AUSTRALIAN NATIONAL UNIVERSITY

DOCTORAL THESIS

State Space Search in Fuzzing

Author:
Adrian HERRERA

Panel:
Dr. Antony L. HOSKING,
Dr. Mathias PAYER,
Dr. Michael NORRISH

A thesis submitted for the degree of Doctor of Philosophy
of The Australian National University

November 17, 2024

Declaration of Authorship

I, Adrian HERRERA, declare that this thesis titled, "State Space Search in Fuzzing" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: Adrian Herrera

Date: 19/11/2024

"Space is big. You just won't believe how vastly, hugely, mind-boggling big it is."

Douglas ADAMS

Acknowledgements

I am eternally indebted to my panel: Tony Hosking, Mathias Payer, and Michael Norrish. Each of them brought their own special *something* to my PhD journey. Tony's boundless optimism kept me going through all of those paper rejections. Mathias' technical chops are unparalleled, and he always provided insightful comments and feedback on my work. I am forever grateful for his unwavering dedication and support for a student on the other side of the world. Michael unquestionably made me a better writer. Those who say "*you can't polish a turd*" have not witnessed Michael's formidable wordsmithing abilities.

I am grateful to my friends and former colleagues at the *Defence Science and Technology Group* (DSTG) for providing a workplace where undertaking further study is not only supported, but actively encouraged. In particular: Shane Magrath, for encouraging me to do this in the first place; Jon Arnold, Nick McConnell, Adrian Caldow, Julian Costa, and Gareth Parker, for supporting me while I context-switched between managing and PhD-ing; Ben Cheney, Danny Coscia, Josh Green, and Suneel Randhawa, for getting me into cyber security research; Jamie Harwood, David Jamieson, Vi Li, Chenni Shettigara, and Nicole Stephens, for allowing us DSTG interlopers to work in the *Australian Signal's Directorate*'s (ASD) CoLab and being so welcoming; and finally the *Software Systems Security* (S3) team: Luke Croak, Lauren Nelson-Lee, James Shaker, and Melissa Turner, for keeping me sane (primarily by distracting me with management-y things).

Every morning—modulo a global pandemic—I would eagerly join my *Australian National University* (ANU) colleagues on campus at the *Little Pickle* café. In particular, I thank Steve Blackburn, Thomas Haines, and Peter Höfner for the chats about academia, research, and life. Thanks also to Luke for keeping me caffeinated ☕.

In addition to drinking coffee at the Little Pickle, I had the pleasure of working with several bright and inspiring ANU staff and students. I thank: Zixian Cai, for not only maintaining the Moma computing cluster and *not* crashing my car when learning to drive, but for being an impressive and inspiring young systems researcher; Joshua Corner, Kathryn Gill, and Amy Samson, for being smart, dedicated, and—above all—*fun* students; Hendra Gunadi, for his early experimental work on the *MoonLight* project, paving the way for the "Seed Selection for Successful Fuzzing" paper; and finally, Alwen Tiu, for trusting me to help develop and teach COMP2710 *Software Security* (and then asking me to return—multiple times—when it was rebadged as COMP3703).

Outside of ANU and DSTG, I had the opportunity to meet and work with some

super-smart and cool people. Arlen Cox and Andrew Ruef were always up for a chat, providing feedback on my papers, ideas, and sharing their (impressive) knowledge on all things program analysis and Datalog. I look forward to future discussions on the latest-and-greatest program analysis research.

I was fortunate to be on the organizing committee for the 9th *International Summer School on Information Security and Protection* (ISSISP) in Canberra. *Trail of Bits* generously sponsored this even and sent out Peter Goodman to give a workshop on binary lifting. Peter and I have kept in regular contact since; hacking on PASTA, lifting binaries to LLVM, and talking all things program analysis. Peter was also generous with his time and gave me invaluable feedback on my work. I hope to keep these conversations going well into the future.

Despite not physically being in Switzerland, I always felt like a member of the *HexHive* lab. In particular, the Slack workspace was a place of vibrant discussion, lively debate, and sharing of knowledge. I thank all of the HexHive crew; particularly Atri Bhattacharyya, Priyam Biswas, Manuel Egele, Ahmad Hazimeh, Yuseok Jeon, Zhiyuan Jiang, Hui Peng, Andrés Sanchez, Prashast Srivastava, Antony Vennard, and Chibin Zhang, for their passes on my papers and discussions on my research ideas.

Marcel Böhme greatly influences how I think about fuzzing. In particular, his statistical modeling of fuzzing has brought rigor to what I once considered a "wild-west" research field. The fuzzing research community is much healthier with people like Marcel.

As my PhD journey drew to a close, I started looking for the next adventure and joined the team at *Interrupt Labs*. Interrupt Labs have encouraged me to take the skills and techniques I developed during my PhD and apply them to "real-world" problems. I am very fortunate to work with a great team of world-class vulnerability researchers, and I thank James Loureiro, Mark Barnes, and Mat Ramsey for having me onboard.

I feel incredibly lucky to have worked with (and learned from) the team at *Infosect*, particularly Kylie McDevitt and Silvio Cesare. Not only are Kylie and Silvio first-class hackers, but they are also incredibly kind and supportive people. I thank them for allowing me to speak at their conferences and inviting me to their events (in particular, their excellent Christmas parties). I plan to attend many more of their conferences!

I thank my Mum, Julie Herrera; Dad, Carlos Herrera; Nan, Nancy Gallagher (RIP); and brother, Daniel Herrera. Despite not really understanding the technical intricacies of fuzzing, they always supported what I was doing. They encouraged me to study

what I wanted to study (at both school and university), ultimately positioning me where I am today. Thank you.

I met my partner Jessica Hughes halfway through my PhD. She made the second half of this PhD so much easier than it should have been; supplying me with not only coffee, icecream, and trashy reality TV shows, but unflinching support, time, and space for me to do my research. Meeting Jess also meant becoming a "bonus dad" to her two girls, Fleur and Willa. Fleur and Willa also provided a welcome distraction from the depths of this dissertation, albeit a very different kind of distraction! I love you all ♡.

