump = firtable[firorder];
  const void *iirjump = iirtable[iirorder];
```

Fig. 4.3: Computed goto usage example taken from `libavcodec`, which is part of `ffmpeg` v4.1.3. Input variables `fireorder` and `iirorder` are not intra-procedurally bounded. Therefore, precise identification of the bound limits requires an inter-procedural analysis, which is complex and might still not be precise enough.

such a case, bcov can identify the base and maybe speculatively recover some jump table targets, but it cannot be sure about the bound limits. A precise *inter-procedural* analysis is needed to address this challenge.

4.3 Experiments

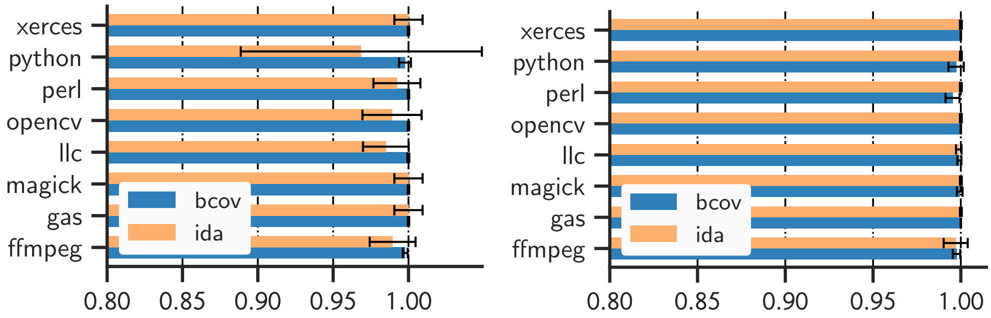
As part of this work, we implemented two different techniques for jump tables analysis. We started with an analysis that combines pattern matching and data flow analysis. We implemented this technique in Spedi, which supports ARM Thumb binaries. The technique demonstrated promising results in comparison to IDA Pro even on optimized binaries [14]. However, pattern matching is generally brittle and does not generalize well. Also, the used benchmarks were relatively small.

These shortcomings were addressed in sliced microexecution, which is implemented in bcov, and the focus of our experiments. The experimental setup is discussed in Section 7.2.2. Our experiments are guided by the following research questions:

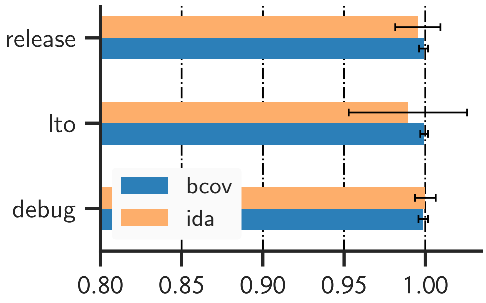
- Have we improved upon the state of the art in jump table analysis?
- What insights can we gain from our large dataset of jump tables?

4.3.1 Comparison with the state of the art

Evaluating sliced microexecution requires comparing bcov with representative binary analysis tools. We experimented with BAP [26] and angr [114], which are the leading academic tools. However, BAP does not have built-in support for jump table analysis. On the other hand, angr (tested version 8.18.10) does support such an analysis. However, it crashed on opencv and llc binaries. For the remaining binaries, angr reported significantly fewer jump tables



(a) clang binaries **(b)** gcc binaries



(c) build type

Fig. 4.4: Normalized jump table analysis results in comparison to IDA Pro. **(a)** IDA Pro shows significant variance on clang binaries. **(b)** both tools are comparable on gcc binaries. **(c)** varying the build type did not affect bcov.

compared to IDA Pro. Sometimes less than half the number of existing jump tables. Therefore, we compare bcov only with IDA Pro (version 7.2). This should not affect our results since IDA Pro is the leading industry disassembler.

Next, we have to establish the ground truth of jump table addresses. Specifically, the addresses of their indirect `jmp` instructions. This is challenging as compilers do not directly emit such information. Therefore, we conducted a differential comparison. We observed that bcov and IDA Pro agree on the majority of jump tables, including their targets, so we manually examined the remaining cases where they disagree. Both tools did not report false-positives, i.e., they only missed jump tables. This is expected in bcov as repeated microexecution inspires high confidence in its results. Therefore, our ground truth is the union of jump table addresses recovered by both tools. This dataset contains 46,425 jump tables found in 95 binaries.

Figure 4.4 depicts the recovery percentages relative to this ground truth. We control for different factors affecting compilation. We observe that IDA Pro delivers lower accuracy on clang binaries compared to gcc binaries. Also, the accuracy of IDA Pro was affected by compiler optimizations as we changed the build type. On the other hand, bcov demonstrates high robustness across the board.

4.3.2 Distribution of jump tables

The number of jump tables in our dataset is relatively large. To be exact, there are 46,425 jump tables. In this section, we query this dataset to gain more insights about how jump tables are generated by compilers. Fig. 4.5 shows various distributions of our dataset divided by four key factors, namely, build type, used compiler, benchmark subject, and entry type.

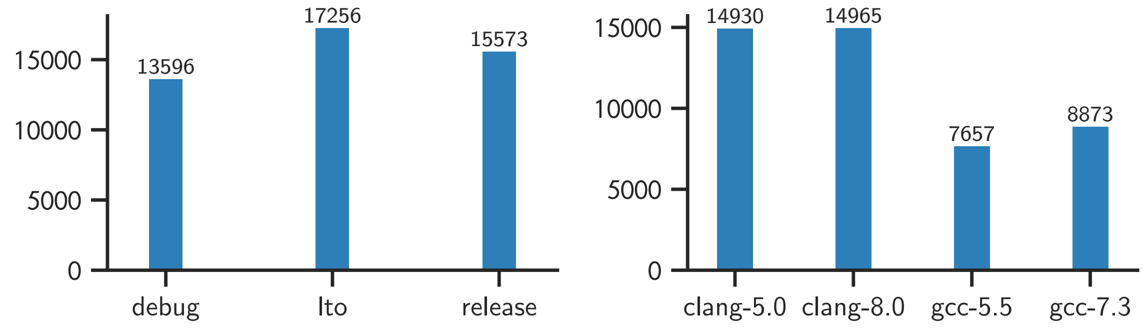
We observe that higher compiler optimizations lead to emitting more jump tables. Note that the number of `switch` statements in the source code remains unchanged. This suggests that splitting large `switch` statements can be a useful optimization. Additionally, it is remarkable that `clang` emits significantly more jump tables compared to `gcc`. Also, we find that the number of jump tables does not correlate well with binary size. For example, the `xerces` library is a bit larger than `magick` library. Despite this, it contains less than third of the jump tables contained in the latter.

Now we discuss the types of jump table entries, which is a deciding factor in the ability to patch jump tables. Generally, an entry can either store an absolute address or an offset. The latter would be added to the current `pc` to determine the target address. Smaller offsets cannot be used in patching, as the code section of the trampolines cannot be reached from the current `pc`. For example, an unsigned 8-bit offset enables reaching only 255 bytes further apart. Fortunately, 8-bit and 16-bit offset represent only 8.3% of the jump tables. In fact, such small offsets are used exclusively in `opencv`, which makes it an outlier case. Other binaries rely either on `soff32` or `abs64` entries. It worth noting that `abs64` entries constitute the majority of entry types. This can be attributed to the fact that our executable subjects are position-dependent. We expect `soff32` entries to be the main choice for compilers, as the current trend towards more position-independent executables continues in the future.

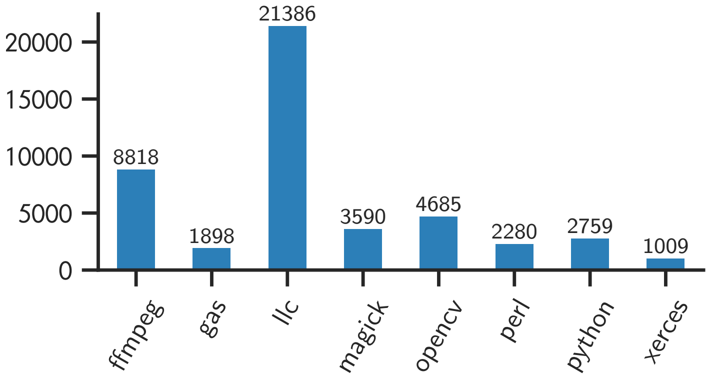
4.4 Discussion and Related Work

In this chapter, we discussed sliced microexecution, a novel technique for jump table analysis that combines classical program slicing with microexecution. Our experiments show that this technique is fast, precise, and robust. Our discussion was purposefully focused on the general case. In practice, however, we encountered several corner cases that are worth highlighting.

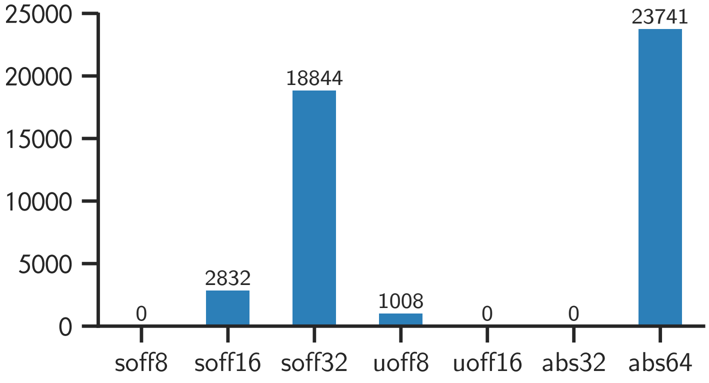
Generally, the targets of jump tables are intra-procedural. This distinguishes them from the (rather rare) construct of tail-call tables. The targets of the latter are inter-procedural, i.e., they target other functions. However, we observed that jump tables can, in certain situations, contain inter-procedural entries. The observed instances were limited to `llc`. For example, the function `PromoteFloatResult` consists of a single `switch` statement which, in one of its labels, calls the leaf function `PromoteFloatRes_BITCAST`. The compiler decided to



(a) build type **(b)** used compiler



(c) benchmark



(d) entry type

Fig. 4.5: Distribution of jump tables across our dataset. The type of a jump table entry depends on its signedness, size, and value type. For example, an entry of type `soff32` is a signed 32-bit offset, while `abs64` is an unsigned 64-bit absolute address.

optimize out the tail call to a leaf function. Instead, it inserted a jump table entry that directly targets the function `PromoteFloatRes_BITCAST`.

Also in `llc`, the function `SoftenFloatResult` represents another interesting example. Again it consists of a single switch statement where the `default` label leads to executing the function `llvm_unreachable`. In debug builds, `llvm_unreachable` simply logs an error message and aborts. In other builds, however, it is replaced with `__builtin_unreachable`. This signals to the compiler that the code in question cannot be reachable. In other words, its behavior is undefined. The compiler leverages this hint by inserting an entry in the jump table, targeting the padding bytes of the current function. In this way, should the program ever reach this code, it will simply continue executing any function that happens to be the successor of the current function.

The analysis of jump tables has been considered in several works. A combination of pattern matching and data-flow analysis has been proposed by Meng et al. [92]. Our work in [14] explores similar techniques. Cifuentes et al. [31] are probably the first to use backward slicing. The generated slice is converted to a canonical IR expression, and then checked against known jump table forms. A custom value-set analysis using SMT solving has been implemented in JTR [34]. It is applied after lifting instructions to LLVM IR. Kinder et al. [74] approached the problem from an abstract interpretation perspective. They proposed several abstract domains, which build on a custom IR. However, their precision and performance evaluations are relatively limited.

In contrast to other works, our sliced microexecution technique semantically reasons about jump tables without manual pattern matching. Also, we do not lift instructions to an IR. Lifting binary instructions to an IR is known to be challenging and error-prone [73]. Additionally, it can drastically affect the performance of the analysis [131]. Instead, sliced microexecution leverages the *executable* instruction semantics already existing in off-the-shelf multi-architectural emulators, such as QEMU. As a result, porting our technique to other ISAs, we believe, will not require a major effort.

Chapter 5

Probe Optimizations

Overview; Probe Pruning; Probe Pruning Alternatives; Optimized Probe Selection; Experiments; Discussion and Related Work.

5.1 Overview

Tracking code coverage requires instrumenting the program under test with probes, such that visiting a probe implies that a certain number of code artifacts, e.g., basic blocks, have been covered. Probe instrumentation needs to be transparent and efficient. Sometimes both goals are interdependent. For example, inefficient instrumentation might break transparency by introducing regressions on performance tests.

This chapter focuses on improving the efficiency of probe instrumentation. Specifically, we discuss techniques for pruning redundant probes. Here, we build on the work of Agrawal [1], where we leverage dominance relationships between basic blocks to build superblocks, which we connect in a superblock dominator graph. First, we provide the necessary background about this technique and discuss how it can be used for probe optimization. Then, we compare it to other well-known alternatives. Lastly, we show how to select the best basic block to instrument among several ones that share the same superblock.

5.2 Probe Pruning

Tracking basic block coverage by instrumenting every basic block can be highly inefficient. Additionally, such instrumentation is quite challenging to do in ISAs that feature variable-size instructions. For example, a basic block can occupy only one byte in x86-64 ISA, which leaves no room to insert a detour. Probe pruning reduces the need to instrument smaller basic blocks.

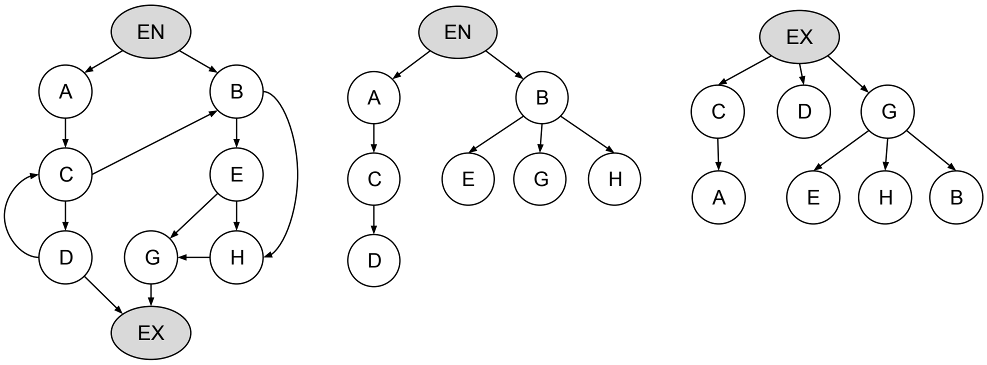
Moreover, it makes testing more efficient by reducing both the instrumentation overhead and the size of coverage data. In this section, we provide the necessary background on the probe pruning technique implemented in bcov based on Agrawal [1]. The original work considered source-level probe pruning for C programs. We focus instead on binary-level pruning for both C and C++ programs.

Given a function F with a set of basic blocks B connected in a CFG. The straightforward way to obtain complete coverage data is to probe every basic block $bb \in B$. However, it is possible to significantly reduce the number of required probes by computing *dominance* relationships between basic blocks in a CFG. We say that bb_i predominates bb_j, $bb_i \xrightarrow{pre} bb_j$, iff every path from function entry (EN) to bb_j goes through bb_i. Similarly, bb_i postdominates bb_j, $bb_i \xrightarrow{post} bb_j$, iff every path from bb_j to function exit (EX) goes through bb_i. We say that bb_i dominates bb_j iff $bb_i \xrightarrow{pre} bb_j \vee bb_i \xrightarrow{post} bb_j$. The predominator and postdominator relationships are represented by the trees T_{pre} and T_{post} respectively. The dominator graph (DG) is a graph that captures all dominance relationships. It is obtained by merging both trees $DG = T_{pre} \cup T_{post}$, i.e, by merging the edges of both trees. This results in a directed graph that is usually cyclic.

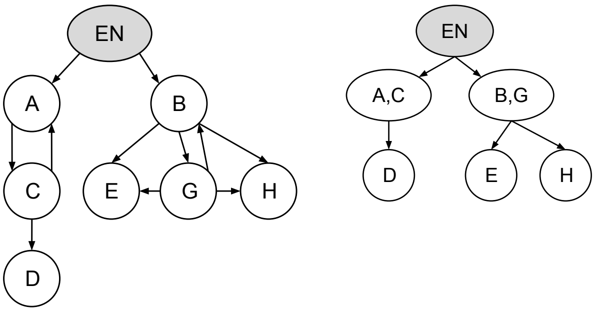
Given a dominator graph and the fact that a particular bb is covered, this implies that all dominators (predecessors) of bb in DG are also covered. This allows us to avoid probing basic blocks that do not increase our coverage information. However, we are interested in moving a step further by leveraging strongly-connected components (SCCs) in the DG. Each SCC represents a *superblock*, a set of basic blocks with equivalent coverage information. The superblock dominator graph (SDG) is constructed by merging SCCs in the DG. That is, each SB node in SDG represents a SCC in the DG. An edge is inserted between SB_i and SB_j iff $\exists\, bb \in SB_i, \exists\, bb' \in SB_j$ where bb dominates bb'.

Constructing the SDG offers several benefits. First, it is a convenient tool to measure the coverage information gained from probing any particular basic block. Second, it enables compressing coverage data by tracking superblocks instead of individual basic blocks. For example, we found that the number of superblocks represents 46% of all basic blocks in our precise any-node policy. This percentage goes does down to about 30% in the heuristic leaf-node policy. Third, it provides flexibility in choosing the best basic block to probe in a superblock. We show later in Section 5.4 how this flexibility can be leveraged to reduce instrumentation overhead.

Figure 5.1 depicts an SDG that is obtained from a CFG. Note that EN and EX are *virtual* nodes commonly used to simplify dominance analysis. The SDG can easily tell us that probing D provides strictly more information than probing C or A because covering D implies that A and C are also covered. Moreover, given that A and C share the same SB, probing A might



(a) CFG **(b)** Predominator tree **(c)** Postdominator tree



(d) Dominator graph **(e)** SDG graph

Fig. 5.1: An example CFG and its corresponding SDG. First, predomominator and postdominator trees are constructed and merged in a dominator graph (DG). Nodes in SDG represent strongly-connected components in DG. In the *leaf-node* policy, we probe only the leaf nodes in SDG, namely, D, E, and H. In the *any-node* policy, we additionally probe either A or C.

be a better choice since C is a loop head. That is, instrumenting it can unnecessarily trigger multiple coverage updates.

We implemented two instrumentation policies in bcov. They are the *leaf-node* and *any-node* policies. In the leaf-node policy, we instrument only the leafs of the SDG. Covering *all* such leaf nodes implies that all nodes in SDG are also covered, i.e., achieving 100% coverage. However, writing, maintaining, continuously running tests is costly. Moreover, the correlation between high coverage ratio and defect freedom is questionable [67]. Therefore, developers in practice might settle for 80% as a target coverage ratio [69]. Nevertheless, leaf nodes can still provide high coverage information. This makes the leaf-node policy useful to approximate the coverage of a test suite at a relatively low overhead.

Generally, we are also interested in inferring the exact set of covered basic blocks given *any* test input. This is usually not possible in the leaf-node policy. For example, given an input that visits the path $A \longrightarrow C \longrightarrow B \longrightarrow H \longrightarrow G$, the leaf-node policy can report that the covered set is $\{B, H, G\}$. However, this policy can make no statement about the coverage of A and C since they do not dominate the visited probe in H. We address this problem in the

Algorithm 7: Identification of critical superblocks in the any-node policy

```
Input   : Superblock $sb$
Input   : Superblock dominator graph $SBG$
Input   : Control flow graph $CFG$
Output  : Is critical superblock? (boolean)
if child_count($SBG$, $sb$) < 2 then
    return true
end
Mark all basic blocks in CFG as unvisited
Mark a representative basic block $bb$ in $sb$ as visited
for $sb_i \in$ get_children($SBG$, $sb$) do
    Mark a representative basic block $bb$ in $sb_i$ as visited
end
// CFG traversal
visit_predecessors($CFG$, $bb$)
visit_successors($CFG$, $bb$)
// Now check results
$en$ = get_entry($CFG$)
$ex$ = get_exit($CFG$)
if visited($en$) and visited($ex$) then
    return true
else
    return false
end
```

any-node policy. The set of superblocks instrumented in this policy is a super set of those of the leaf-node policy. More precisely, $S_{any} = S_{leaf} \bigcup S_c$, where S_c represents the set of *critical* superblocks. Each $sb \in S_c$ can be visited by at least one path p in the CFG, such that the path p bypasses all the children of sb in the SDG.

It is possible to determine S_c using an $\mathcal{O}(|V| + |E|)$ algorithm where V and E are the nodes and edges in the CFG, respectively. The steps are described in Algorithm 7. We first check the number of SDG children of a given sb. If it equals zero, then this is a leaf node and, therefore, will be probed in both instrumentation policies. However, it can be shown that should an sb have a single child, then it must be critical. Because otherwise, it would have been merged in the same SCC together with its child.

We come to the general case of an sb that has two or more children in the SDG. For each such superblock, we mark its children in the SDG as visited in the CFG. Then, we pick an arbitrary basic block $bb \in sb$ and traverse the CFG backwards and forwards starting from bb. If EN and EX are both *both* reachable, then we add sb to S_c, otherwise we consider sb to be non-critical since it can not bypass its children in the SDG. For further details, we advise interested readers to refer to [1]. In Figure 5.1, the superblock $\{B, G\}$ is non-critical. However, the superblock $\{A, C\}$ is critical. Subsequently, it will be probed in the any-node policy.

5.3 Probe Pruning Alternatives

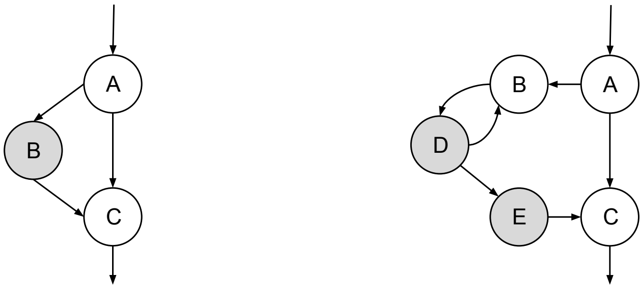
There are several probe pruning techniques proposed in the literature. We ultimately decided to choose Agrawal's technique [1]. First, it is optimal in terms of probe count. More importantly, its distinct notion of superblocks provides crucial flexibility in choosing probe locations. This, in turn, allows us to further reduce instrumentation overhead.

In this section, we compare Agrawal's pruning technique to other well-known techniques. This discussion offers a better understanding of alternative design options. The first alternative in our discussion is the proposal of Tikir and Hollingsworth [119], which we will simply refer to as TH technique. Also, the popularity of SanitizerCoverage [109] (henceforth sancov) make us include it in this comparison since it implements a modified version of the TH technique.

We first need to introduce a couple of definitions. We say that a basic block is a *full predominator* if it predominates all of its successors in the CFG. Similarly, a *full postdominator* postdominates all of its predecessors in the CFG. The TH technique prunes full predominators. This is extended in sancov by also pruning full postdominators that have more than one predecessors in the CFG. Therefore, both techniques are simpler to implement in comparison to Agrawal's technique. They require the construction of dominator trees only. However, the simplicity of these alternatives brings along considerable drawbacks, including a potential loss in coverage information and suboptimal instrumentation.

We elaborate on these issues based on Figure 5.2. First, it is possible to visit the *critical* edge A⟶C without signaling coverage feedback, as the probe in B is not visited. However, sancov solves this issue using its *edge* coverage mode, which is the default, by splitting critical edges with dummy basic blocks *before* pruning probes. In comparison, the any-node policy in bcov ensures that either A or C is instrumented, which has the same effect. This allows us to avoid introducing dummy basic blocks, which is cumbersome to do at the binary level. Second, the techniques of TH and sancov may introduce redundant instrumentation and thereby increase the overhead. In Figure 5.2b, they would instrument D and E, although both nodes provide identical coverage information. After all, D and E belong to the same superblock. bcov does not suffer from such redundancy since it strictly picks only one probe per superblock.

We implemented the TH and sancov techniques and tested them along with bcov on the same dataset. This allows us to fairly quantify the difference. The dataset consists of our



(a) Lost coverage **(b)** Redundancy

Fig. 5.2: Basic block pruning techniques in TH and sancov would instrument highlighted nodes. **(a)** Coverage for critical edge A ⟶ C is lost. **(b)** Instrumentation of D is redundant.

benchmarks, which are described in Section 7.2.2, compiled with gcc-7 in release build. It contains more than 2.6×10^6 basic blocks. TH and sancov report that 49% and 43% of the basic blocks need to be probed, respectively. Compare this to 45% reported by the any-node policy of bcov. Additionally, we noticed that 7% of the probes in TH are either completely redundant (multiple probes per superblock), or do not provide optimal coverage (inner nodes in the SB-DG). Compare this to around 2.5% in the sancov technique. This suggests that bcov instruments a comparable percentage of basic blocks while providing superior edge-level precision. Equally important is the fact that the notion of superblocks, which is unique to Agrawal's technique, enables us to optimize the selection of probes, as discussed in the next section.

5.4 Optimized Probe Selection

Generally, probing a basic block (BB) requires inserting a detour targeting its designated trampoline. A detour occupies 5 bytes and can either be a direct `jmp` or `call`. Consequently, one or more original instructions must be relocated to the trampoline. This *relocation* overhead varies due to the instruction-size variability in the x86-64 ISA. Note that a pc-relative `mov`, which occupies 7 bytes, represents an unavoidable overhead in each trampoline, which is required to update coverage data. Hence, our goal is to reduce the relocation overhead.

To this end, we iterate over all BBs in a given superblock and select the one expected to incur the lowest relocation overhead. First, we have to establish whether a detour can be accommodated in the first place. A BB that satisfies $s + p < 5$ is considered a guest, where s and p are the byte size and padding size respectively. A superblock that contains only guest BBs is handled via detour hosting, which is discussed later in Section 6.2.

Now we examine (1) the type and size of the last instruction of each basic block (BB) and (2) whether the BB is targeted by a jump table. These parameters are translated into the BB

Type	RP	Relocation overhead
return	maybe	Can be only 1 byte depending on padding size
long-jump	no	Size of `jmp` instruction which is ≥ 5 bytes
long-call	no	Size of `call` instruction which is ≥ 5 bytes
jump-tab	no	Size of `jmp` instruction to original code (5 bytes)
short-call	yes	Similar to long-call but with RP overhead added
short-jump	yes	Similar to long-jump but with RP overhead added
internal	maybe	Size of relocated instruction(s) inside the BB
long-cond	no	Rewriting incurs a fixed overhead of 11 bytes
short-cond	yes	Similar to long-cond but with RP overhead added

Table 5.1: BB type classification used in probe selection. Types are shown in ascending order based on expected relocation overhead. The terms *long* and *short* are relative to detour size (5 bytes). Short types require relocating preceding (RP) instruction(s).

types depicted in Table 5.1. We show examples of patching these BB types in Table 5.2. We organize these BB types in a total order. This means that, for example, we strictly prefer a long-call over a long-cond should both exist in the same superblock. This type order is primarily derived from empirical observation. However, we did not necessarily experiment with all possible combinations. Preferring long-call over short-call should be intuitive. The latter incurs an additional overhead for relocating at least one instruction preceding the `call`.

We observed that return basic blocks are usually padded (55% on average). Their padding size is often more than 3 bytes, which translates to a relocation overhead of only one byte - the size of a `ret` instruction. Also, favoring long-jump over long-call provided around 3% improvement in both relocation and performance overheads. On the other hand, short-call had only a slight advantage over short-jmp. This might be due to the fixed two-byte size of the latter, which leads to relocating more instructions. However, our experiments were not always conclusive, e.g., choosing between jump-tab and short-call.

Relocating an instruction depends on its relation to the PC (rip in x86-64). Position-independent instructions can be simply copied to the trampoline.

Type	Original code	Patched code	Trampoline
return	411957: pop r14 411959: pop r15 41195b: ret 41195c: nop DWORD PTR [eax]	411957: pop r14 411959: pop r15 41195b: jmp 2016b59	2016b59: mov BYTE PTR [rip+0x3a1a63],1 2016b60: pop rbp 2016b61: ret
long-jump	408449: jmp QWORD PTR [rcx*8+0x136a0f0]	408449: jmp 2012d66 40844e: nop 40844f: nop	2012d66: mov BYTE PTR [rip+0x3a547c],1 2012d6d: jmp QWORD PTR [rcx*8+0x136a0f0]
long-call	13165fd: call QWORD PTR [rip+0xcfa805]	13165fd: nop 13165fe: call 23a1257	23a1257: mov BYTE PTR [rip+0x4e5e2],1 23a125e: jmp QWORD PTR [rip-0x39045c]
jump-tab	408579: cmp al,0x73 40857b: jne 4085f0	408579: cmp al,0x73 40857b: jne 4085f0	2012c33: mov BYTE PTR [rip+0x3a559b],1 2012c3a: jmp 0x408579
short-call	411f84: mov rsi,r15 411f87: mov rdx, rbp 411f8a: call rax	411f84: mov rsi,r15 411f87: call 2016f08	2016f08: mov BYTE PTR [rip+0x3a16f0],1 2016f0f: mov rdx, rbp 2016f12: jmp rax
short-jump	42550e: movsxd rdi,edi 425511: add r13, rdi 425514: jmp 0x425569	42550e: movsxd rdi,edi 425511: jmp 201ee2a	201ee2a: mov BYTE PTR [rip+0x399fed],1 201ee31: add r13, rdi 201ee34: jmp 0x425569
internal	407a41: mov eax,DWORD PTR[rip+0x1416039] 407a47: inc rbp	407a41: jmp 2012648 407a46: nop 407a47: inc rbp	2012648: mov BYTE PTR [rip+0x3a5b23],1 201264f: mov eax,DWORD PTR[rip-0x7f4bd5] 2012655: jmp 0x407a47
long-cond	4058db: test rsi,rsi 4058de: jne 0x405992 4058e4: mov rdx,r13	4058db: test rsi,rsi 4058de: jmp 20113e7 4058e3: nop	20113e7: mov BYTE PTR [rip+0x39cc6b],1 20113ee: jne 0x405992 20113f4: jmp 0x4058e4
short-cond	405f14: add rax,rcx 405f17: mov QWORD PTR [r13+8],rax 405f1b: jns 0x405f7c 405f1d: add rbx,0x1	405f14: add rax,rcx 405f17: nop 405f18: jmp 2011895	2011895: mov BYTE PTR [rip+0x39c802],1 201189c: mov QWORD PTR [r13+8],rax 20118a0: jns 0x405f7c 20118a6: jmp 0x405f1d

Table 5.2: Examples of patching different basic block types taken from a patched ffmpeg v4.1.3 binary compiled with clang v5.0. Types are shown in ascending order based on expected relocation overhead. The terms long and short are relative to detour size (5 bytes).

However, we had to develop a custom rewriter for position-dependent instructions. It preserves the semantics of the original instruction whether it explicitly or implicitly depends on `rip`. Note that it is important for rewritten `call` instructions to preserve the exact return address. In Table 5.2, for example, shifting the long-call detour to `0x13165fd`, i.e., before padding, might seem innocent. The return address would differ by only one byte, after all. However, this change can introduce test suite regressions in some benchmarks.
Specifically, a small modification of the return address has caused test regressions in `libxerces`, compiled with gcc in a debug build. Surprisingly, other builds of the same library did not exhibit similar regressions.

Jump table instrumentation has the unique property of preserving the original code. It is a data-only mechanism that enables us to probe even a one-byte BB. However, in order to be applicable, a BB has to be targeted by a *patchable* jump table. A jump table is patchable if its entries are either 32-bit offsets or absolute addresses. This ensures that the trampoline can be reached from the original code. In our dataset, we observed that about 92% out of a total of 46,425 jump tables are patchable. Actually, we found that 8-bit and 16-bit offsets are used only in `libopencv_core`.

5.5 Experiments

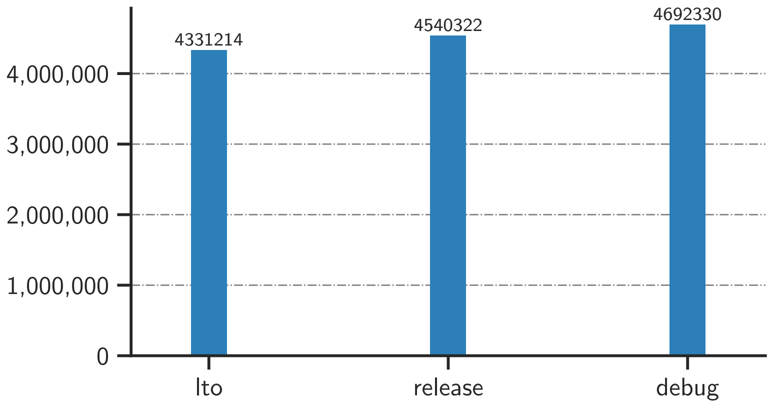
We implemented the probe pruning technique of Agrawal [1] in the tool bcov. Then, we built upon that to leverage its superblocks to optimize probe selection, as discussed in Section 5.4. In this section, we attempt to answer the following research questions:

- How are probes distributed across different builds, compilers, and subject binaries?
- What is the distribution of probe types? What insights can that provide about the effectiveness of the proposed probe selection strategy?

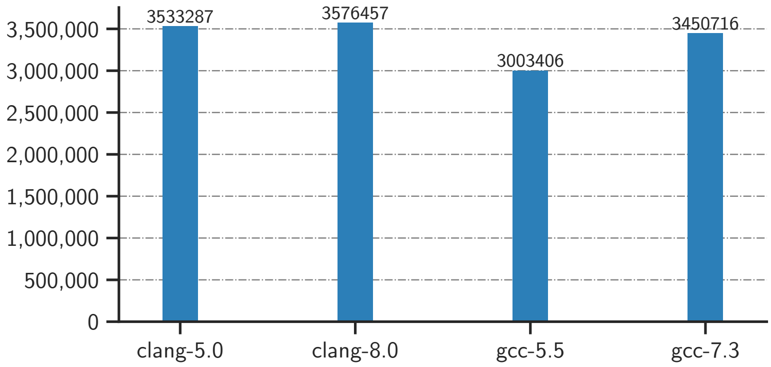
The experimental setup is discussed later in Section 7.2.2. We investigate these research questions based on examining a total of 13,563,866 probes in our dataset, produced by the any-node policy.