AUSTRALIAN NATIONAL UNIVERSITY

Abstract

Doctor of Philosophy

State Space Search in Fuzzing

by Adrian HERRERA

Coverage-guided fuzzers are an indispensable tool in the software-testing toolbox. They uncover bugs in a target program by subjecting it to a large number of automatically-generated inputs. The fuzzer generates these inputs to search the corners of the target's (potentially vast) *state space*, where bugs are more likely to lurk. While fuzzers have successfully uncovered bugs in a range of targets, they struggle to discover "deep bugs" (i.e., bugs triggered under a complex set of control and data dependencies). Moreover, security professionals deploying fuzzers lack observability into fuzzers' state space search (and thus an understanding of *why* these deep bugs are missed).

This dissertation presents a set of techniques for reasoning about and improving a fuzzer's state space search, ultimately enhancing a fuzzer's bug-finding ability.

First, we consider the task of bootstrapping this search process. Fuzzers typically require an initial set of *seeds* (exemplar inputs accepted by the target) to kickstart their state space search. We empirically evaluate various methods for selecting these seeds, designing an optimal technique for reducing large seed sets in the process.

Second, we develop a new state space abstraction. Fuzzers traditionally abstract a target's state space based on *control-flow* features. We present an abstraction based on *data-flow* features and demonstrate how our data-flow-based abstraction uncovers bugs that traditional fuzzers fail to find.

Finally, we investigate how best to measure a fuzzer's state space search after a fuzzing campaign. We use static analysis to quantify a target's state space, allowing us to measure *how much* of this state space a fuzzer has explored. We empirically evaluate several modern static analysis frameworks and propose new approaches for assessing a fuzzer's state space search.

Contents

List of Figures

List of Tables

List of Acronyms

ASAN AddressSanitizer.

UBSAN UndefinedBehaviorSanitizer.

AFL American Fuzzy Lop.

BFF Basic Fuzzing Framework.

CFG control-flow graph.

CGC Cyber Grand Challenge.

CI confidence interval.

CNF conjunctive normal form.

CVE Common Vulnerabilities and Exposures.

DDG data dependency graph.

DTA dynamic taint analysis.

FTS Fuzzer Test Suite.

ICFG interprocedural control-flow graph.

ILP integer linear programming.

IR intermediate representation.

JIT just in time.

LoC lines of code.

LTO link-time optimization.

MaxSAT maximum satisfiability.

NSS Network Security Services.

OS operating system.

PAMD Padding Area MetaData.

PCA Principal Component Analysis.

PCSG probabilistic context-sensitive grammar.

RMST restricted mean survival time.

SAT boolean satisfiability.

SDLC software development lifecycle.

STADS software testing and analysis as discovery of species.

t-SNE t-distributed stochastic neighbor embedding.

To Jess, Fleur, and Willa.

Chapter 1

Introduction

Modern software systems are growing exponentially larger and more complex. At the same time, society is becoming more dependent on these software systems. For example, the Linux kernel—which powers mobile phones, routers, servers, and more—has grown from 13 million lines of code (LoC) in 2016 to over 20 MLoC in 2021 (a 54 % increase in five years). This increased size and complexity results in a greater likelihood of bugs and security vulnerabilities simply through the added code. Moreover, efficiently analyzing these codebases to find bugs and security vulnerabilities is becoming increasingly difficult (e.g., due to their large size; complex interactions between multiple, heterogeneous components).

An abundance of automatic bug-finding techniques have been developed in an attempt to keep up with the growing size and complexity of modern codebases. These techniques are either *static* or *dynamic*; the former analyzes code without executing it, while the latter observes executed code. Static analyses can reason over *all* possible behaviors that may arise at run time, whereas dynamic analyses can only reason about behaviors observed over a (finite) set of inputs. However, Rice's theorem [180] means that non-trivial semantic questions a static bug finder would like answered (e.g., "*is an arbitrary program free of security vulnerabilities?*") are undecidable. This forces static analyses to be conservative, leading to false positives (i.e., claiming a bug exists when it does not) and false negatives (i.e., misdetecting a bug). Unfortunately, static analyses have garnered a reputation for being more harmful than helpful [45, 96, 109] (e.g., due to the prohibitively high number of false positives/negatives some static bug finders emit) and difficult to scale [54, 182]. This has led to a renewed interest in dynamic analyses, with a particular focus on *fuzz testing*.

Fuzz testing ("fuzzing") has become the de facto dynamic analysis for discovering bugs and vulnerabilities in large, complex codebases. In contrast to other bug-finding and verification techniques—e.g., model checking [106], symbolic execution [14], and static analysis [11]—fuzzing is (a) highly scalable, (b) free of false positives, and

(c) unencumbered by models of the external environment. These advantages have led to the widespread adoption of fuzzing, and the subsequent discovery of tens of thousands of bugs in popular programs [3, 31, 33, 52, 181]. For example, Google's OSS-Fuzz service has helped fix over 8,800 vulnerabilities and 28,000 bugs across 850 open-source projects [31].

A fuzzer finds bugs by executing the target codebase (the "target") multiple times with a large number of automatically generated (and possibly malformed) inputs. These inputs are generated to maximize exploration of the target's *state space*. A program *state* is a snapshot of memory and registers, as well as relevant aspects of the operating system (OS) affecting the program (e.g., file descriptors, network sockets) [7, 183]. The target's state space is thus the set of all possible states a program can be in, and the target's behaviors are described by paths through this state space. Intuitively, exploration of the target's state space correlates with bug discovery [23]; after all, you can only trigger a bug in code executed.

Early fuzzers treated the target as a black box. In particular, these fuzzers had no internal view of the target's state space; their only feedback was whether the target crashed when executing a given input. In contrast, modern fuzzers typically use lightweight instrumentation to track which parts of the target's state space they have explored. This information is fed back to the fuzzer, helping to guide its state space search.

Unfortunately, working with the previous definition of a target's state space is unwieldy and quickly makes any program analysis (static or dynamic) intractable. In particular, it is impractical for a fuzzer to measure a state with this level of detail; prior work has shown the importance of maintaining a fuzzer's *iteration rate* (the number of inputs the target executes per unit of time), and this level of detail will significantly reduce this iteration rate, harming fuzzing outcomes.