5.5.1 Probe count distribution

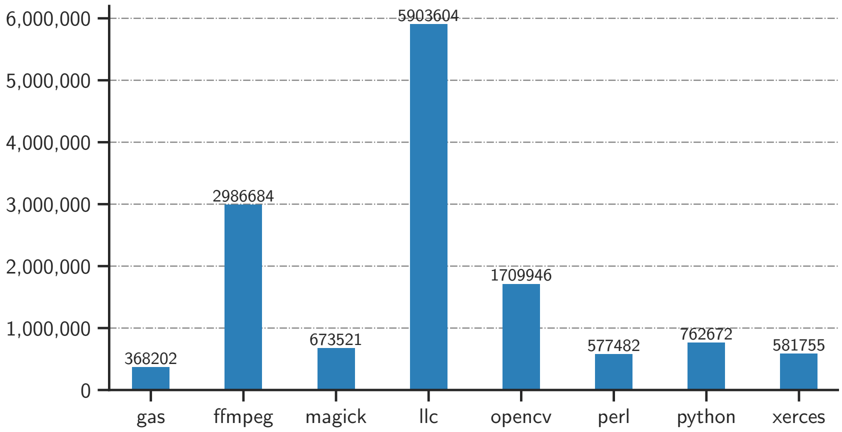
The various distributions of instrumentation probes are depicted in Fig. 5.3. We observe that, for the same instrumentation policy, debug builds generally require more probes. We attribute this to having more code. After all, reducing code size is a primary goal for compiler optimizations. Besides, note that debug builds can emit several unreachable basic blocks in



(a) build type



(b) used compiler



(c) benchmark

Fig. 5.3: Distribution of probe count across our dataset. (a) the higher number of probes in debug builds can be attributed to their larger code size. (b) gcc-5.5 was unable to build `llc` in `lto` build, which caused its number of probes in to be lower for `lto` builds. (c) the average number of probes is not highly correlated with the code size. For example, the smaller `python` binary has unexpectedly higher number of probes compared to `perl`.

each function, i.e., dead code. To maintain high precision, bcov instruments all dead code basic blocks, which further increases the total probe count.

Varying the compiler did not significantly affect the probe count. Note that `gcc-5.5` was unable to compile `llc` in `lto` build. Expectedly, the number of probes in its case is lower than other compilers. Fig. 5.3c depicts the distribution of the average probe count across different binary subjects. The probe count is usually, but not always, proportional to code size. Here, `python` is notable as, despite having a smaller code size, it required significantly more probes compared to `perl`.

5.5.2 Probe type distribution

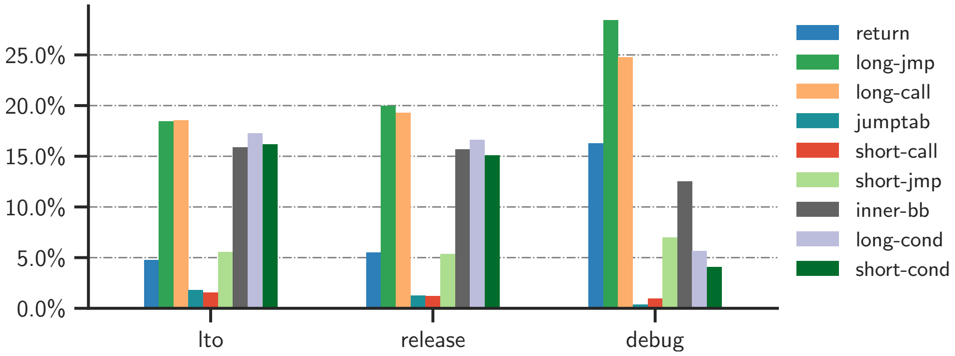
In the previous section, we looked at the distribution of probe *counts* in our dataset. Now, we move to discuss the distribution of various probe *types*. Specifically, we examine the average percentage of each probe type in our subject binaries. Recall that the probe type is the primary criterion relied upon in the probe selection scheme of Section 5.4. Our results are depicted in Fig. 5.4. Similar to the previous section, we divide the results according to the build type, compiler, and subject binary.

We start by examining the effect of modifying the build type. Most notable here is that `debug` builds offer an abundance of `return`, `long-jump`, and `long-call` probes. In turn, bcov seems to be capable of effectively leveraging this observation to reduce overhead. In fact, we observe that, on average, these three probe types alone constitute over 70% of all probes in `debug` builds. However, this percentage drops to about 41% in `lto` builds, as compilers try to reorder code and use smaller instructions to reduce the code size.

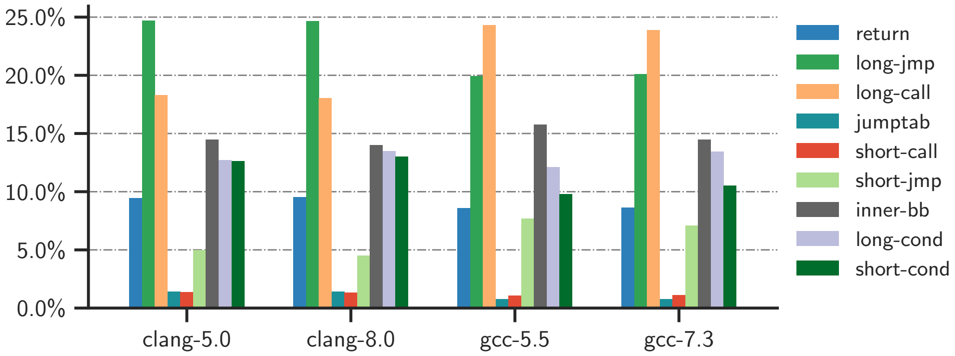
Varying the compiler does affect the distribution of probe types. But using a different version of the same compiler does not seem to have much effect. While the ratio of `return` probes is comparable across compilers, the ratio of `long-jump` is distinctly different. Specifically, about 24% of the probes of `clang` binaries are `long-jump`. Compare this to 20% in the case of `gcc`. Instead, `long-call` is the most common probe type in `gcc` representing about 23% of all probes. Finally, we observe that subject binaries seem to have inherent differences that make the distribution of probe types quite different. For example, `long-call` probes represent about 29% of all probes in `magick`, but only 13% in `ffmpeg`.

5.6 Discussion and Related Work

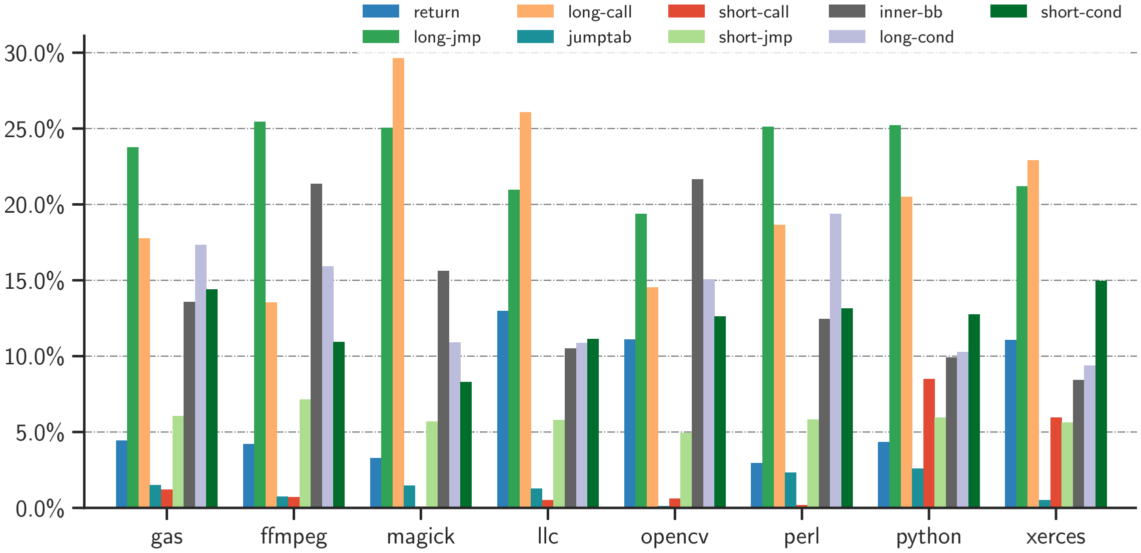
In this chapter, we discussed the optimizations of probes in bcov. First, we reviewed the probe pruning technique of Agrawal's [1], which we adopt in our work. Then, we compared it



(a) build type



(b) used compiler



(c) benchmark

Fig. 5.4: Distribution of probe type ratios across our dataset. (a) debug builds provide more probes of the types `return`, `long-jump`, and `long-call`. (b) `clang` provides more `long-jump` probes compared to `gcc`. Conversely, `gcc` might provide more `long-calls`. (c) subjects vary significantly in probe type ratios.

with alternative techniques implemented in Tikir et al. [119] `sancov` [109]. We showed that both techniques are simpler to implement, but suffer from inefficiencies that are alleviated by Agrawal's technique. Later, we discussed a strategy to optimize the selection of an instrumentation probe in a superblock.

Also, we demonstrated that our probe selection strategy is effective in reducing the relocation overhead. However, it is not necessarily optimal. We observed high variance in the percentage of padded `return`, i.e., a `return` is not always the best choice. Also, a loop-aware strategy might reduce performance overhead by avoiding loop heads. Such optimizations are left for future work.

Marre and Bertolino [88] have attempted to generalize Agrawal's technique by introducing the concept of spanning sets. A spanning set is a minimum subset of entities with the property that any set of test cases covering this subset also covers every entity in the program. However, such generalization is out of the scope of this work. Also, we believe that Agrawal's technique is sufficient for our purpose. On another note, Ball and Larus [9] have investigated the problem of optimal basic block profiling. Profiling counts the number of times each basic block is executed. On the other hand, our work is only concerned with optimal basic block *coverage*, i.e., whether a basic block is executed or not.

Chapter 6

Static Instrumentation

Overview; Detour Hosting; ELF Patching; Experiments; Discussion and Related Work.

6.1 Overview

Instrumenting a binary for coverage tracking can be achieved either by modifying the binary on disk (static) or by instrumenting it in memory at run-time (dynamic). Static instrumentation offers several attractive properties. First, a binary needs to be analyzed only once. Its instrumented version can then be executed several times in a test suite incurring only the overhead caused by instrumentation. In comparison, dynamic binary instrumentation (DBI) tools interleave analysis and instrumentation, which not only increases the overhead but also makes them challenging to implement correctly [25]. Additionally, static instrumentation is more usable compared to DBI. It is often sufficient to replace the original binary with an instrumented version. In comparison, DBI requires modifying the build system to allow it to intercept the binary before it executes. However, it is worth noting that DBI tools can provide superior observability, which includes JIT-compiled and self-modifying code.

This work builds on static instrumentation. Here, we discuss the techniques we implemented to (1) cope with the instruction-size variability of x86-64 ISA, and (2) extend an ELF binary with two additional ELF segments, which are used to host trampoline code and coverage data.

6.2 Detour Hosting

The instruction-size variability in x86-64 suggests that some BBs are simply too short to insert a detour without overwriting the following BB. In our dataset, we found that about 6.5% of all

```
ad67f3: jmp   ad6803
ad67f5: nop   [multi-byte]
ad6800: call  QWORD PTR [rax+0x58]
```

(a) original code

```
ad67f3: jmp   1d31afa  ; jump to relocated host
ad67f8: call  1d31b39  ; hosted detour
ad67fd: nop   DOWRD PTR [rax]
ad6800: jmp   ad67f8   ; jump to hosted detour
```

(b) patched code

```
1d31b39: mov  BYTE PTR [rip+0x4f8d01],1
1d31b40: sub  QWORD PTR [rsp], -6
1d31b45: jmp  QWORD PTR [rax + 0x58]
```

(c) trampoline

Fig. 6.1: Detour hosting example taken from llc v8.0 compiled with clang v5.0. **(a)** host is a short `jmp` at `0xad67f3` followed by 11 padding bytes. **(b)** inserting 2 detours leaves 3 padding bytes. **(c)** return address adjusted at `0x1d31b40`. Original `call` at `0xad6800` is rewritten to a matching `jmp` at `0x1d31b45`

BBs are short. That is, their byte size is less than 5 bytes. Left without a probe, we risk losing coverage information of a particular short BB and, potentially, all of its dominators.

One possible solution is to relocate the entire function to a larger memory area. In this way, we can inline coverage update code without affecting adjacent functions. However, function relocation is costly in terms of code size and the engineering effort required to fix relocated code references. For example, throwing an exception from a relocated function without fixing its corresponding CFI record can lead to abrupt process termination.

The method adopted in bcov is *detour hosting*, where the detour of a short BB (guest) can be hosted in a larger BB (host). This method does not require function relocation and preserves the stability of code references at the basic block level. To be applicable, the size of a guest BB needs to be at least 2 bytes, which is sufficient to insert a short detour targeting a host BB that is *reachable*, i.e., within about ±128 bytes. The host BB must be large enough to accommodate two regular detours, i.e., at least 10 bytes. The first detour targets its trampoline while the other detours would target the trampolines of their respective guests. Note that we can safely overwrite padding bytes of both the guest and host. Also, the host does not need to be entirely relocated. Relocating a subset of its instructions might be sufficient.

Figure 6.1 depicts a detour hosting example. It involves a guest consisting of an indirect call (3 bytes). The tricky part about a call instruction is that we must preserve its return address. To this end, we use a `sub` instruction (5 bytes) in the trampoline to adjust the return address from `0xad67fd` to its original value of `0xad6803`. This adjustment also clobbers the

CPU flags. In this context, however, the flags can be safely clobbered since they are not preserved across function calls in the x86-64 ABI. Note that this is the only case where we modify the CPU state.

Now, we have the following allocation problem: given a guest g and a set of suitable hosts $H = \{h_1, h_2, .., h_n\}$, find the host h_i whose selection incurs minimal overhead. Moreover, we are also interested in the more general formulation: given a set of guests $G = \{g_1, g_2, .., g_k\}$ and a set of hosts $H = \{h_1, h_2, .., h_n\}$, where each host is suitable for at least one guest, find a function mapping $M : G \longrightarrow H$ such that the overhead is minimal. We approach this problem using a greedy strategy where we prefer, in this order, (1) packing more guests in a single host, (2) a host already selected to be probed over an intact host, (3) a host that is closer to the guest. Basically, for each guest, we iterate over all reachable BBs. A BB can offer a hosting offset, if possible. A higher offset means that more guests are packed in this host. The initial offset is 5 bytes from the start of host. Should offered offsets be equal, we look into (2) in order to avoid, as much as possible, relocating otherwise intact BBs. Finally, should both (1) and (2) be equal, then we look into (3) in order to have better code cache locality.

It is not possible to patch one-byte guests. Also, neighboring basic blocks might be too small to host a detour for others. Moreover, our current prototype does not support hosting guests in some corner cases. For example, a `call` instruction that depends on `rsp`, like `call QWORD PTR [rsp+0x38]`, can not be safely rewritten to a matching pair of `call` detour and `jmp` in the trampoline. In such a case, the first `call` to the trampoline will modify `rsp` value. In turn, the corresponding `jmp` in the trampoline will read an incorrect `rsp` value. We decided to leave this additional work for the future. Fortunately, these corner cases did not significantly affect the correctness of the reported coverage.

In the case that a guest cannot be hosted, we can still reduce the potential loss in coverage information. To this end, we try to probe all of the immediate predecessors of the current SB, containing the guest, in the SB-DG. However, this does not necessarily imply adding more probes. For example, additional probes are unnecessary in the leaf-node policy, should the current SB have siblings in the SB-DG. Also, the predecessors might be probed already in the any-node policy.

6.3 ELF Patching

The Executable and Linkable Format (ELF) is a popular file format for executables, shared libraries, and object code. It is part of the System V application binary interface [58]. The specification was originally developed by UNIX System Laboratories (USL), and subsequently adopted by most Unix and Unix-like systems. The ELF format is designed to be flexible,

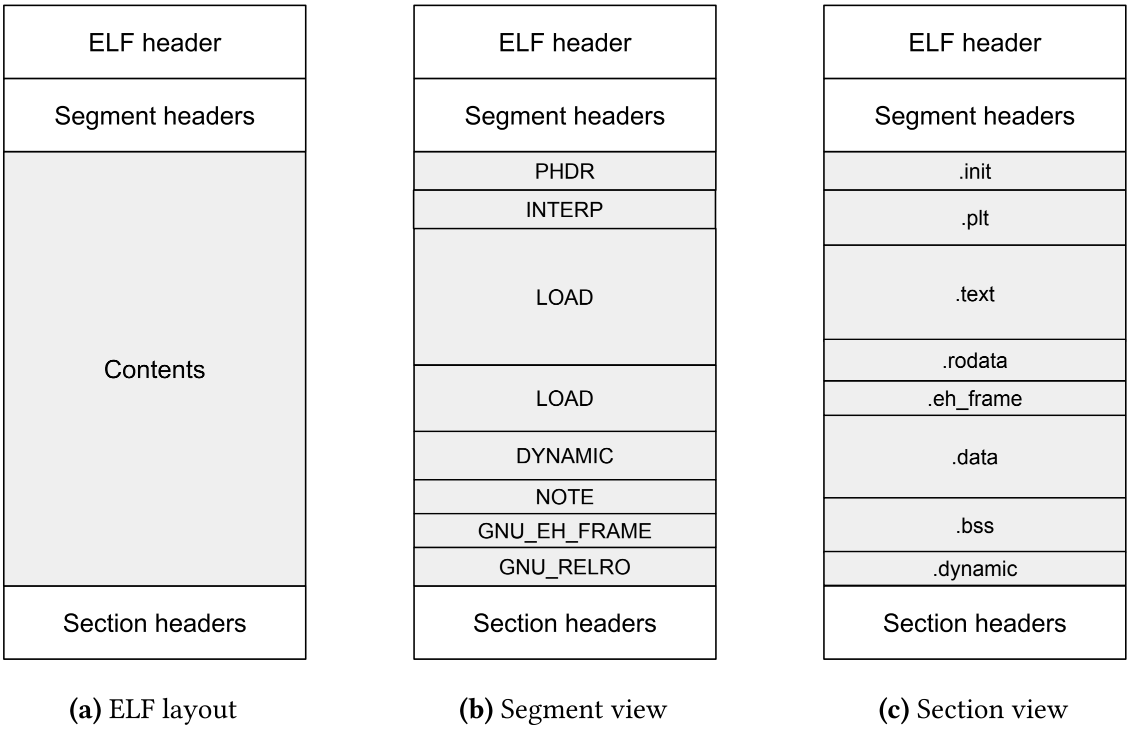


Fig. 6.2: Typical layout of an ELF file. The segment (program) and section headers provide two different views that can be used to interpret the file's contents. For each view, we show some of the most common headers. Note that the areas described by different segment headers often overlap. In fact, LOAD segments usually act as containers for other segment types.

extensible, and cross-platform. It consists of an ELF file header that describes two subsequent header tables which are:

- Segment headers (officially program headers). Specifies zero or more memory segments to be loaded into the main memory.
- Section headers. Specifies zero or more sections, which can be used for debugging among other tasks.

The segment and section header tables provide two different views to interpret the file's contents. These views are depicted in Fig. 6.2. Note that the format is flexible. For example, the segment header table does not necessarily need to be located directly after the ELF header.

We statically instrument binaries by leveraging the flexibility offered by the ELF format. Our goal is to extend a given ELF file F with two additional segments. The first segment will be used to write trampoline code, while the second stores coverage data. Let us refer to these segments as S_c and S_d, respectively. Extending the input file F can be achieved in the following steps:

```
ELF Header:
Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class:                             ELF64
Data:                              2’s complement , little endian
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              DYN (Shared object file)
Machine:                           Advanced Micro Devices X86 -64
Version:                           0x1
Entry point address:               0x5850
Start of program headers:          64 (bytes into file)
Start of section headers:          132000 (bytes into file)
Flags:                             0x0
Size of this header:               64 (bytes)
Size of program headers:           56 (bytes)
Number of program headers:         9
Size of section headers:           64 (bytes)
Number of section headers:         28
Section header string table index: 27
```

Fig. 6.3: ELF header example produced by `readelf` utility. Patching the file may affect the start of program headers. Also, patching should increase the number of program headers by two.

- Create the output file F' such that $size(F') \geq size(F) + size(S_c) + size(S_d) + size(T')$, where T' refers to the new segment header table.
- Copy the contents of F to F', except the section header table, which will be appended to the end of F'.
- Add a new segment header table T'. The new table maintains all the headers of T but adds two headers to describe S_c and S_d.
- Patch the ELF header of F' to point to the relocated segment header table T'. Figure 6.3 shows an ELF header, which is dumped using `readelf` utility, together with the fields that need to be modified.

Generally, these steps are effective in patching any shared library. However, the default ELF loader (ld.so), at least for the Linux systems we tested, makes them unusable as-is to patch executables, because it expects the segment header table, specifically the header `PT_INTERP`, to be part of the first loadable segment. Developing a customized loader can provide a workaround. But, this solution is intrusive and not easily portable. To address this issue, we implemented three techniques in bcov which are discussed in the following:

- **Head patching**. We instruct the linker to leave enough space for two segment headers after the original headers. This space will be used later by bcov. Note that the size of single segment header in x86-64 is 56 bytes. Hence, we only need 112 bytes. To support this feature, the user has to modify one line in the default linker script to add 112 bytes. Specifically, it is the line that sets `SEGMENT_START`.

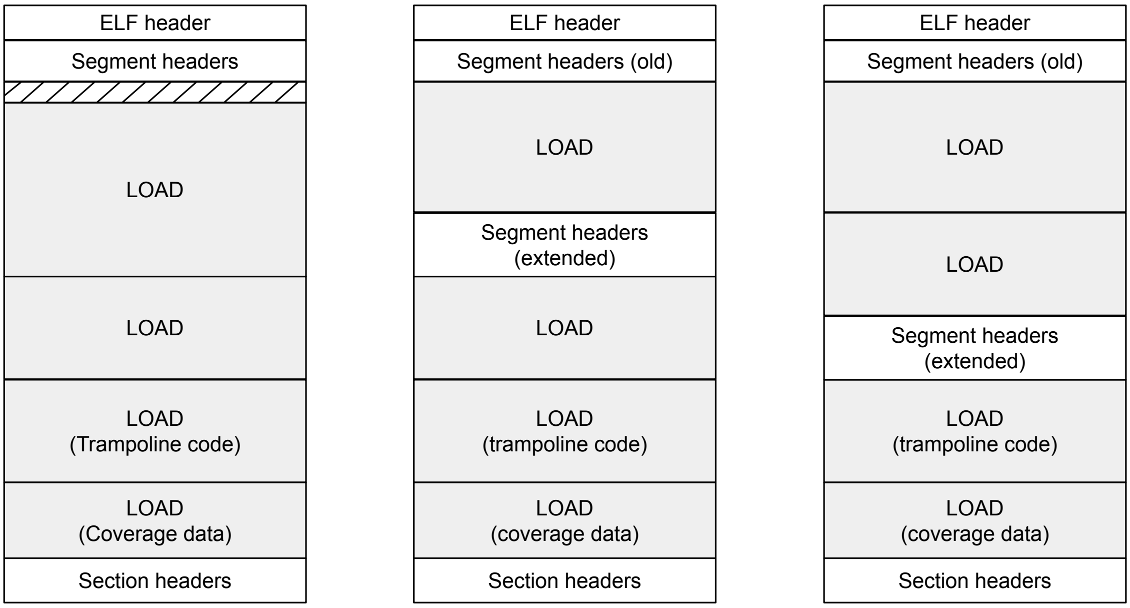
- **Middle patching**. Assuming that an ELF segment has a file offset P_{off}, a virtual address P_{addr}, and an alignment value P_{align}. Then, the system loader expects that:

  $$P_{off}\ mod\ P_{align} = P_{addr}\ mod\ P_{align}$$

  The common value of P_{align} on Linux is 0x200000, which is relatively large. Therefore, there is usually an empty space between the end of the first loadable segment and beginning of the subsequent segment. If large enough, this space can be used to relocate the segment header table. In such a case, the size of first loadable segment needs to be also increased to include the relocated table. We empirically observed that the majority of binaries in Ubuntu have 9 ELF segments or more. Patching the binary would add another two. Therefore, middle patching may require more than 616 bytes of free space after the first loadable segment, given that a single entry in the header table occupies 56 bytes. This space might not be available, especially if P_{align} was set to a low value like 4KB.

- **Overlay patching**. In the case where middle patching is not possible, we can still attempt a generalization of it, which we call overlay patching. The basic idea is to relocate the header table to a free space between any two segments, if possible, or to the end of the file's contents. However, instead of increasing the first segment's size to just include the segment header table, as in middle patching, now the size should additionally include all segments in-between. In effect, the first segment becomes overlaid on top of the other segments. This will not dramatically increase the file size. However, the memory overhead can be high as all the appended segments will be loaded twice; first, as part of the first segment and then as a standalone segment.

The previously discussed techniques explore different design trade-offs. Head patching is elegant but requires modifying the build system, which makes it generally inapplicable to patching off-the-shelf binaries. Overlay patching may solve the latter problem but have a high memory cost where one (or more) segments will be loaded twice. On the other hand, middle patching, if applicable, can offer a reasonable compromise. Figure 6.4 depicts the layout of files produced by these patching techniques.



(a) Head patching **(b)** Middle patching **(c)** Overlay patching

Fig. 6.4: ELF patching types based on the location of segment header table, (a) the linker is instructed to leave space after the original header table, (b) header table relocated to the space after the first loadable segment, (c) header table relocated to the end of original segments. In the latter two techniques, the size of the first segment must be increased to include the relocated header table. This is required to satisfy the requirements expected by the program loader.

6.4 Experiments

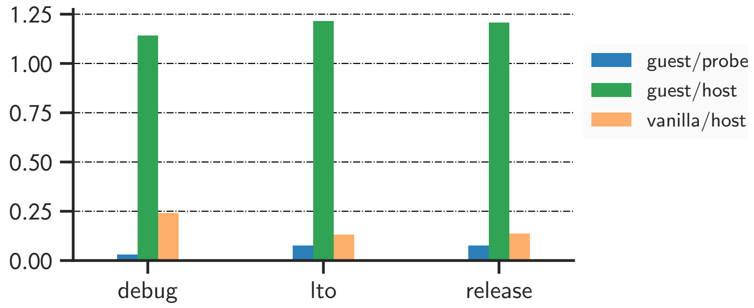
The static instrumentation techniques discussed in this chapter are implemented in the tool bcov. In this section, we evaluate them guided by the following research questions:

- What are the key characteristics of detour hosting? How is it affected by different build types, compilers, and binaries?
- Can bcov patch off-the-shelf binaries without modifying the build system?

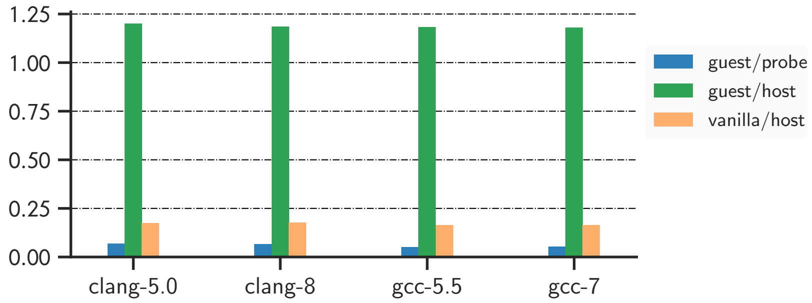
6.4.1 Detour hosting effectiveness

To evaluate the detour hosting strategy of section 6.2, we use the following metrics:

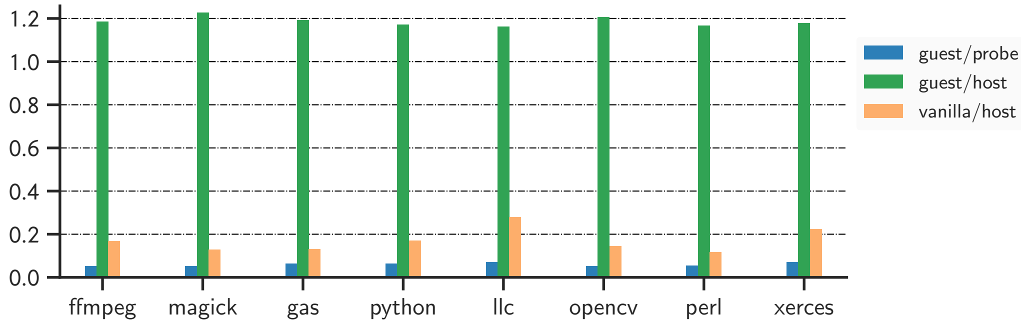
- Mean ratio of guest probes to the number of all probes (guest/probe). A very low ratio indicates that detour hosting might not be effective. In other words, disabling it might not cause a significant loss in the reported coverage.



(a) Ratios across build type



(b) Ratios across compilers



(c) Ratios across binaries

Fig. 6.5: Evaluation of detour hosting by showing how key metrics vary across different build types, compilers, and binaries, (a) guest probes are more sparse in debug builds which reduces the ratios guest/probe and guest/host, (b) there is no significant difference across the compilers, (c) differences are more noticeable but still small across the binaries.

- Mean guest-hosting ratio (guest/host). A high ratio may indicate that our strategy is effective in packing more guests in a single host, which can reduce the number of affected hosts and, subsequently, also the instrumentation overhead.
- Mean ratio of vanilla hosts among all hosts (vanilla/host). Recall that we prefer instrumenting hosts that are chosen by the current instrumentation policy over hosts that would otherwise remain intact. The reason is that instrumentation overhead is

expected anyway for the former type of hosts. A lower vanilla/host ratio indicates that our strategy is more effective at avoiding vanilla hosts.

Figure 6.5 depicts our evaluation results. We vary the build type, used compiler, and binary to see how this affects our ratios of interest. However, despite these changes, we observe that the ratios do not vary significantly. The mean guest/probe ratio is about 5% in release and link-time optimization (LTO) builds, but falls to 3% in debug builds. We attribute this to code size optimizations, which confronts bcov with smaller basic blocks and forces it to fall back to guest probes. The question now is whether a ratio like 3% is low enough to justify disabling detour hosting. Unfortunately, it is difficult to give a general answer because this depends on the instrumented program and the testing targets.

For example, a single superblock, which is represented by a guest probe, might be dominated by several superblocks in the superblock dominator graph of a particular function. Consequently, and without detour hosting, the loss in reported coverage can be high. On the other hand, a single guest probe might represent only itself in other functions, which leads to missing a single basic block. To summarize, the coverage lost by omitting a particular guest probe can vary considerably not only between different functions but also within the same function given different test inputs.

We move to discuss the remaining metrics of interest. In debug builds, the guest/host ratio is 1.14 but a bit higher in the other build types at 1.20. This indicates that guest probes in debug builds are not only fewer in number, but also further apart and, therefore, are less likely to share the same host. As for the vanilla/host ratio, it is relatively high in debug builds at 23%. This is another indicator of low probe density, which makes bcov resort to instrumenting more vanilla hosts.

6.4.2 Patching off-the-shelf binaries

The ability to patch off-the-shelf binaries is important to the usability of bcov, or any other static instrumentation tool for that matter. It allows users to track coverage without modifying the build system. In section 6.3, we discussed three techniques that we implemented to support ELF patching. These techniques are only needed to patch executables, or more precisely, binaries that have a segment of type `PT_INTERP`. Recall that, in contrast to executables, the segment header table in shared libraries does not necessarily need to be part of the first loadable segment, which greatly simplifies the task.

The instrumentation techniques implemented in bcov explore different trade-offs. Head patching requires modifying the build system. The change is small and does not add any overhead. But still, we do not expect such a change to be widely used in off-the-shelf binaries.

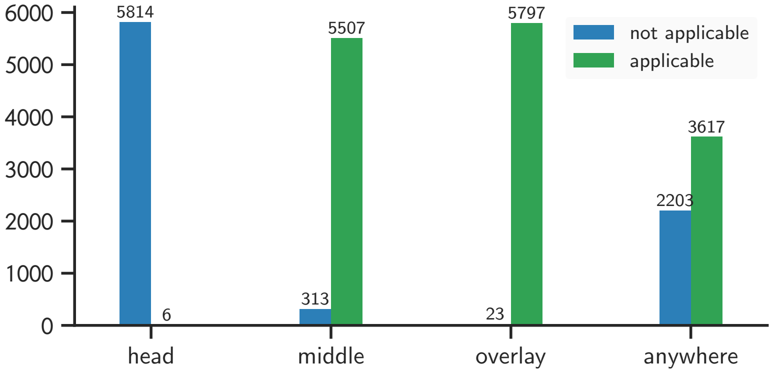


Fig. 6.6: Distribution of the applicability of patching techniques. The dataset consists of 5,820 binaries found in a typical Ubuntu installation. Expectedly, head patching is inapplicable to off-the-shelf binaries. The right-most result shows the number of shared libraries, which allow the segment header table to be relocated anywhere in the file.

At least not in the near future. This leaves us to choose between middle and overlay patching. The latter adds significant overhead because all the segments that lay between the first segment, and the relocated header table, shall be duplicated by the loader. Naturally, this begs the question of whether or not middle patching is sufficient in practice.