Abstraction is a solution to this problem, improving an analysis' tractability [113]. In fuzzing and symbolic execution, *control* abstraction is common, while *data* abstraction is customary in abstract interpretation and model checking. In the former, a target program's state space is described by the set of computation sequences and control flow; e.g., basic blocks/edges in the target's control-flow graph (CFG) [8, 9, 16, 19, 21, 36, 37, 39, 40, 51, 61, 68, 142, 167]. In the latter, the target's state space is described by interpretations (i.e., valuations) of variables [48, 56, 106]. These abstractions provide complementary views of the target's state space.

Despite these abstractions, even trivially small programs can have innumerably

large state spaces.[1] This is particularly problematic for coverage-guided greybox fuzzers—the focus of this dissertation—which randomly walk the target's state space.[2] Notably, the fuzzer must satisfy increasingly complex control- and data-flow-based dependencies to explore every corner of the target's state space. Unfortunately, despite improvements in fuzzing techniques, fuzzers still have difficulty solving complex control- and data-flow constraints, leading to "deep" bugs continuing to evade long-running fuzzing campaigns. Moreover, security professionals deploying fuzzers lack *observability* into the fuzzer's state space search, making it difficult to understand *why* these bugs remain undiscovered.

1.1 Thesis

This dissertation describes techniques, frameworks, and methodologies for measuring and improving a fuzzer's ability to explore a target program's state space. My thesis is:

Thesis statement

Fuzzing outcomes (i.e., bug discovery) are enhanced by improving fuzzers' state space search.

We present a high-level overview of a typical fuzzing campaign in Fig. 1.1. Given the target program to fuzz, a set of well-formed, exemplar inputs (of the format accepted by the target) are sourced (①). As previously described, modern fuzzers instrument the target to track state space exploration (②). Following this, the fuzzer is run (③), mutating inputs to explore the target's state space.[3] Finally, the results are analyzed after completing the campaign (④). This analysis includes triaging crashes (to determine a bug's root cause) and examining which parts of the state space were explored.

We can improve a fuzzer's state space search in several ways. First, rather than starting the search "from scratch", we can cover a subset of the search space before beginning a fuzzing campaign (i.e., at ①). Second, we can avoid revisiting previously-visited states, giving the fuzzer more time to discover new states (during ③). Third, we can adopt different views of a target's state space (at ②), revealing states that may not be explicit in conventional abstractions. Finally, we can use more powerful

[1] I.e., the state space cannot be enumerated before the heat death of the universe.

[2] We leave details on how coverage-guided fuzzers operate to Chapter 2.

[3] We use the rabbit icon as a homage to the American Fuzzy Lop (AFL) fuzzer, named after the rabbit breed.

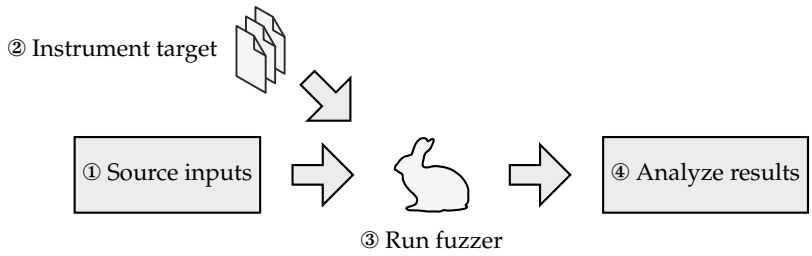


FIGURE 1.1: High-level overview of a fuzzing campaign.

techniques to gain deeper insight into a fuzzer's state space search (at ④). This dissertation explores all of these improvements. Before outlining our specific contributions (Section 1.3), we provide several examples to motivate our work.

1.2 Motivating Examples

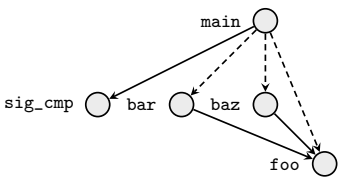
CVE-2021-43527 [158] is a heap buffer overflow in Mozilla's Network Security Services (NSS) library that occurs when handling particular cryptographic signatures. While the bug itself is unremarkable, the reasons why it remained undiscovered in widely-fuzzed code for almost ten years [162] makes it a motivating example for this dissertation.

How did this bug remain undiscovered for so long? One reason was "*misleading [coverage] metrics*" [162]. A coverage metric guides a fuzzer's state space search by abstracting the target's state space (we present a more thorough discussion of fuzzer coverage metrics in Chapter 3). Moreover, fuzzers are commonly evaluated and compared by how much of the target's state space they explore (under a given coverage metric). Despite "*good test coverage for the vulnerable areas*" [162], the bug was only found when fuzzing with an "unconventional" coverage metric (specifically, context-sensitive edge coverage, discussed further in Section 3.2.4).

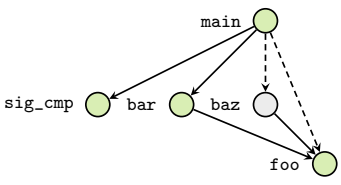
Similarly, we use the program in Fig. 1.2 (adapted from Ormandy [162]) to demonstrate the need to rethink fuzzers' state space search. The bug at line 3 (in Fig. 1.2a) is only triggered when `foo` is called directly from `main` (i.e., when `data[i] == 0x66` at line 34); `buf` cannot overflow if `bar` or `baz` are called (lines 4 and 5, respectively). However, this is problematic for fuzzers driven by control-flow coverage alone. Consider a function call graph abstraction of the target's state space (Fig. 1.2b) and two inputs (byte arrays represented as integer sequences, read from the file descriptor `fd` at line 27): $\iota_1 = \{\texttt{0xa}, \texttt{0xb}, \texttt{0xc}, \texttt{0x61}, \texttt{0x40}, \texttt{0x40}\}$ and $\iota_2 = \{\texttt{0xa}, \texttt{0xb}, \texttt{0xc}, \texttt{0x66}, \texttt{0x40}, \texttt{0x40}\}$. Under this abstraction, ι_1's coverage is a superset

```c
static char buf[128];

void foo(int a, size_t b) {  memset(buf, a, b); }
void bar(int a, size_t b) {  foo('A', sizeof(buf)); }
void baz(int a, size_t b) {  bar(a, sizeof(buf)); }

const struct handler_t {
  char code;
  void (*handler)(int, size_t);
} handlers[] = {{0x66, foo}, {0x61, bar}, {0x7a, baz}};

int sig_cmp(void *data, size_t start, size_t len) {
  if (start + 3 > len)
    return 1;

  static const char sig[3] = {0xa, 0xb, 0xc};
  return memcmp(&data[start], sig, 3);
}

int main(int argc, char *argv[]) {
  struct stat st;

  int fd = open(argv[1], O_RDONLY);
  fstat(fd, &st);
  size_t size = st.st_size;
  char *data = malloc(size);
  read(fd, data, size);

  if (sig_cmp(data, 0, size))
    return 1;

  for (unsigned i = 4; (i + 3) <= size; i += 3)
    for (unsigned j = 0; j < 3; ++j)
      if (data[i] == handlers[j].code)
        handlers[j].handler(data[i + 1], data[i + 2]);

  free(data);
  close(fd);

  return 0;
}
```