To answer this question, we first look at our benchmark subjects, which are 95 binaries in total. bcov successfully patched 81 binaries using middle patching. Also, it employed of overlay patching in 4 instances. However, both techniques were inapplicable to the remaining 10 binaries, of which two were, fortunately, shared libraries and, therefore, their segment header table can be freely relocated. In effect, we utilized head patching for 8 out of 95 binaries. Hence, we had to modify the build system, but that was only on infrequent occasions.

These positive results motivate a wider experiment. We developed a python script that given an ELF binary, it reports back the patching techniques applicable to it. This script additionally reports "anywhere" for shared libraries to indicate that their segment header table can be freely relocated as long as it does not overwrite the data of original segments. Then, we collected a large dataset of ELF binaries by searching the default executable and library paths in a typical Ubuntu workstation used by the author. This dataset of 5,820 binaries was given to our checker script as input.

Figure 6.6 depicts the distribution of our results. It clearly reveals that bcov, and static instrumentation in general, is largely applicable, without requiring any modifications to how software is built. Middle patching would work for 94.62% of the binaries. This percentage goes to over 99.99% in overlay patching, which remains a reasonable option despite its higher memory overhead. Note that 3617 (or about 62%) of the binaries in our dataset are shared libraries. We note again that libraries can be easily patched by bcov without incurring additional memory overhead.

6.5 Discussion and Related Work

6.5.1 Detour hosting

The variability in the size of x86-64 instructions poses a significant challenge. It allows basic blocks to be smaller than 5 bytes, which is too small to fit a detour instruction. To address this challenge, some tools rely on inserting `INT3`, a one-byte instruction that is usually used by debuggers to set breakpoints. Upon reaching an `INT3`, a running process will trigger a `SIGTRAP` signal, which must be handled either by the same process, as used in DynInst [44], or a monitoring process, as used in Untracer [94]. This mechanism comes at a high cost as every trap involves context switching by the operating system. Additionally, it may affect the functionality of programs that implement custom signal handling.

In the design of bcov, we relied on detour hosting, instead. The basic idea is simple; A basic block needs to fit a short detour (2 bytes) that targets a long detour (5 bytes) located nearby, more precisely, within a range of about ±128 bytes. This technique is already known in the reverse engineering and systems programming communities. For example, the library `syscall_intercept` [116], which is part of Intel's persistent memory programming kit [57], uses a limited form of detour hosting where the long detour can be inserted only in the padding area of a given function. However, patching would fail if such an area does not exist or is not close enough.

Inserting a long detour in the body of a nearby host basic block assumes that the start of the host is precisely known based on the CFG. Otherwise, we risk breaking the binary. In this work, we addressed this challenge by recovering the CFG with high precision outperforming the leading binary analysis tools, as demonstrated in the evaluation of our jump table analysis of Section 4.3 and the transparency results of Section 7.3. This precision enabled bcov to choose a host for a given guest probe in a way that is both correct and optimized.

Our detour hosting strategy targets a sweet spot balancing performance and relocation overheads. It was able to host up to 14 guests in a single host. Also, vanilla hosts represent only about 20% of all the hosts. That is, the strategy is effective in targeting hosts where we expect relocation overhead anyway. Moreover, it was capable of hosting about 94% of all the guests. We observed that the remaining guests are usually located in relatively small functions, having less than five basic blocks. In such a case, relocating the entire function, where coverage update code can be inlined, maybe a better alternative to detour hosting.

Generally, the fixed size of instructions in RISC ISAs alleviates the need for detour hosting. However, their addressing range can be significantly lower than the ±2GB offered by x86-64. Note that we patch each ELF module individually. This means that we only need an addressing range that is large enough to reach our patch code segment from the original code. For example, a range of 60MB would be sufficient for our largest benchmark. As a consequence, AArch64 can accommodate a wide variety of binaries as it offers a detour range of ±128MB. In comparison, AArch32 offers a range of just ±32MB. In such a case, a single detour instruction might not be sufficient. Additional options need to be investigated, such as relocating functions, literal pools, and changes to linker scripts.

On a similar note, x86-64 allows us to update coverage data using a single `mov` instruction, which has a memory operand with 32-bit offset. Generally, emulating the same functionality in RISC ISAs requires more instructions and would clobber at least one register. Nevertheless, viable alternatives do exist. For example, it is possible to update coverage data in AArch64 using the following sequence:

```
adrp x1, #pimm1    ; load page-aligned address
strb w1,[x1],#pimm2; set coverage byte to zero
```

Basically, `ardp` loads a page-aligned address into `x1` with a comfortable pc-relative range of ±4GB. Then, `strb` would use the least significant byte of `x1`, which always has a zero value, to update coverage where `pimm2` indexes into a 4KB page. This assumes that coverage data is initialized to a value other than zero. Saving and restoring the clobbered register, which is `x1` in our example, is not always necessary. A liveness analysis can help us acquire a register that stores a dead value. Similar analyses are already implemented in DBI tools like Pin [102] and DynamoRIO [24].

6.5.2 ELF patching

In Section 6.3, we discussed techniques for extending an existing ELF binary by allocating additional code and data segments. Our tool, bcov, requires only two segments. Also, we believe that supporting these two segments is sufficient for the majority of static instrumentation applications. It is worth noting that other ELF patching tools, like the LIEF library [107], may attempt to be more generic by adding support for more than two segments. However, we believe that this would complicate the implementation and increase the memory overhead, all without providing significant added value.

We have shown that the ELF patching techniques implemented in bcov, namely, head, middle, and overlay, are powerful and generic enough to apply to off-the-shelf binaries. This an encouraging result demonstrating the usability of our tool and static binary instrumentation in general. We are not aware of any other instrumentation tool with such broad applicability. For example, LIEF [107] supports only overlay patching, which comes at a considerable memory overhead, as discussed previously. Moreover, we are not aware of any tool that supports the head and middle patching techniques.

Instrumentation using trampolines is known for a long time [50, 46]. It is typically used in restricted applications such as API hooking in Detours [40] and CWSandbox [106]. In comparison, we systematically use trampolines at a fine granularity to instrument individual basic blocks. Also, our tool does not require compiler support, thanks to the precision of the recovered CFG. Note that compilers offer limited support for inserting padding `nop` instructions, which can be later overwritten by detours. For example, to specify the size of the padding area at function entry, `gcc` offers the command line option `-fpatchable-function-entry`.

As an alternative to trampolines, several works have recently considered static instrumentation using reassembly [124, 42, 133] and recompilation [96, 4, 127]. Both techniques allow instrumentation code to be inlined in the recovered assembly and IR, respectively. Instead of using detours, this would allow us to directly inline coverage update code. However, code inlining means that code references need to be relocated and fixed, e.g., in CFI records, which is challenging to correctly implement, as distinguishing references from scalars is an undecidable problem in general. In comparison, detours maintain reference stability and, thereby, allowed us to scale to large C/C++ binaries transparently. Additionally, trampolines make it easy to map analysis results of instrumented binaries back to original ones.

Chapter 7

Tool Evaluation

Tool Implementation; Experimental Setup; Evaluation Results; Discussion and Limitations.

7.1 Tool Implementation

Our contributions are implemented the in the tools Spedi and bcov. In this section, we discuss the implementation of both tools.

7.1.1 Implementation of Spedi

The tool Spedi is implemented in about 4,500 LoC in C++. The source code is publicly available: `https://github.com/abenkhadra/spedi`. This tool implements the speculative disassembly method proposed in Chapter 2, and the function identification technique of Section 3.3, which focuses on identifying functions in the binary-only settings.

The architecture of Spedi is depicted in Fig. 7.1. Basically, it consists of an ELF reader, a disassembly framework, and the core engine. We adapted our ELF reader, with few modifications, from [33] . ELF is a popular standard format for binaries in Unix-like operating systems. However, our method is generic and is not tied to ELF or any other binary format. To decode bytes to corresponding assembly instructions, we use the `capstone` framework [29]. It is a multi-architecture disassembly framework, which makes our implementation easier to port to other ISAs supported by `capstone` like x86 and PowerPC. Our main contribution is the core speculative disassembly engine.

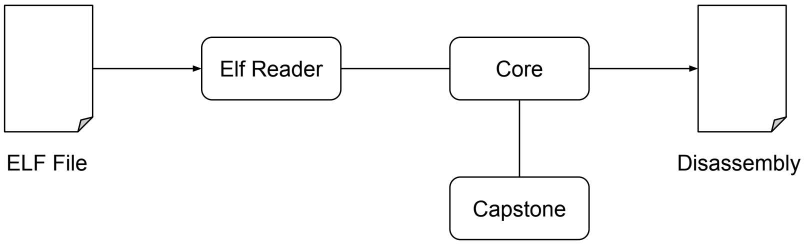


Fig. 7.1: Software architecture of Spedi tool.

7.1.2 Implementation of bcov

We implemented the majority of the techniques discussed in this work in the tool bcov, which is publicly available: `https://github.com/abenkhadra/bcov`. Our tool accepts an ELF module (executable or shared library) as input. It starts with a set of module-level analyses such as reading function definitions, parsing CFI records, and building the call graph. Our non-return analysis implementation is similar to [92]. We omit the details as they are not part of our core contribution. Then, bcov moves to function-level analyses such as building the CFG (including jump tables), dominator trees, and superblock dominator graph.

Probes are determined based on the instrumentation policy set by the user. bcov can be used for patching or coverage reporting. The latter mode requires a data file dumped from a patched module. The instrumentation policy used for coverage reporting must match the one used for patching. For example, to patch a binary like `perl`, the user can issue,

```
bcov -m patch -p any -i perl -o perl.any
```

The tool mode is set using the option `-m`. The instrumentation policy (option `-p`) can be set to `any`, which refers to the any-node policy, or `all` which refers to the leaf-node policy.

Coverage data can be dumped by injecting `libbcov-rt.so`, our run-time library, using the LD_PRELOAD mechanism. For example, the following command produces a dump file that has the extension '.bcov' in the current working directory,

```
export LD_PRELOAD="[full-path-to-bcov-rt]/libbcov-rt.so"
./perl.any -e 'print "Hello, bcov!\n"'
```

The dump data file can be supplied to bcov for coverage reporting,

```
bcov -m report -p any -i ./perl -d perl.any.bcov > report.out
```

Currently, bcov cannot persist analysis results to disk. Therefore, the original binary must be re-analyzed to report coverage. Coverage will be reported for each basic block in the file `report.out`. The data in each line lists (1) BB address, (2) BB instruction count, (3) is covered, and (4) is fallthrough ,i.e., does not terminate with a branch.

We implemented the modern SEMI-NCA dominator tree algorithm [49] and Tarjan's classical SCC algorithm. We used capstone [29] for disassembly and implemented a wrapper around unicorn [121] for microexecution. In total, this required about 17,000 LoC in C++ (testing code excluded). The run-time library bcov-rt is implemented in C in 250 LoC.

7.2 Experimental Setup

We discuss in this section the experimental setup used to evaluate our tools.

7.2.1 Experimental Setup of Spedi

Our experiments have been conducted on a Linux machine with Intel® Core™ i7-4800 processor and 16 GB of RAM. The experimental binaries are selected from the popular benchmark suites Coreutils (available at: `https://www.gnu.org/software/coreutils/`) and CoreMark Pro (available at: `https://www.eembc.org/coremark-pro/`). They are cross-compiled for ARM® Thumb using gcc versions 4.8 and 5.1 respectively. This shows that our results are not limited to a specific compiler version. We selected the biggest five binaries in each benchmark suite. These binaries were previously described in Table 2.1. Compilation was done at the highest optimization option (`-O3`) to give the compiler enough chance to challenge the disassembler.

Our results are reported for the disassembly of the `.text` section in each executable binary. Executables are dynamically linked to demonstrate that our techniques remain effective even when not all functions are available. To compare our results to alternative disassemblers, we picked IDA Pro, the leading industry disassembler, and `objdump` which is a popular disassembly tool from the GNU binutils package (available at `https://www.gnu.org/software/binutils/`). We used IDA Pro v6.9.1 in our experiments. Experimental data was extracted from IDA Pro using its IDAPython API. IDA^{s} refers to results obtained from IDA Pro for non-stripped binaries, i.e., with linker symbols available. Otherwise, all results are obtained from stripped binaries.

7.2.2 Experimental Setup of bcov

For our evaluation, we used eight modules which are summarized in Table 7.1. They are selected from popular open-source packages offering diverse functionality. We compiled each module using four compilers in three different build types. Specifically, we used the compilers gcc-5.5, gcc-7.4, clang-5.0, and clang-8.0. This gives us a representative snapshot of the past

Module	Package	Lang.	Domain
gas	binutils-2.32	C	Assemblers
perl	perl-5.28.1	C	Interpreters
python	cpython-3.7.3	C	Interpreters
libMagickCore	ImageMagick-7.0.8	C	Image processing
ffmpeg	FFmpeg-4.1.3	C	Video processing
libxerces-c-3.2	xerces-c-3.2.2	C++	XML processing
libopencv_core	opencv-4.0.1	C++	Computer vision
llc	llvm-8.0.0	C++	Compilers

Table 7.1: Selected subjects used to evaluate bcov. Used recent package versions.

three years of developments in gcc and clang respectively. The build types are debug, release, and lto. The latter refers to link-time optimizations. Compiler optimizations were disabled in debug builds and enabled in release and lto builds. Enabled optimizations depend on the default options of their respective package, which can be at levels O2 or O3.

Compilation results in 12 versions of each module and a total of 95 binaries. [1] Our tool was capable of patching 88 binaries without modifying the build system. However, we had to modify the linker script in 7 instances where relocating ELF program headers was not possible. We instructed the linker to leave 112 bytes, which is enough for our segment headers, after the original program headers. This change is small affecting only one line in the linker script. The bcov-rt runtime was injected using the LD_PRELOAD mechanism. All experiments were conducted on an Ubuntu 16.04 PC with Intel® i7-6700 CPU and 32GB of RAM.

7.3 Evaluation Results

This section is dedicated to the evaluation of bcov as standalone tool. We separately evaluated several techniques implemented in bcov, like sliced execution, in the previous chapters. Recall that the techniques implemented in Spedi are already evaluated in Section 2.5 and Section 3.4. Our evaluation of bcov is guided by the following research questions:

RQ1 Can bcov scale to large real-world binaries transparently?

RQ2 What is the instrumentation overhead in terms of performance, memory, and file size?

RQ3 To what extent can bcov provide better efficiency in comparison to its direct alternatives, namely, DBI tools?

RQ4 Can bcov accurately report binary-level coverage?

---

[1] Compiling llc with gcc-5.5 in lto build resulted in a compiler crash.

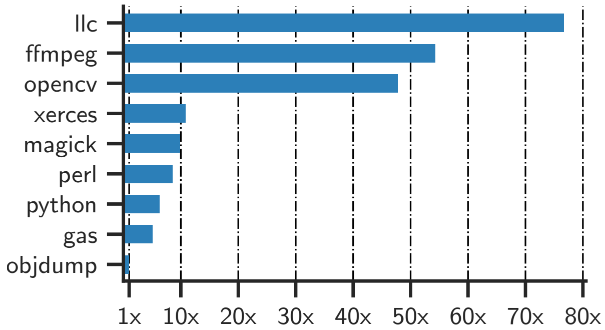


Fig. 7.2: Comparing the code size of our subjects to objdump (code size about 339KB). Code size reported with GNU size utility.

We investigate each of these research questions in the following sections.

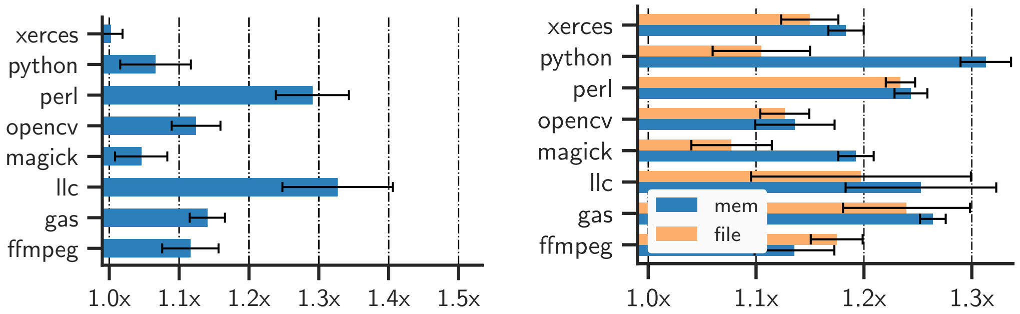
7.3.1 Scalability and transparency

Our choice of subjects directly supports the claim regarding the scalability of the proposed techniques. Figure 7.2 shows a comparison in terms of code size relative to objdump, a commonly used subject in binary analysis research. Note that bcov can analyze and patch llc, our largest subject, in 30 seconds. In our experiments, we used bcov to instrument all functions available in the `.text` section. This amounts to more than 1.6×10^6 functions across 95 binaries. The policies leaf-node and any-node have been applied separately, i.e., subjects were instrumented twice.