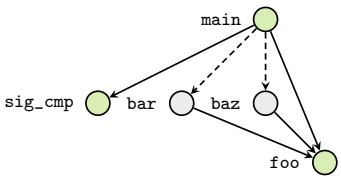
(A) Source.



(B) Call graph. The dashed lines indicate indirect calls that are resolved at run time.



(C) Call graph for the input $\iota_1 = \{0xa, 0xb, 0xc, 0x61, 0x40, 0x40\}$ (covered nodes are green).



(D) Call graph for the input $\iota_2 = \{0xa, 0xb, 0xc, 0x66, 0x40, 0x40\}$.

FIGURE 1.2: A simple dispatch table, adapted from Ormandy [162].

of ι_2's coverage (see Figs. 1.2c and 1.2d).[4] This means that even before fuzzing begins, *corpus minimization* (the process of minimizing the number of inputs that maximize coverage of the target's state space; discussed further in Chapter 4) would discard inputs covering only `foo` in favor of inputs covering `bar` and `baz` (because covering `bar` or `baz` also covers `foo`).

Observation

Conventional fuzzer coverage metrics (e.g., those based off of control flow) may lead to inaccurate and misleading measurements of program state.

We also use Fig. 1.2 to demonstrate the need to avoid starting a fuzzing campaign "from scratch". In particular, `sig_cmp` (line 12) rejects inputs not starting with the signature $\{$`0xa`, `0xb`, `0xc`$\}$. While this is a relatively simple check to overcome, one can imagine more intricate file formats with more complex checks (e.g., a JavaScript interpreter). In general, these checks are difficult for a fuzzer to "guess" (e.g., via random mutation). However, if the fuzzer is bootstrapped with knowledge of the input format (e.g., with exemplar inputs starting with the three-byte signature), then the fuzzer can spend more time exploring the handler (rather than wasting cycles overcoming the signature check).

Observation

A fuzzer's ability to reach "deep" states in a target may depend on how the fuzzer's search was bootstrapped.

These examples demonstrate the need to rethink and improve fuzzers' state space search. This includes appropriately bootstrapping the state space search, adopting coverage metrics that go beyond traditional control-flow-based metrics, and improving insights into what states the fuzzer has covered. The following section outlines how we achieve this.

1.3 Structure and Contributions

This dissertation describes how we reason about and improve the state space exploration capabilities of coverage-guided fuzzers. We start in Chapter 2 by providing the background material necessary to motivate and understand our contributions.

[4] The same is true for a CFG abstraction; we use call graphs to simplify presentation.

We also summarize the current state-of-the-art fuzzing research. The structure of the remaining chapters echoes a fuzzing campaign (Fig. 1.1). In these chapters we make the following contributions:

Systematization of coverage metrics. In Chapter 3 we explore the different ways a fuzzer represents a program's state space, presenting a systematization of coverage metrics and taxonomy of greybox fuzzers.

The impact of seed selection and optimal corpus minimization. In Chapter 4 we focus on the initial set of inputs required to bootstrap the fuzzing process (① in Fig. 1.1). We systematically (a) investigate how these initial input sets are constructed, and (b) evaluate how they affect a fuzzer's ability to find bugs. We also present OPTIMIN, a fuzzing corpus minimization tool. Unlike other corpus minimizers, OPTIMIN is capable of deriving an *optimal* corpus by encoding corpus minimization as a boolean satisfiability (SAT) problem.

Data-flow-guided fuzzing. In Chapter 5 we present DATAFLOW, a greybox fuzzer guided by lightweight data-flow profiling. Unlike most greybox fuzzers (i.e., those discussed in Chapter 3), DATAFLOW's instrumentation eschews control-flow coverage metrics in favor of *def-use chain* coverage (②).

Quantifying state space search. In Chapter 6 we look at measuring a fuzzer's state space search after a fuzzing campaign (④). We use static analysis to quantify a target's state space, aiming to produce a more-thorough analysis and evaluation of a fuzzer's ability to explore a target's state space.

Comparing coverage metrics. Finally, in Chapter 7 we conduct a large-scale fuzzing campaign to evaluate the full range of coverage metrics discussed throughout this dissertation.

We support our contributions with extensive empirical evaluation on real-world software. In total, we report on over 44 CPU-yr of fuzzing campaigns. Finally, we release as open source all code associated with our contributions (details are provided in each chapter).

1.3.1 Publications

While this dissertation is self-contained, its contributions draw from the following published material:

- Ahmad Hazimeh, **Adrian Herrera**, and Mathias Payer. *Magma: A Ground-Truth Fuzzing Benchmark* [84]. SIGMETRICS. 2020.
- **Adrian Herrera**, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. *Seed Selection for Successful Fuzzing* [92]. International Symposium on Software Testing and Analysis (ISSTA). 2021.
- **Adrian Herrera**, Mathias Payer, and Antony L. Hosking. *Registered Report:* DATAFLOW*– Toward a Data-Flow-Guided Fuzzer* [91]. International Fuzzing Workshop (FUZZING). 2022.
- Zhiyuan Jiang, Shuitao Gan, **Adrian Herrera**, Flavio Toffalini, Lucio Romerio, Chaojing Tang, Manuel Egele, Chao Zhang, and Mathias Payer. EVOCATIO*: Conjuring Bug Capabilities from a Single PoC* [107]. Computer and Communications Security (CCS). 2022.
- **Adrian Herrera**, Mathias Payer, and Antony L. Hosking. DATAFLOW*: Toward a Data-Flow-Guided Fuzzer* [90]. Transactions on Software Engineering and Methodology (TOSEM). 2023.
- Simon Luo, **Adrian Herrera**, Paul Quirk, Michael Chase, Damith C. Ranasinghe, Salil S. Kanhere. *Make out like a (Multi-Armed) Bandit: Improving the Odds of Fuzzer Seed Scheduling with T-Scheduler* [132]. Asia Computer and Communications Security (AsiaCCS). 2024.