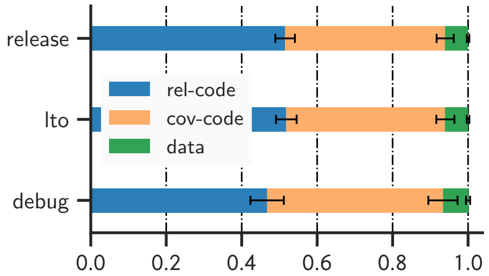
Transparency is important in coverage instrumentation. This practically means that bcov should not introduce test regressions. We evaluated this criterion by replacing original binaries with instrumented versions and re-running their test suites. *Our instrumentation did not introduce any regressions* despite the fact that (1) we systematically patch all functions, even compiler-generated ones, and (2) our benchmark packages include extensive test suites. For example, the perl test suite runs over one million checks.

7.3.2 Instrumentation overhead

Figure 7.3 depicts the instrumentation overhead relative to the original binaries. The average performance overhead of the leaf-node and any-node policies are 8% and 14%, respectively. The overhead is measured based on the wall-clock time required to run individual test suites, e.g., run “make test” to completion. This covers the overhead associated with instrumentation in addition to dumping coverage data to disk. The latter overhead varies depending on the number of processes spawned during testing.



(a) performance overhead **(b)** size overhead



(c) size overhead distribution

Fig. 7.3: Overhead of the any-node instrumentation policy. (a) performance overhead accounts for instrumentation and dumping coverage data, (b) memory and file size overhead (c) distribution of memory overhead between code (relocated and coverage update) and coverage data.

For example, all opencv tests are executed within a single process that dumps coverage data only once. In contrast, unit testing of llc spawns over 7,500 processes in about 40 s. This results in dumping 4 GB of coverage data, which significantly contributes to the overall delay. Online merging of coverage data might reduce this disk IO overhead. In such settings, and before dumping new coverage data, the current process try to merge its current data with previously dumped data. This will reduce the required space at the cost of slowing down the dumping of data to disk. To give a better intuition, we note that without online merging, llvm-cov would dump over 320 GB of coverage (and profiling) data for the same benchmark.

The average memory and file size overheads introduced by bcov are 22% and %16, respectively. We measure the memory overhead relative to loadable ELF segments only since bcov does not affect the run-time heap or stack. Note that other static instrumentation techniques need to duplicate the code segment [4, 82]. This suggests that the overhead of bcov is reasonable. Coverage data represents only 6% of the memory overhead. It is worth noting that compiler optimizations can force bcov to relocate more instructions. This might be due to emitting smaller basic blocks. However, our static instrumentation techniques are effective in reducing the difference in relocation overhead between debug and optimized builds, as shown in Figure 7.3c.

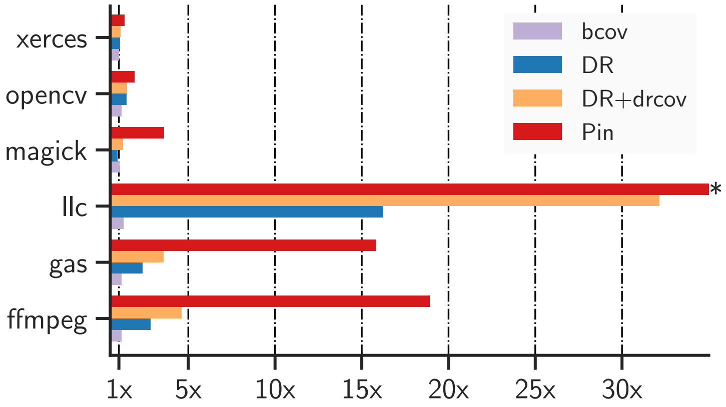


Fig. 7.4: Comparing the performance overhead of Pin, DynamoRIO (DR), drcov, and bcov. Omitted perl and python as DR was unable of completing their test suite runs. (*) The actual overhead of Pin for llc is over 130x.

7.3.3 Comparison with DBI tools

Pin and DynamoRIO (DR) are the most popular DBI tools. Both act as a process virtual machine that instruments programs while JIT emitting instructions to a code cache. This complex process creates the following sources of overhead: (1) JIT optimization, and (2) client instrumentation. To evaluate this overhead on our test suites, we installed the latest stable releases of both tools, namely, Pin v3.11 and DR v7.1. We then replaced each of our subjects with a wrapper executable. In the case of shared libraries, we replaced their test harness with our wrapper. In doing so, we make the test system run our wrapper, which, in turn, runs its corresponding original binary, but under the control of a DBI tool. The wrapper reads a designated environment variable to choose between Pin and DR.

Fig. 7.4 depicts the performance overhead of Pin and DR without client instrumentation. It also shows the overhead of DR after enabling drcov, its code coverage client. Note that Pin does not have a similar coverage client built-in. The overhead is measured relative to original binaries and is averaged for four different release builds. Both tools introduced regressions on perl and python. However, DR made tests hung on perl and crashed on the python test suite. This highlights the challenges of maintaining transparency in DBI tools. Note that the DBI overhead of executable subjects is significantly higher than that of shared libraries. This can be attributed to the start-up delay, which dominates in short-running tests. The average performance overheads of Pin and DR are 29.1x and 4.1x, respectively. Enabling drcov increases DR's average overhead to 7.3x. Our experiments show that bcov can provide drastically better performance, transparency, and usability.

7.3.4 Accuracy of the reported coverage

We evaluate the accuracy of the coverage reported by our tool via a comparison with a corresponding instruction trace. Note that for the same test input, non-determinism may cause spurious errors in the coverage report if we compare original binaries separately to instrumented ones. For example, repeatedly printing a simple "Hello World" message using perl produces different instruction traces. Therefore, we obtain the trace from binaries that are instrumented with the any-node policy.

Initially, we obtained the ground truth traces using Intel PT (IPT). To this end, we collected about 2,000 sample tests from our test suites. Running these tests produces 104 GB of IPT data and 444 MB of bcov coverage data. We used the standard perf tracing facilities in the kernel v4.15, and later also kernel v5.3. We tried many IPT configurations and restricted ourselves to tests terminating in ≤ 5 seconds. Despite these efforts, we could not reliably evaluate bcov due to non-deterministic loss in IPT data. After all, disks might just be incapable of keeping up with the CPU [72].

We then turned to drcov to obtain the ground truth. This DR client dumps the address of encountered basic blocks heads, i.e., first instructions. We leverage the fact that our instrumentation does not modify BB heads. Based on this, we expect BBs reported as covered by bcov to appear in the trace of drcov. We consider these BBs to be true positives (TP). On the other hand, a BB reported by bcov that was not found in the trace represents a false-positive (FP). Similarly, a false-negative (FN) is a tracked BB that was missed by bcov. Both FPs and FNs represent errors in the reported coverage.

Also, we take into account the fact that drcov reports the heads of *dynamic* BBs. This means that should A and B be consecutive BBs where A is fallthrough, i.e., does not end with a branch, drcov might only report the head of A. Our evaluation method is conservative given the potential overapproximation of the CFG.

Our results are shown in Table 7.2. They are based on running the test suites of subjects compiled with gcc-7 in release build. The results are representative of other build types. The subjects are instrumented with bcov and also run under the control of DR's drcov. We list the average/maximum of TPs across all test processes. For example, the average number of TP BBs among 7,862 llc processes is 45,184.5, and the maximum is 90,952. The average precision and recall across all subjects are 99.97% and 99.95%, respectively. This results in an average F-score of 99.86%. Our evaluation suggests that the reported coverage errors are practically negligible. Nevertheless, there is still room for further improvement. Specifically, improving CFG precision and detour hosting can reduce FPs and FNs respectively.

module	proc. #	drcov size	bcov size	BB	Inst.	TP BB	TP Inst.	Precision	Recall
xerces	80	12.34	4.32	116378	420096	9523 / 21927	40651 / 92144	99,98%	99.42%
magick	58	7.71	2.90	125521	521107	5689 / 20709	21614 / 83444	99,98%	99.94%
llc	7862	3481.97	4176.16	1067151	4343021	45184 / 90952	257209 / 461656	99,98%	99.68%
gas	1235	71.94	38.56	60511	220447	2916 / 5015	11045 / 19578	99.93%	99.67%
ffmpeg	3309	423.45	762.39	496404	3050228	9682 / 14489	41439 / 63591	99.98%	99.94%

Table 7.2: Evaluating the accuracy of bcov based on drcov traces. We show the number of processes spawned during testing, corresponding dump sizes in MB, and the total number of BBs and instructions in original binaries. Both tools dump one data file per process. For each subject, we list the average/maximum of true positives (TP). FPs and FNs are also considered by listing the average precision and recall, respectively. DR could not complete the test suite runs of `perl` and python. Omitted opencv as drcov's data was invalid due to a bug.

7.4 Discussion and Limitations

CFG precision. The precision of the CFG recovered by bcov can significantly affect the accuracy of its coverage report. While the implemented jump table and non-return analyses significantly increase CFG precision, they are still not perfect. Specifically, our prototype might still miss jump tables, albeit only in a few situations. Also, while our experiments show that the non-return analysis implemented in bcov is comparable to that of IDA Pro, both tools face the challenge of *may-return* functions. Such functions usually, but not always, return to their caller. We encountered a may-return function in `perl` that is particularly noteworthy. It is function `Perl__force_out_malformed_utf8_message` shown in Fig. 7.5. In one binary, it is called 88 times out of 89 total, with the argument `die_here` set, i.e, it will not return. Developers can signal to the compiler that a particular call will not return using, for example, `__builtin_unreachable`. Such information is not available in the binary, so we simply assume that all calls to may-return functions are return calls. As a result, bcov may spuriously report basic blocks that follow a may-return call as covered.

Conditional instructions. Simple `if` statements might be compiled to conditional (predicated) instructions. For example, several C/C++ compilers may use a single conditional instruction, like `cmov` and `setge`, to assign a value to the variable `c`,

```
if (a < b) c = 0; else c = 1;
```

Binary-level coverage analysis tools like bcov can report whether a particular `cmov` is covered. However, this does not imply that it actually took effect, i.e., modified the target value. Tracking such effects is out of our scope, as we are not concerned with *data-flow* coverage. Note that source-level coverage tools like `gcov` have the same limitation.

Nevertheless, bcov can be extended to handle such cases. Basically, we can check that the assignment of a particular `cmov<condition>` did take effect by inserting a detour tar-

```
if (c_len == (STRLEN)-1) {
    _force_out_malformed_utf8_message(p, p_end,
    utf8n_flags,
    1 /* 1 means die */ );
    NOT_REACHED; /* NOTREACHED */
}
if (c > 255 && OP(n) == ANYOFL && !
   ANYOFL_UTF8_LOCALE_REQD(flags)) {
    _CHECK_AND_OUTPUT_WIDE_LOCALE_CP_MSG(c);
}
```

Fig. 7.5: Example of a may-return call taken from function `S_reginclass` in `perl` v5.28. The call to `_force_out_malformed_utf8_message` will not return since the boolean flag `die_here` is set to 1. The macro `NOT_REACHED` expands to `__builtin_unreachable` in `gcc` and `clang`. That is, it hints the compiler that this particular call will not return.

geting a *guarded* coverage update instruction. The guard is simply a short jump `j<negated condition>`. In this way, the coverage update instruction will be executed only if the guard `<condition>` is satisfied. The detour can be flexibly inserted around a particular `cmov`, but we must ensure that the guard `jmp` reads the exact CPU flags of `cmov`. However, this scheme might not be practical in ISAs like ARMv7, where the majority of instructions can be predicated.

Threats to validity. The subjects we selected for evaluation are written in both C and C++ by different software teams. Also, they are publicly available, well maintained, and include extensive test suites. Moreover, our subjects are relatively large and offer diverse functionality. Based on this, we believe that we addressed the *internal* threats to validity to a satisfactory level. However, similar to the majority of scientific experiments, there might still be concerns regarding the *external* validity of our results. Our subjects are representative of user-space software in Linux. However, generalizing our results to other operating systems requires further investigation. Also, our static approach cannot be directly applied to dynamic code, e.g., JIT code. While the simple primitives we use to implement detours and update coverage apply to system software like kernel modules, special considerations might exist, e.g., overlapping instructions.

Chapter 8

Conclusion and Outlook

We conclude by summarizing our key contributions and discussing potential future work.

Speculative disassembly. Disassembly is the first step for virtually all static binary analyses. We have shown that speculative disassembly can provide a principled framework to accurately identify instructions in stripped binaries. Our method proceeds in two main phases. First, we recover all potential basic blocks in the binary and group them into maximal blocks. Second, we use conflict analyses to identify the most likely basic blocks. Our experiments demonstrate interesting results outperforming the leading industry disassembler, IDA Pro. In future work, we can implement further analyses to assist in the case where conflict analyses are insufficient. For example, a statistical bi-gram analysis can further indicate that a particular sequence of instructions is more likely.

Function identification. The function is a core abstraction in software analysis. In the case where symbol information is available, we have shown how functions can be easily identified based on the contiguous function model. In stripped binaries, Call-Frame Information (CFI) records are a valuable source of function definitions. However, this source is often overlooked in the literature. Therefore, we show that CFI records are largely complete in stripped off-the-shelf binaries, which further widens the applicability of our techniques. Furthermore, we proposed a CFG-based identification technique to address the case where binaries are stripped from CFI records.

Jump table analysis. Tracking coverage requires a fine-grained level of instrumentation, which requires, in turn, a precise analysis for jump tables. To this aim, we proposed sliced microexecution as a novel technique that combines backward program slicing with microexecution. Basically, for each indirect jump, we try hard to falsify a few hypotheses about the behavior of jump tables. If we are not successful, then we can conclude with high confidence that the indirect jump at hand represents a jump table. Our technique does not rely on static approximations or brittle heuristics. The conducted experiments show that sliced microexecution provides superior precision and robustness compared to IDA Pro. Additionally, we moved a step further by instrumenting jump table entries for the first time to our knowledge, which further pushed the state of the art in binary instrumentation.

Probe optimization. Instrumenting all basic blocks is both expensive and unnecessary. Therefore, a probe pruning technique should be implemented to improve efficiency. We have chosen Agrawal's technique and demonstrated that in comparison to alternatives, it provides superior accuracy while instrumenting a comparable number of basic blocks. Further, we leveraged the flexibility that superblocks provide to optimize the selection of probes. Our probe selection scheme is simple yet effective in reducing overhead. We believe that more sophisticated selection techniques can offer additional efficiency gains, for example, by avoiding loop heads when possible.

Static instrumentation. In this work, we developed several techniques to transparently instrument binaries with low overhead. We showed that a limited number of simple yet powerful instruction rewriting primitives, examples of which are depicted in Table 5.2, can allow code to be transparently relocated to the trampoline segment. Additionally, the unprecedented precision of CFG recovered by bcov has opened new opportunities for exploiting padding bytes and systematically hosting detours in neighboring basic blocks. These techniques allowed us to relocate only what is necessary, instead of relocating entire functions as implemented in PEBIL [82], for example. Also, we discussed how to patch an ELF file by inserting additional segments and shown that it is generally possible to patch ELF binaries without modifying the build system.

Outlook. If we had to summarize the core contribution of this work in one sentence, then it will be: *coverage analysis at the binary level can be efficient and transparent without requiring compiler support*. To validate this claim, we combined several techniques and efficiently implemented them in software prototypes. Then, we used this software to conduct large experiments on several popular packages like FFmpeg and LLVM. Our prototypes are publicly available and open source. This lays the groundwork for interesting future work that can follow many directions. For example, researchers might want to explore other applications of static instrumentation beyond coverage analysis like profiling and fuzzing.

Additionally, insisting on not requiring compiler support had allowed us to gain generality, but this comes at a high cost in instrumentation efficiency. In fact, our experiments show that inserting detours and subsequent code relocation account for 96% of the overall performance overhead. This overhead can be largely avoided by inlining coverage update instructions directly in the original code. However, code inlining would probably require some support from the compiler to correctly relocate basic blocks. For example, the post-link optimizations implemented in BOLT [99], which include basic block reordering, require the compiler to emit code relocations, among other information.

To conclude, coverage analysis was traditionally limited to compiler instrumentation and dynamic binary instrumentation, we hope that this work has convinced you that static binary instrumentation is a viable alternative even without compiler support.

Chapter 9

Zusammenfassung

Die Code-Abdeckungsanalyse spielt eine wichtige Rolle im Software-Testprozess. Strukturelle Abdeckungsmetriken wie Statement- und Verzweigungsabdeckung können das Vertrauen in ein zu testendes Programm (PUT) stärken oder zumindest ungetesteten Code identifizieren. In letzte Jahre hat die bemerkenswerte Effektivität des Abdeckungsfeedbacks ein breites Interesse an Feedback-gesteuertem Fuzzing ausgelöst. In dieser Arbeit diskutieren wir statische Instrumentierungstechniken, die eine Abdeckungsanalyse auf binärer Ebene ohne Compilerunterstützung ermöglichen. Wir zeigen, dass die vorgeschlagenen Techniken präzise, effizient und transparent sind und den Stand der Technik deutlich übertreffen.