Chapter 2

Background and Related Work

2.1 Introduction

The term "fuzz testing" first appeared in an assignment in Prof. Barton Miller's 1988 advanced operating system (OS) class[1] at the University of Wisconsin [146]. Since then, fuzz testing ("fuzzing") as both a research field and industrial practice has progressed in leaps and bounds. Nowadays, you can browse the proceedings of the top security and software engineering conferences and find numerous fuzzing papers, or read countless blog posts about how software companies are scaling-up fuzzing to find bugs in their code (before it reaches production). Here, we distill this prior work and describe both fuzzing fundamentals and the current state-of-the-art.

Chapter outline. This chapter provides the background material required to motivate and understand our research contributions. We begin by describing fuzzing and how it relates to other software testing techniques. This is followed by an outline of the different "shades" of fuzzer: black-, grey-, and white-box (Section 2.3) and a more in-depth discussion of *coverage-guided greybox fuzzing* (Section 2.4), the focus of this dissertation. We discuss related prior work in Section 2.5 and conclude with a discussion on the experimental methodologies we use to ensure statistically-sound empirical evaluation in Section 2.6.

2.2 Software Testing

Most software requires testing to ensure it operates as intended. A primary goal of testing is to find bugs *before* the software is released to consumers. Figure 2.1 summarizes different testing techniques. *Static* techniques include formal verification (mathematically proving the system is correct with respect to a formal specification),

[1] CS763, to be precise.

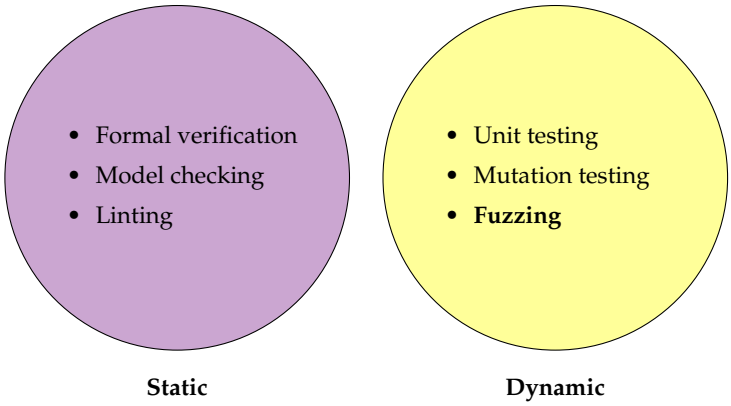


FIGURE 2.1: Examples of software testing techniques.

model checking (checking a finite-state model of the system meets a given specification), and linting (pattern matching against known bug classes and suspicious programming constructs). *Dynamic* techniques include unit tests (checking isolated code—typically at the function level—exhibits expected behavior), mutation testing (modifying the program—typically to cause an error—to ensure the test suite can detect the modifications), and fuzzing (the focus of this dissertation).

While it is often impossible to identify *all* bugs and errors in software before it is released, testing is an important component of the software development lifecycle (SDLC). In particular, testing reduces the risk of *security critical* bugs that can be exploited by attackers for nefarious purposes.

2.3 Fuzz Testing

Fuzzing is a dynamic software testing technique for finding bugs. While some forms of testing (e.g., unit testing) attempt to verify code behavior based on valid inputs ("positive" testing), fuzzing investigates code behavior on invalid or unexpected inputs ("negative" testing). Fuzzers do this by executing code with automatically-generated inputs and triggering crashes [138]. How a fuzzer produces these inputs depends on whether it is *generational* or *mutational*.

Generational fuzzers (e.g., QuickFuzz [77], NAUTILUS [8], Gramatron [199], and Favocado [53]) require a specification/model of the input format. They use this specification to synthesize inputs. In contrast, mutational fuzzers (e.g., libFuzzer [188], Angora [37], REDQUEEN [9] and INVSCOV [61]) use an initial corpus of seed inputs (e.g., files, network packets, and environment variables) to bootstrap the input-generation process. New inputs are then generated by mutating existing seeds in this corpus. Mutational fuzzers are more common than generational fuzzers, so when

discussing fuzzers we assume the mutational kind (rather than generational) unless otherwise stated.

In addition to how inputs are produced, fuzzers are further categorized by the "*granularity of semantics [i.e., behaviors] a fuzzer observes in each fuzz run*" [138]. These categories—black-, grey-, and white-box—are summarized below.

2.3.1 Blackbox Fuzzers

Blackbox fuzzers have no internal view of the target; they can only observe input/output relationships. Without any introspection, blackbox fuzzers are unencumbered by run-time overheads (e.g., associated with instrumentation) and thus achieve high *iteration rates* (i.e., the number of inputs executed per unit of time). However, they cannot measure/reason about how much/which parts of the target's state space have been explored. Example blackbox fuzzers include Radamsa [87], the CERT Basic Fuzzing Framework (BFF) [196], and CodeAlchemist [82].

2.3.2 Greybox Fuzzers

A greybox fuzzer uses lightweight instrumentation (typically compiler-based instrumentation that avoids the need for heavyweight program analysis at run time) to collect run-time information about the target. This run-time information is fed back to the fuzzer to guide it toward unexplored parts of the state space (and, ultimately, uncover bugs). Example greybox fuzzers include American Fuzzy Lop (AFL) [231], NAUTILUS [8], and `DDFuzz` [141]. We discuss greybox fuzzing in greater detail in Section 2.4.

2.3.3 Whitebox Fuzzers

Whitebox fuzzers rely on heavyweight program analysis to guide target exploration. This program analysis is typically provided by a symbolic execution engine ("symbolic executor") [14]. While capable of overcoming complex conditionals and other fuzzer roadblocks, symbolic executors incur a high run-time cost (e.g., the KLEE [27] and angr [194] symbolic executors are ∼3,000× and ∼321,000× slower than native execution, respectively [230]). Moreover, *path explosion*—where branches and symbolic memory accesses lead to a large number of program states that cannot be feasibly explored—makes systematically exploring a target's state space intractable.

These performance and scalability issues have led to the emergence of concolic (concrete + symbolic) execution engines ("concolic executor"). Rather than exploring the target's state space (and following multiple execution paths simultaneously), a

concolic executor traces the constraints for a single execution path and uses these constraints to generate new inputs that follow new paths (that deviate from the single execution path in a given branch). Example whitebox fuzzers include KLEE [27], *BuzzFuzz* [69], and SAGE [73]. Example concolic fuzzers include Driller [201], QSYM [230], and SYMCC [173].

2.4 Coverage-guided Greybox Fuzzing

Coverage-guided greybox fuzzers are the most pervasive type of fuzzer: they have successfully uncovered tens of thousands of bugs in a range of widely-used software [3, 33, 52, 181]. We describe their operation in greater detail in this section.

Algorithm 1: Mutational greybox fuzzing.

```
   Input: Target 𝒫, starting seeds 𝒮
   Output: Crashing inputs 𝒮✗
1  𝒮✗ ← ∅, 𝐂𝒮 ← ∅
2  if 𝒮 = ∅ then /* Use the empty seed if no seeds are provided */
3      𝒮 ← {EmptySeed}
4  ℛ ← Preprocess(𝒫)
5  foreach ι ∈ 𝒮 do /* Save starting seeds' coverage */
6      𝐂ι ← Execute(𝒫, ι)
7      𝐂𝒮 ← 𝐂𝒮 ∪ 𝐂ι
8  repeat/* The main fuzzing loop */
9      ι ← ChooseNext(𝒮, ℛ)
10     e ← AssignEnergy(ι, ℛ)
11     foreach i from 1 to e do
12         ι′ ← MutateInput(ι, ℛ)
13         𝐂ι′ ← Execute(𝒫, ι′)
14         if CrashDetected then
15             𝒮✗ ← 𝒮✗ ∪ {ι′}
16         else if IsInteresting(𝐂ι′, 𝐂𝒮, ℛ) then
17             𝒮 ← 𝒮 ∪ {ι′}
18         𝐂𝒮 ← 𝐂𝒮 ∪ 𝐂ι′
19 until timeout reached or abort signal
```

Algorithm 1—adapted from Böhme, Pham, and Roychoudhury [22] and Böhme et al. [24]—provides an overview of a generic coverage-guided greybox fuzzer. The user provides an instrumented target program to fuzz, $\mathcal{P}$, and an optional set of starting inputs ("seeds"), $\mathcal{S}$. An "empty seed" is used if $\mathcal{S}$ is not provided (lines 2 to 3). The target may first undergo a preprocessing/analysis stage (line 4) before the fuzzer enters the main fuzzing loop (lines 8 to 19), consisting of the following steps:

1. The fuzzer selects an input $\iota \in \mathcal{S}$ (line 9) and assigns it an *energy*, e (line 10). This energy "*specifies the number of inputs to be generated from that seed [i.e., input]*" [22].

The assignment of e depends on the fuzzer (e.g., AFL assigns energy based on the inputs execution time, coverage, and creation time).

2. The fuzzer mutates ι, producing ι' (line 12). These mutation operations can be *structure agnostic* or *structure aware*, depending on how much knowledge (if any) the fuzzer has of $\mathcal{P}$'s input format. Structure-agnostic mutations operate on the raw input string and include bit-flipping, byte substitution, and string concatenation. Structure-aware mutations are typically applied when $\mathcal{P}$'s input format is expressible by a *grammar*. In this case, mutations occur on an input's *parse tree*[2] and include node insertion/deletion, replacing nodes with those derived by a different set of grammar rules, and splicing parse trees together.

3. The target $\mathcal{P}$ is executed with ι' (line 13). During execution, the fuzzer records coverage information and stores this information in a coverage map $\mathbb{C}_{\iota'}$. This coverage approximates a program's state space and takes various forms (e.g., basic block, edges between basic blocks). For performance reasons, coverage is typically based on an approximation of *control-flow* information.

4. The fuzzer detects crashes and updates the global coverage map (lines 14 and 18). Crash-inducing inputs are saved for further (offline) triage and analysis. Further, an input is added to $\mathcal{S}$ if the fuzzer considers it "interesting"; i.e., it satisfies some objective function (e.g., generates new coverage, gets "closer" to a target location). Otherwise, the input is discarded (line 16). If interesting, the coverage obtained by ι' is added to the global coverage map $\mathbb{C}_{\mathcal{S}}$, which maintains coverage for *all* inputs.

5. Return to 1 (or terminate if the timeout is reached, line 19). The fuzzer may increase its aggressiveness (i.e., increase the energy e) on already-explored inputs.

The analysis report $\mathcal{R}$ may guide seed selection, energy assignment, mutation, and/or assist in determining if ι' is interesting (Steps 1, 2 and 4, respectively). For example, directed fuzzers (e.g., AFLGo [24], CAFL [121], BEACON [99]) precompute a function call graph to a guide the fuzzer toward specific target locations. An input is thus interesting if it reduces the "distance" to a target location.

The software testing and analysis as discovery of species (STADS) model [17] provides an alternative, probabilistic model of fuzzing. STADS is an ecology-inspired

[2] A parse tree—also known as a *derivation tree* or *concrete syntax tree*—represents the syntactic structure of ι according to a specific grammar.

statistical framework that enables accurate extrapolation of a fuzzer's future performance (based on past performance). This is important to fuzzing because it "*provides statistical correctness guarantees with quantifiable accuracy*" [17].