Wir implementieren diese Techniken in zwei Tools, nämlich Spedi und bcov. Beide Tools sind quelloffen und öffentlich verfügbar. Spedi zeigt, dass die Disassemblierung und Funktionsidentifikation von gestrippten Binärdateien ohne Rückgriff auf externe Informationen sehr genau sein kann. Wir bauen auf diesen wichtigen Ergebnissen in bcov auf, wo wir x86-64 ELF-Binärdateien statisch instrumentieren, um Code-Abdeckung zu ermitteln. Die Verbesserung der Effizienz und die Skalierung auf große, existierende Software erforderte jedoch eine orchestrierte Lösung, die viele Techniken kombiniert.

Zuerst bringen wir eine bekannte Probe-Beschneidungstechnik zum ersten Mal zur Binär-Instrumentierung ein und nutzen effektiv den Begriff der Superblöcke, um den Overhead zu reduzieren. Zweitens führen wir Sliced Microexecution ein als eine robuste Technik für die Analyse von Sprungtabellen, die CFG-Präzision verbessert und die Instrumentierung von Sprungtabelleneinträgen ermöglicht. Zusätzlich stellen die kleineren Anweisungen in x86-64 eine Herausforderung für das Einfügen von Detours dar. Um diese Herausforderung zu meistern, nutzen wir Padding Bytes aggressiv aus. Außerdem führen wir ein Greedy-Algorithmus ein, um systematisch Detours in benachbarte Basisblöcke einzubauen.

Wir evaluieren bcov auf einem Korpus von 95 Binärdateien, die aus acht populären und gut getesteten Programmen wie FFmpeg und LLVM kompiliert wurden. Zwei Instrumentierungsverfahren mit unterschiedlicher Kantengenauigkeit werden verwendet, um alle Funktionen in diesem Korpus zu patchen - über 1,6 Millionen Funktionen. Unsere präzises Verfahren hat einen durchschnittlichen Leistungs- und Speicher-Overhead von 14 % bzw. 22 %. Instrumentierte Binärdateien führen keine Testregressionen ein. Die Genauichkeit der berichteten Abdeckung ist hoch mit einem durchschnittlichen F-Score von 99,86 %. Schließlich ist unsere Sprungtabellenanalyse vergleichbar mit der von IDA Pro auf gcc-Binärdateien und übertrifft sie auf Clang-Binärdateien.

Unsere Arbeit zeigt, dass statische Instrumentierung einzigartige Vorteile im Vergleich zu etablierten Methoden wie Compiler-Instrumentierung und dynamischer Binärinstrumentierung bieten kann. Sie öffnet auch die Tür für viele interessante Anwendungen der statischen Instrumentierung, die weit über die Code-Abdeckungsanalyse hinausgehen können.

9.1 Zusammenfassung der Techniken

Hier werden die Techniken und Ergebnisse dieser Arbeit kurz zusammengefasst.

9.1.1 Spekulative Disassemblierung

Die Disassemblierung ist der erste Schritt für praktisch alle statischen Binäranalysen. Wir haben gezeigt, dass spekulative Disassemblierung ein prinzipielles Rahmenwerk zur genauen Identifizierung von Anweisungen in gestrippten Binärdateien. Unsere Methode läuft in zwei Hauptphasen ab. Als erstes stellen wir alle potenziellen Basisblöcke in der Binärdatei und gruppieren sie in Maximalblöcke. Zweitens verwenden wir Konfliktanalysen, um die wahrscheinlichsten Basisblöcke zu identifizieren. Unsere Experimente zeigen interessante Ergebnisse, die den führenden Industrie-Disassembler, IDA Pro, übertreffen. In zukünftiger Arbeit können wir weitere Analysen implementieren, um den Fall zu unterstützen, dass Konfliktanalysen unzureichend sind. Zum Beispiel eine statistische Bi-Gramm-Analyse anzeigen kann, dass eine bestimmte Sequenz von Anweisungen wahrscheinlicher ist.

9.1.2 Funktionsidentifikation

Die Funktion ist eine Kernabstraktion in der Softwareanalyse. Für den Fall, dass Symbolinformationen zur Verfügung stehen, haben wir gezeigt, wie Funktionen auf der Grundlage des zusammenhängenden Funktionsmodells leicht identifiziert werden können. In gestrippten Binärdateien sind Call-Frame Information (CFI)-Datensätze eine wertvolle Quelle für Funktionsdefinitionen. Diese Quelle wird jedoch in der Literatur oft übersehen. Daher zeigen wir, dass CFI-Datensätze in gestrippten Standard-Binärdateien weitgehend vollständig sind, was die Anwendbarkeit unserer Techniken weiter ausweitet. Außerdem stellen wir eine CFG- basierte Identifizierungstechnik vor für den Fall, dass Binärdateien aus CFI Datensätzen entfernt worden sind.

9.1.3 Analyse der Sprungtabellen

Die Analyse der Code-Abdeckung benötigt eine feinkörnige Instrumentierung, die wiederum eine präzise Analyse für Sprungtabellen erfordert. Zu diesem Zweck haben wir die Sliced Microexecution als neue Technik vorgeschlagen, die Backward Program Slicing mit Microexecution kombiniert. Grundsätzlich versuchen wir für jeden indirekten Sprunganweisung, einige Hypothesen über das Verhalten von Sprungtabellen zu falsifizieren. Wenn uns das nicht gelingt, dann können wir mit hoher Sicherheit davon ausgehen, dass der betreffende indirekte Sprunganweisung eine Sprungtabelle darstellt. Unsere Technik verlässt sich nicht auf statische Approximationen oder spröde Heuristiken. Die durchgeführten Experimente zeigen, dass Sliced Microexecution im Vergleich zu IDA Pro eine höhere Präzision und Robustheit bietet. Außerdem sind wir einen Schritt weiter gegangen, indem wir Sprungtabelleneinträge instrumentiert, was den Stand der Technik im binären Instrumentierung weiter vorantreibt.

9.1.4 Optimierung der Probes

Die Instrumentierung aller Basisblöcke ist sowohl aufwändig als auch unnötig. Daher sollte eine Probe-Beschneidungstechnik implementiert werden, um die Effizienz zu verbessern. Wir haben die Technik von Agrawal gewählt und gezeigt, dass sie im Vergleich zu Alternativen eine höhere Genauigkeit bietet trotz die vergleichbare Anzahl von instrumentierten Basisblöcken. Außerdem haben wir die Flexibilität genutzt, die die Superblöcke bieten, um die Auswahl der Probes zu optimieren. Unsere Strategie zur Auswahl der Probes ist einfach aber effektiv bei der Reduzierung des Overheads. Wir glauben, dass ausgefeiltere Auswahltechniken zusätzliche Effizienzgewinne bieten können, zum Beispiel durch die Vermeidung von Schleifenköpfen, wenn dies möglich ist.

9.1.5 Statische Instrumentierung

In dieser Arbeit haben wir mehrere Techniken entwickelt, um Binärdateien mit geringem Overhead transparent zu instrumentieren. Außerdem haben wir gezeigt, dass eine begrenzte Anzahl einfacher, aber leistungsfähiger Anweisungen-Umschreibprimitive, von denen Beispiele in Tabelle 5.2 dargestellt sind, eine transparente Verlagerung von Code in das Trampolin-Segment ermöglichen können. Zusätzlich eröffnete die einzigartige Präzision der von bcov rekonstruierten CFG neue Möglichkeiten zur Ausnutzung von Padding-Bytes und zur systematischen Unterbringung von Umwegen in benachbarten Basisblöcken. Diese Techniken erlaubten es uns, nur das Nötigste zu verlagern, anstatt ganze Funktionen zu verlagern, wie es z. B. in PEBIL [82] implementiert ist. Außerdem haben wir besprochen, wie man eine ELF-Datei durch Einfügen zusätzlicher Segmente patchen kann, und gezeigt, dass es generell möglich ist, ELF-Binärdateien zu patchen, ohne das Build-System zu modifizieren.

9.2 Ausblick

Wenn wir den Kernbeitrag dieser Arbeit in einem Satz zusammenfassen müssten, dann wäre es sein: Die Abdeckungsanalyse auf binärer Ebene kann effizient und transparent sein, auch ohne Compiler-Unterstützung. Um diese Behauptung zu validieren, haben wir mehrere Techniken kombiniert und effizient in Software-Prototypen implementiert. Anschließend haben wir mit diesen Prototypen umfangreiche Experimente mit mehreren populären Programmen wie FFmpeg und LLVM durchgeführt. Unsere Prototypen sind öffentlich verfügbar und quelloffen. Dies legt den Grundstein für interessante zukünftige Arbeiten, die in viele Richtungen gehen können. Zum Beispiel könnten Forscher andere Anwendungen von statischer Instrumentierung jenseits der Abdeckungsanalyse wie Profiling und Fuzzing erforschen.

Außerdem haben wir durch den Verzicht auf Compiler-Unterstützung an Allgemeinheit gewonnen, was jedoch einen hohen Preis für die Effizienz der Instrumentierung bedeutet. In der Tat zeigen unsere Experimente, dass das Einfügen von Detours und die anschließende Codeverschiebung 96 % der Gesamtleistung Overhead verursacht. Dieser Overhead kann weitgehend vermiedet werden durch das Inlinen von Anweisungen zur Aktualisierung der Abdeckung direkt in den ursprünglichen Code. Allerdings würde Code-Inlining wahrscheinlich eine gewisse Unterstützung durch den Compiler, um Basisblöcke korrekt zu verschieben. Zum Beispiel sind die Post-Link-Optimierungen, die in BOLT [99] implementiert sind und die die Neuordnung von Basisblöcken beinhalten, erfordern vom Compiler “Code Relocations”, neben anderen Informationen, auszugeben.

Abschließend sei gesagt, dass die Code-Abdeckungsanalyse traditionell auf die Compiler-Instrumentierung und die dynamische Binär-Instrumentierung beschränkt war. Wir hoffen, dass diese Arbeit Sie davon überzeugt hat, dass die statische Binär-Instrumentierung auch ohne Compiler-Unterstützung eine sinnvolle Alternative darstellt.

References

[1] Hiralal Agrawal. 1994. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM symposium on principles of programming languages - POPL '94*. ACM Press, New York, New York, USA, 25–34. https://doi.org/10.1145/174675.175935

[2] Eric Allman. 2012. Managing technical debt. *Commun. ACM* 55, 5 (may 2012), 50–55. https://doi.org/10.1145/2160718.2160733

[3] Paul Ammann and Jeff Offutt. 2016. *Introduction to Software Testing*. Cambridge University Press. https://doi.org/10.1017/9781316771273

[4] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of 8th ACM European Conference on Computer Systems - EuroSys '13*. ACM Press, Prague, Czech Republic, 295–308. https://doi.org/10.1145/2465351.2465380

[5] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *25th USENIX Security Symposium*. USENIX Association, Austin, TX, 583–600. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse

[6] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *IEEE European Symposium on Security and Privacy - EuroS&P'17*. IEEE, 177–189. https://doi.org/10.1109/EuroSP.2017.11

[7] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. 2019. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium - NDSS'19*. https://www.ndss-symposium.org/ndss-paper/redqueen-fuzzing-with-input-to-state-correspondence/

[8] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX:What You See Is Not What You eXecute. *ACM Transactions on Programming Languages and Systems* 32, 6 (aug 2010), 1–84. https://doi.org/10.1145/1749608.1749612

[9] Thomas Ball and James R. Larus. 1994. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems* 16, 4 (jul 1994), 1319–1360. https://doi.org/10.1145/183432.183527

[10] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceeding of the 23rd USENIX Security Symposium*. USENIX Association, San Diego, CA, 845–860. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao

[11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 171–177. http://www.cs.stanford.edu/{~}barrett/pubs/BCD+11.pdf

[12] Robert Beers. 2008. Pre-RTL formal verification. In *Proceedings of the 45th annual conference on Design automation - DAC '08*. ACM Press, New York, New York, USA, 806. https://doi.org/10.1145/1391469.1391675

[13] M. Ammar Ben Khadra. 2017. E3Solver: decision tree unification by enumeration. arXiv:1710.07021 http://arxiv.org/abs/1710.07021

[14] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2016. Speculative disassembly of binary code. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems - CASES'16*. Pittsburgh, PA, USA. https://doi.org/10.1145/2968455.2968505

[15] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2017. goSAT: Floating-point Satisfiability as Global Optimization. In *Proceedings of Formal Methods in Computer-Aided Design - FMCAD'17*. 11–14. https://doi.org/10.23919/FMCAD.2017.8102235

[16] M. Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2020. Efficient Binary-Level Coverage Analysis. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE'20*. ACM Press, Virtual Event, USA, 1153–1164. https://doi.org/10.1145/3368089.3409694

[17] Antonia Bertolino. 2007. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering (FOSE '07)*. IEEE, 85–103. https://doi.org/10.1109/FOSE.2007.25

[18] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. 1999. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th ACM/IEEE conference on Design automation conference - DAC '99*. ACM Press, New York, New York, USA, 317–320. https://doi.org/10.1145/309847.309942

[19] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. 2019. GRIMOIRE: Synthesizing Structure while Fuzzing. In *28th USENIX Security Symposium*. 1985–2002. https://www.usenix.org/conference/usenixsecurity19/presentation/blazytko

[20] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11*. ACM Press, New York, New York, USA, 30. https://doi.org/10.1145/1966913.1966919

[21] Marcel Bohme and Soumya Paul. 2016. A Probabilistic Analysis of the Efficiency of Automated Software Testing. *IEEE Transactions on Software Engineering* 42, 4 (apr 2016), 345–360. https://doi.org/10.1109/TSE.2015.2487274

[22] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. ACM Press, New York, New York, USA, 1032–1043. https://doi.org/10.1145/2976749.2978428

[23] Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Verification, Model Checking, and Abstract Interpretation*, Ranjit Jhala and David Schmidt (Eds.). Lecture Notes in Computer Science, Vol. 6538. Springer Berlin Heidelberg, Berlin, Heidelberg, 70–87. https://doi.org/10.1007/978-3-642-18275-4

[24] Derek Bruening. [n.d.]. DynamoRIO: Dynamic Instrumentation Tool Platform. https://github.com/DynamoRIO/dynamorio

[25] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments - VEE '12*. ACM Press, New York, New York, USA, 133. https://doi.org/10.1145/2151024.2151043

[26] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. 2011. BAP: A Binary Analysis Platform. In *Computer Aided Verification (LNCS)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, 463–469. https://doi.org/10.1007/3-540-55179-4_18 arXiv:1301.4779

[27] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. 1992. Symbolic model checking: 1020 States and beyond. *Information and Computation* 98, 2 (jun 1992), 142–170. https://doi.org/10.1016/0890-5401(92)90017-A

[28] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation - OSDI'08*. USENIX Association, 209–224. http://dl.acm.org/citation.cfm?id=1855741.1855756

[29] Capstone. [n.d.]. Multi-architecture disassembly framework. https://github.com/aquynh/capstone

[30] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security - CCS'10*. ACM Press, New York, New York, USA, 559–572. https://doi.org/10.1145/1866307.1866370

[31] C. Cifuentes and M. Van Emmerik. 1999. Recovery of jump table case statements from binary code. In *Proceedings of the 7th International Workshop on Program Comprehension*. IEEE Comput. Soc, 192–199. https://doi.org/10.1109/WPC.1999.777758

[32] Edmund M. Clarke and E Allen Emerson. 1982. Design and synthesis of synchronization skeletons using branching time temporal logic. *Logics of Programs* 131 (1982), 52–71. https://doi.org/10.1007/BFb0025774

[33] Austin Clements. [n.d.]. Libelfin: Elf Parsing library. https://github.com/aclements/libelfin

[34] Lucian Cojocar, Taddeus Kroes, and Herbert Bos. 2017. JTR: A Binary Solution for Switch-Case Recovery. In *International Symposium on Engineering Secure Software and Systems - ESSoS'17*. Springer, Cham, 177–195. https://doi.org/10.1007/978-3-319-62105-0_12

[35] Marco Cova, Viktoria Felmetsger, Greg Banks, and Giovanni Vigna. 2006. Static Detection of Vulnerabilities in x86 Executables. In *Proceeding of 22nd Annual Computer Security Applications Conference - ACSAC'06*. IEEE, 269–278. https://doi.org/10.1109/ACSAC.2006.50

[36] Ward Cunningham. 1992. The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger* 4, 2 (apr 1992), 29–30. https://doi.org/10.1145/157710.157715

[37] David Currie, Xiushan Feng, Masahiro Fujita, Alan J. Hu, Mark Kwan, and Sreeranga Rajan. 2006. Embedded Software Verification Using Symbolic Execution and Uninterpreted Functions. *International Journal of Parallel Programming* 34, 1 (feb 2006), 61–91. https://doi.org/10.1007/s10766-005-0004-8

[38] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. 1979. Social processes and proofs of theorems and programs. *Commun. ACM* 22, 5 (may 1979), 271–280. https://doi.org/10.1145/359104.359106

[39] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C R Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[40] Detours. [n.d.]. a software package for monitoring and instrumenting API calls on Windows. https://github.com/microsoft/Detours

[41] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction - CC'17*. ACM Press, New York, New York, USA, 131–141. https://doi.org/10.1145/3033019.3033028

[42] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization. In *IEEE International Symposium on Security and Privacy - S&P'20*.

[43] DO178C. [n.d.]. Software Considerations in Airborne Systems and Equipment Certification. https://www.rtca.org

[44] Dyninst. [n.d.]. Tools for binary instrumentation, analysis, and modification. https://github.com/dyninst/dyninst

[45] Sebastian Elbaum and David S. Rosenblum. 2014. Known unknowns: testing in the presence of uncertainty. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE'14*. ACM Press, New York, New York, USA, 833–836. https://doi.org/10.1145/2635868.2666608

[46] Thomas G. Evans and D. Lucille Darley. 1966. On-line debugging techniques. In *Proceedings of the November 7-10, 1966, fall joint computer conference - AFIPS '66*. ACM Press, New York, New York, USA, 37. https://doi.org/10.1145/1464291.1464295

[47] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *IEEE Symposium on Security and Privacy - S&P'18*. IEEE, 679–696. https://doi.org/10.1109/SP.2018.00040

[48] David Garlan and David. 2010. Software engineering in an uncertain world. In *Proceedings of the FSE/SDP workshop on Future of software engineering research - FoSER '10*. ACM Press, New York, New York, USA, 125. https://doi.org/10.1145/1882362.1882389

[49] Loukas Georgiadis. 2005. *Linear-Time Algorithms for Dominators and Related Problems*. Ph.D. Dissertation. Princeton University. https://www.cs.princeton.edu/research/techreps/TR-737-05

[50] Stanley Gill. 1951. The diagnosis of mistakes in programmes on the EDSAC. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences* 206, 1087 (may 1951), 538–554. https://doi.org/10.1098/rspa.1951.0087

[51] Patrice Godefroid. 2014. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering - ICSE'14*. ACM Press, New York, New York, USA, 539–549. https://doi.org/10.1145/2568225.2568273

[52] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2008. Automated Whitebox Fuzz Testing. In *Network and Distributed System Security Symposium - NDSS'08*. https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[53] Rahul Gopinath, Carlos Jensen, and Alex Groce. 2014. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. ACM Press, New York, New York, USA, 72–82. https://doi.org/10.1145/2568225.2568278

[54] Group. [n.d.]. *2020 Linux Kernel History Report*. Technical Report. The Linux Foundation. https://www.linuxfoundation.org/blog/2020/08/download-the-2020-linux-kernel-history-report/

[55] Group. [n.d.]. Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi/

[56] Group. [n.d.]. Itanium C++ ABI: Exception Table Format. https://itanium-cxx-abi.github.io/cxx-abi/exceptions.pdf

[57] Group. [n.d.]. Persistent Memory Programming Kit. https://pmem.io/

[58] Group. 1997. System V Application Binary Interface. http://www.sco.com/developers/gabi/latest/contents.html

[59] Muhammad Ali Gulzar, Yongkang Zhu, and Xiaofeng Han. 2019. Perception and Practices of Differential Testing. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice - ICSE-SEIP*. IEEE, 71–80. https://doi.org/10.1109/ICSE-SEIP.2019.00016

[60] Maurice H. Halstead. 1977. *Elements of Software Science*. Elsevier Science Inc., USA. 128 pages.

[61] Mark Harman and Peter O'Hearn. 2018. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 1–23. https://doi.org/10.1109/SCAM.2018.00009

[62] Laune C. Harris and Barton P. Miller. 2005. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News* 33, 5 (dec 2005), 63–68. https://doi.org/10.1145/1127577.1127590

[63] John Harrison. 1999. A Machine-Checked Theory of Floating Point Arithmetic. In *Theorem Proving in Higher Order Logics*, Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 113–130.

[64] Hex-Rays. [n.d.]. IDA Pro: the interactive disassembler. https://www.hex-rays.com

[65] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235.363259

[66] R. N. Horspool and N. Marovac. 1980. An approach to the problem of detranslation of computer programs. *Computer Journal* 23, 3 (aug 1980), 223–229. https://doi.org/10.1093/comjnl/23.3.223

[67] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering - ICSE'14*. 435–445. https://doi.org/10.1145/2568225.2568271

[68] ISO-26262. [n.d.]. Road vehicles – Functional safety - Part 6: Product development at the software level. https://www.iso.org/obp/ui/{#}iso:std:iso:26262:-1:ed-2:v1:en

[69] Marko Ivankovic, Goran Petrovic, René Just, and Gordon Fraser. 2019. Code coverage at Google. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE'19*. 955–963.

[70] Jawahar Jain, Rajarshi Mukherjeey, and Masahiro Fujita. 1995. Advanced Verification Techniques Based on Learning. In *32nd Design Automation Conference - DAC'95*. ACM, 420–426. https://doi.org/10.1109/DAC.1995.249984

[71] Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *Comput. Surveys* 41, 4 (oct 2009), 1–54. https://doi.org/10.1145/1592434.1592438

[72] Linux Kernel. [n.d.]. Intel Processor Trace Documentation. https://github.com/torvalds/linux/blob/master/tools/perf/Documentation/perf-intel-pt.txt

[73] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungll Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing intermediate representations for binary analysis. In *32nd IEEE/ACM International Conference on Automated Software Engineering - ASE'17*. IEEE, 353–364. https://doi.org/10.1109/ASE.2017.8115648

[74] Johannes Kinder and Helmut Veith. 2008. Jakstab: A Static Analysis Platform for Binaries. In *Computer Aided Verification - CAV'08*. Springer Berlin Heidelberg, Berlin, Heidelberg, 423–427. https://doi.org/10.1007/978-3-540-70545-1_40

[75] Johannes Kinder, Florian Zuleger, and Helmut Veith. 2009. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *Verification, Model Checking, and Abstract Interpretation*, Neil D. Jones and Markus Müller-Olm (Eds.). Lecture Notes in Computer Science, Vol. 5403. Springer Berlin Heidelberg, 214–228. https://doi.org/10.1007/978-3-540-93900-9

[76] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (jul 1976), 385–394. https://doi.org/10.1145/360248.360252

[77] Gerwin Klein, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, and Rafal Kolanski. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. ACM Press, New York, New York, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[78] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy - SP'19*. IEEE, San Francisco, CA, USA, 1–19. https://doi.org/10.1109/SP.2019.00002

[79] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static disassembly of obfuscated binaries. In *Proceedings of 13th conference on USENIX Security Symposium*. USENIX Association, 255–270. http://dl.acm.org/citation.cfm?id=1251375.1251393

[80] W. Kunz and D.K. Pradhan. 1994. Recursive learning: a new implication technique for efficient solutions to CAD problems-test, verification, and optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13, 9 (1994), 1143–1158. https://doi.org/10.1109/43.310903

[81] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems* 1, 4 (dec 1992), 323–337. https://doi.org/10.1145/161494.161501

[82] Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. 2010. PEBIL: Efficient static binary instrumentation for Linux. In *IEEE International Symposium on Performance Analysis of Systems & Software - ISPASS'10*. IEEE, 175–183. https://doi.org/10.1109/ISPASS.2010.5452024

[83] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th Conference on Programming Languages Design and Implementation - PLDI'14*. ACM, 216–226. https://doi.org/10.1145/2666356.2594334

[84] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. 2017. Steelix: program-state based binary fuzzing. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ACM Press, New York, New York, USA, 627–637. https://doi.org/10.1145/3106237.3106295

[85] LibFuzzer. [n.d.]. a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html

[86] Cullen Linn and Saumya Debray. 2003. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of 10th Conference on Computer and Communication Security (CCS '03)*. ACM Press, New York, New York, USA, 290. https://doi.org/10.1145/948109.948149

[87] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium*. USENIX Association, Baltimore, MD, 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[88] M. Marre and A. Bertolino. 2003. Using spanning sets for coverage testing. *IEEE Transactions on Software Engineering* 29, 11 (nov 2003), 974–984. https://doi.org/10.1109/TSE.2003.1245299

[89] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering - TSE'76* SE-2, 4 (dec 1976), 308–320. https://doi.org/10.1109/TSE.1976.233837

[90] William M McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[91] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-scale continuous testing. In *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track - ICSE-SEIP'17*. IEEE, 233–242. https://doi.org/10.1109/ICSE-SEIP.2017.16

[92] Xiaozhu Meng and Barton P. Miller. 2016. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA'16*. ACM Press, New York, New York, USA, 24–35. https://doi.org/10.1145/2931037.2931047

[93] Andreas Moser, Christopher Kruegel, and Engin Kirda. 2007. Limits of Static Analysis for Malware Detection. In *Proceedings of 23rd Annual Computer Security Applications Conference (ACSAC '07)*. IEEE, Miami Beach, FL, USA, 421–430. https://doi.org/10.1109/ACSAC.2007.21

[94] Stefan Nagy and Matthew Hicks. 2019. Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing. In *IEEE Symposium on Security and Privacy - S&P'19*. IEEE, 787–802. https://doi.org/10.1109/SP.2019.00069

[95] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Commun. ACM* 58, 4 (mar 2015), 66–73. https://doi.org/10.1145/2699417

[96] Trail of Bits. [n.d.]. McSema: Framework for lifting x86, amd64, and aarch64 program binaries to LLVM bitcode. https://github.com/lifting-bits/mcsema

[97] Alessandro Orso and Gregg Rothermel. 2014. Software testing: a research travelogue (2000–2014). In *Proceedings of the on Future of Software Engineering - FOSE'14*. ACM Press, New York, New York, USA, 117–132. https://doi.org/10.1145/2593882.2593885

[98] OSS-Fuzz. [n.d.]. continuous fuzzing of open source software. https://github.com/google/oss-fuzz

[99] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Press, 2–14.

[100] Spencer Pearson, Jose Campos, Rene Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *IEEE/ACM 39th International Conference on Software Engineering - ICSE'17*. IEEE, 609–620. https://doi.org/10.1109/ICSE.2017.62

[101] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru R Caciulescu, and Abhik Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2941681

[102] Pin. [n.d.]. A Dynamic Binary Instrumentation Tool. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[103] Brian Potchik. [n.d.]. Architecture Agnostic Function Detection In Binaries. https://binary.ninja/2017/11/06/architecture-agnostic-function-detection-in-binaries.html

[104] Rui Qiao and R. Sekar. 2017. Function Interface Analysis: A Principled Approach for Function Recognition in COTS Binaries. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 201–212. https://doi.org/10.1109/DSN.2017.29

[105] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *Network and Distributed System Security Symposium - NDSS'17*. https://www.vusec.net/download/?t=papers/vuzzer{_}ndss17.pdf

[106] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. [n.d.]. Learning and Classification of Malware Behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment - DIMVA'08*. Springer Berlin Heidelberg, Berlin, Heidelberg, 108–125. https://doi.org/10.1007/978-3-540-70542-0_6

[107] Thomas Romain. [n.d.]. LIEF - Library to Instrument Executable Formats. https://github.com/lief-project/LIEF

[108] Nathan Rosenblum, Xiaojin Zhu, Barton Miller, and Karen Hunt. 2008. Learning to analyze binary computer code. In *Proceedings of the 23rd national conference on Artificial intelligence - AAAI'08*. AAAI Press, Chicago, Illinois, USA, 798–804. http://dl.acm.org/citation.cfm?id=1620163.1620196

[109] SanitizerCoverage. [n.d.]. LLVM coverage instrumentation. https://clang.llvm.org/docs/SanitizerCoverage.html

[110] Bastian Schlich. 2010. Model checking of software for microcontrollers. *ACM Transactions on Embedded Computing Systems* 9, 4 (mar 2010), 1–27. https://doi.org/10.1145/1721695.1721702

[111] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: hardware-assisted feedback fuzzing for OS kernels. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, 167–182. https://dl.acm.org/citation.cfm?id=3241204

[112] Hovav Shacham. 2007. 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*. ACM Press, New York, New York, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[113] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 611–626. https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin

[114] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy - S&P'16*. IEEE, 138–157. https://doi.org/10.1109/SP.2016.17

[115] Robert Swiecki. [n.d.]. Honggfuzz: a security oriented fuzzer. https://github.com/google/honggfuzz

[116] Syscall-intercept. [n.d.]. A system call intercepting library. https://github.com/pmem/syscall{_}intercept

[117] Aditya Thakur, Junghee Lim, Akash Lal, Amanda Burton, Evan Driscoll, Matt Elder, Tycho Andersen, and Thomas Reps. 2010. Directed proof generation for machine code. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Lecture Notes in Computer Science, Vol. 6174. Springer Berlin Heidelberg, Berlin, Heidelberg, 288–305. https://doi.org/10.1007/978-3-642-14295-6

[118] Henrik Theiling. 2000. Extracting safe and precise control flow from binaries. In *Proceedings 7'th International Conference on Real-Time Computing Systems and Applications ((RTCSA'00).* IEEE Computer Society, 23–30. https://doi.org/10.1109/RTCSA.2000.896367

[119] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2002. Efficient instrumentation for code coverage testing. In *Proceedings of the international symposium on Software testing and analysis - ISSTA '02*, Vol. 27. ACM Press, New York, New York, USA, 86. https://doi.org/10.1145/566172.566186

[120] Paul Turner. [n.d.]. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886

[121] Unicorn. [n.d.]. CPU emulator framework. https://github.com/unicorn-engine/unicorn

[122] Carlos Villarraga, Bernard Schmidt, Joerg Bormann, Christian Bartsch, Dominik Stoffel, and Wolfgang Kunz. 2013. An equivalence checker for hardware-dependent embedded system software. In *Proceddings of 11th IEEE/ACM International Conference on Formal Methods and Models for Codesign - MEMOCODE'13.* 119–128.

[123] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-Aware Greybox Fuzzing. In *IEEE/ACM 41st International Conference on Software Engineering - ICSE'19.* IEEE, 724–735. https://doi.org/10.1109/ICSE.2019.00081

[124] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable disassembling. *Proceedings of the 24th USENIX Security Symposium* (2015), 627–642. https://www.usenix.org/node/190921

[125] Richard Wartell, Yan Zhou, Kevin W. Hamlen, and Murat Kantarcioglu. 2014. Shingled Graph Disassembly: Finding the Undecideable Path. In *Proceedings of 18th Pacific-Asia Conference on Knowledge Discovery and Data Mining - PAKDD'14.* 273–285. https://doi.org/10.1007/978-3-319-06608-0_23

[126] Reinhard Wilhelm, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, Per Stenström, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, and Reinhold Heckmann. 2008. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems* 7, 3 (apr 2008), 1–53. https://doi.org/10.1145/1347375.1347389

[127] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'20.* ACM, New York, NY, USA, 133–147. https://doi.org/10.1145/3373376.3378470

[128] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security - CCS'13.* ACM Press, New York, New York, USA, 511–522. https://doi.org/10.1145/2508859.2516736

[129] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd conference on Programming language design and implementation - PLDI'11*. ACM, 283–294. https://doi.org/10.1145/1993316.1993532

[130] S. Yoo and M. Harman. 2010. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (mar 2010), n/a–n/a. https://doi.org/10.1002/stvr.430

[131] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium*. 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun

[132] Michal Zalewski. [n.d.]. Technical whitepaper for afl-fuzz. http://lcamtuf.coredump.cx/afl/technical{_}details.txt

[133] Mingwei Zhang and R. Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium*. USENIX Association, 337–352. http://dl.acm.org/citation.cfm?id=2534766.2534796

[134] Michael Zhivich and Robert K. Cunningham. 2009. The Real Cost of Software Errors. *IEEE Security & Privacy Magazine* 7, 2 (mar 2009), 87–90. https://doi.org/10.1109/MSP.2009.56

Lebenslauf

Name:	Ben Khadra, Mohamed Ammar
Geburtsort:	Damaskus, Syrien

Ausbildung

09/2001 - 08/2006	:	Bachelor Informatik Damaskus Universität, Syrien
10/2011 - 10/2013	:	Master Eingebettete Systeme (Erasmus Mundus) University of Southampton, Vereinigte Königreich und Technische Universität Kaiserslautern, Deutschland

Berufstätigkeit

12/2006 - 09/2007	:	Softwareentwickler Al-Khawarezmi Computer Center, Syrien
10/2007 - 09/2008	:	Softwareentwickler NeoTech Solutions, Syrien
10/2008 - 08/2011	:	Systemingenieur iNET ISP, Syrien
02/2014 - 08/2014	:	Stipendium der TU Kaiserslautern für Promotionstudium
09/2014 - 01/2021	:	Wissenschaftlicher Mitarbeiter am Lehrstuhl Entwurf Informationstechnischer Systeme Technischen Universität Kaiserslautern