Under the STADS model, greybox fuzzing is considered a random walk through $\mathcal{P}$'s state space [22]. This walk is achieved by probabilistically sampling (with replacement) from $\mathcal{P}$'s input space, which is divided into subdomains called *species*. These species are defined based on behavior observed when executing $\mathcal{P}$ with input ι'. Observed behaviors are approximated by one or more *coverage metrics* and are recorded in $\mathbb{C}_{\mathcal{S}}$. Thus, improving the state space exploration capabilities of a greybox fuzzer is equivalent to improving its *ability to discover new species* under STADS.

2.5 Related Work

Significant attention—from both academia and industry—has been given to many of the elements and steps in Algorithm 1. We summarize this work in the following sections.

2.5.1 Benchmark Suites

Appropriate benchmarks—from which $\mathcal{P}$ is commonly sourced—are required for fair and meaningful fuzzer comparisons. Popular open-source software (e.g., GNU Binutils [72]) are frequently used to demonstrate a fuzzer's ability to discover *zero-day* vulnerabilities (i.e., bugs that have not been patched). However, determining the root cause of a "real-world" zero-day is a time-consuming and largely manual process, increasing the time and effort required to evaluate and compare multiple fuzzers. To rectify this, benchmarks such as LAVA [55], the Cyber Grand Challenge (CGC) [131], UNIFUZZ [124], Fuzzer Test Suite (FTS) [75], FUZZBENCH [144], Magma [84], and REVBUGBENCH (from FIXREVERTER [234]) have been developed. These benchmarks evaluate and compare fuzzers across a range of performance metrics, including: coverage profiles (FUZZBENCH); synthetic bug counts (LAVA, CGC, and REVBUGBENCH); and real-world bug counts (UNIFUZZ, FTS, and Magma). Is there a preferred set of metrics for measuring fuzzing outcomes?

Code coverage has long been used as a proxy measure for a fuzzer's bug-finding ability: a bug cannot be found in code never executed. Thus, it is common for fuzzer evaluations to use an independent code coverage metric to enable comparisons; commonly, lines of code (LoC) in the target (we revisit and expand on this approach in Chapter 7). However, controversy surrounds the strength of the correlation between

coverage profiles and bugs [102], with seemingly contradictory results across multiple works due to the incorrect use of statistical analyses (i.e., stratification) [42]. Even when the correlation is strong [23, 76, 117], the *agreement* on which fuzzer is superior may be weak; i.e., "*the fuzzer best at achieving coverage may not be best at finding bugs*" [23].

The potentially weak correlation between code coverage and bug finding has led to the development of "bug-based" benchmarks. A fuzzer's ultimate goal is to find bugs, so why not measure this directly? Unfortunately, realistic bug-based benchmarks are challenging to build and maintain: they require ground-truth knowledge of bugs [84] and must be regularly updated to prevent overfitting [18, 23]. Moreover, the distribution of bugs may not be uniform or related to the actual possibility of bugs in the target [74].

The LAVA, CGC, and REVBUGBENCH benchmarks contain both a large number of programs and bugs; however, these bugs—and, in the case of the CGC, programs—are synthetic and are considered unrepresentative of "real-world" codebases (e.g., because of their small size). In contrast, Goerz et al. [74] use *mutation analysis* to assess a fuzzer's performance, creating (faulty) variants of real-world programs. In contrast, Magma and UNIFUZZ contain both real-world targets and bugs, but the former contains only a small number of each, while the latter has no ground truth to validate results.

Table 2.1 summarizes our discussion on benchmark suites. We use both coverage- and bug-based benchmarks throughout this dissertation; primarily FUZZBENCH and Magma.

2.5.2 Preprocessing

Directed fuzzers (e.g., AFLGo [24], CAFL [121], and BEACON [99]) use a preprocessing stage to determine program locations to drive the fuzzer toward. This preprocessing stage computes a distance metric to the target location(s). The distance is calculated using static control- and data-flow analyses, with the result stored in $\mathcal{R}$.

We extend these static control- and data-flow analyses (and augment them with analogous dynamic analyses) in Chapters 6 and 7 to more accurately measure and reason about fuzzers' state space exploration.

2.5.3 Seed Selection

Mutational fuzzers require an initial seed set to bootstrap the fuzzing process. Typically, these seed sets contain a small number of inputs that $\mathcal{P}$ is known to accept (i.e.,

TABLE 2.1: Comparison of fuzzer benchmark suites. We characterize benchmarks across three dimensions: (i) the targets making up the benchmark; (ii) the bugs that exist across these targets (Goerz et al. [74] use mutants rather than bugs); and (iii) the coverage measure used. For both "targets" and "bugs" we count the number of targets/bugs ("#") and classify their type as "**R**eal" or "**S**ynthetic". Bug density is the mean number of bugs per target. Finally, ground truth may be (a) available ("✓"), (b) available but not easily accessible (" ◗"), or (c) unavailable ("✗").

Benchmark	Targets		Bugs				Coverage
	#	Type	#	Type	Density	Ground Truth	
LAVA-M	4	R	2,265	S	566.25	✓	–
CGC	131	S	590	S	4.50	◗	–
FTS	24	R	47	R	1.96	◗	Known source location
FUZZBENCH	32	R	8	R	0.25	◗	LoC
Magma	9	R	138	R	15.33	✓	–
UNIFUZZ	20	R	?	R	?	✗	LoC
REVBUGBENCH	10	R	7,910	S	791	✓	–
Goerz et al. [74]	7	R	141,278	S	18,754	✓	–
Open-source software	–	R	?	R	?	✗	LoC

the seeds are "well-formed" and do not trigger errors). Perhaps counter-intuitively, the "empty seed"—i.e., a zero-length seed—is also a popular starting seed because it "*is an easy way to baseline a significant variable in the input configuration*" [115].

In contrast, generational fuzzers generate their initial seeds. For example, QuickFuzz [77] generates seeds from file-format-handling libraries available in the Haskell programming language. Finally, some prior work combines both mutational and generational techniques. For example, Skyfire [215] learns a probabilistic context-sensitive grammar (PCSG) from both an initial seed set and ANTLR [169] grammar, and then uses the PCSG to generate uncommon inputs with diverse grammar structures (i.e., those structures with low probability). Similarly, SLF [228] is a "seedless fuzzer" capable of generating valid seeds from scratch using mutational fuzzing.

We analyze both (a) prior work in seed selection, and (b) the impact seed selection has on fuzzing outcomes in more detail in Chapter 4.

2.5.4 Seed Scheduling

"Interesting" inputs (i.e., leading to new coverage) are added to $\mathcal{S}$. Given $\mathcal{S}$ grows rapidly, what is the best approach for selecting the next input to mutate in `ChooseNext`? To answer this question, CollAFL [67] introduced a suite of seed selection techniques prioritizing inputs which (a) exercise edges in $\mathbb{C}_{\mathcal{S}}$ with many unexercised *neighbor edges*, (b) exercise edges in $\mathbb{C}_{\mathcal{S}}$ with many unexercised *descendant paths*, and (c) exercise paths containing many memory access operations. TortoiseFuzz [217] similarly

prioritizes inputs exercising sensitive memory operations at different granularities (function, loop, and basic block levels) to improve the discovery of memory safety vulnerabilities. Finally, `K-Scheduler` [190] applies centrality measure from graph analysis to approximate the likelihood of executing new code by mutating a given input (and subsequently scheduling that input for mutation).

2.5.5 Energy Assignment

Once an input $\iota_i \in \mathcal{S}$ has been selected for mutation, how many times should this input be mutated before selecting a new input $\iota_j \in \mathcal{S}$? This count is determined by how much energy e is assigned to ι_i in `AssignEnergy`. Both AFLFast [22] and AFLGo [24] demonstrated improvements when more energy was assigned to inputs exercising low-frequency coverage elements in $C_{\mathcal{S}}$ (compared to inputs exercising high-frequency elements). This was achieved by modeling coverage-guided fuzzing as a Markov chain. Similar improvements have been made using other statistical models (e.g., Multi-Armed Bandit [212, 229]).

2.5.6 Mutation

The fuzzer must first decide *where* to mutate. Random selection of input bits/bytes is highly inefficient, because most of the input may be unrelated to changes in $\mathcal{P}$'s behavior. Dynamic taint analysis (DTA) is commonly used to address this inefficiency (e.g., in VUzzer [176], Angora [37], and ParmeSan [163]). DTA improves mutation accuracy by tracking the flow of input bytes[3] in $\mathcal{P}$, informing the fuzzer which bytes affect control flow (and hence which bytes to mutate). Unfortunately, accurate DTA is computationally expensive and requires significant manual effort (e.g., to specify taint policies) [47, 95, 192]. This has led to fuzzers avoiding DTA in favor of *approximate taint tracking*. For example, REDQUEEN [9] and AFL++ [63] use "input-to-state correspondence", a technique based on the intuition that "*parts of the input directly correspond to the memory or registers at run time*" [9]. Similarly, GREYONE [68] infers taint by monitoring variable values during mutation. These approaches sacrifice precision for run-time performance.

After deciding where to mutate, the fuzzer must choose *how* to mutate. Following Section 2.4, this depends on how much knowledge the fuzzer has of the input structure. For example, NAUTILUS [8] uses grammar-aware mutations that operate on a parse tree representation of s. Grammatron [199] improves the performance of grammar-aware mutation by restructuring the parse tree as a *finite state automaton*.

[3] The exact granularity (e.g., bits, bytes) depends on the specific *taint policy*. The taint policy specifies the relationship between an instruction's input and output operand(s).

Fuzzers typically resort to bit flipping, byte substitution, and string concatenation when the input structure is unknown. Even if the complete input grammar/structure is unknown, fuzzers may be able to leverage "dictionaries" containing keywords from the input format (e.g., the PNG chunk specifiers `IDAT`, `IEND`, and `PLTE`). Rather than blindly substituting random bytes, the fuzzer can instead substitute keywords from the dictionary.

To alleviate the inefficiencies brought on by the randomness of the mutation process, MOPT [134] uses particle swarm optimization to derive an optimal probabilistically distribution from which a mutation operator is selected. Similarly, EMS [133] captures mutation strategies from previous fuzz runs and uses this history to find interesting inputs that trigger unique paths and crashes.

2.5.7 Execution

Blackbox fuzzers blindly execute $\mathcal{P}$ without any feedback mechanism. In contrast, greybox fuzzers use positive feedback to drive the fuzzer toward unexplored parts of $\mathcal{P}$'s state space. A coverage map $\mathbb{C}_{s'}$ is used to measure which parts of $\mathcal{P}$'s state space was executed with (mutated) input ι'.

We analyze different coverage metrics used by greybox fuzzers in Chapter 3. Moreover, we introduce a new coverage metric based on data flow coverage (specifically, *def-use* chains) in Chapter 5.

2.5.8 Crash Detection

The simplest form of crash detection is to rely on the underlying hardware/OS to report a violation (e.g., segmentation faults, divide-by-zero). Unfortunately, this approach may miss bugs (e.g., an out-of-bounds memory access may not necessarily trigger a segmentation fault). Thus, *sanitizers* are often deployed to detect a wider variety of bug classes more accurately. For example, AddressSanitizer (ASAN) [187] is used for detecting memory safety violations, UndefinedBehaviorSanitizer (UBSAN) [210] is used for detecting undefined behavior, and type confusion errors are detectable with HexType [105]. Song et al. [197] provide a more detailed summary of other sanitizers.

2.5.9 Stopping Condition

How long should a fuzzing campaign last? Klees et al. [115] found campaign timeouts ranging from 1 h to days and weeks, with 24 h being the most common campaign length. Further, Böhme, Liyanage, and Wüstholz [20] developed estimators for

TABLE 2.2: Techniques used to ensure statistically-sound results. "Summary" techniques are used to summarize a set of observations (i.e., over a set of N trials). "Comparison" techniques enable comparisons of observations.

	Bug finding	Coverage profiles
Summary	Restricted mean survival time (RMST)	Arithmetic mean
Comparison	log-rank test	Mann-Whitney U-test

determining when to stop a fuzzing campaign such that the *residual risk* of a bug remaining undiscovered is minimized.

2.6 Ensuring Statistically-Sound Results

Proper benchmarking is essential in systems security [118]. In particular, the highly-stochastic nature of fuzzing (e.g., due to random mutations [223]) requires the use of appropriate statistical techniques when analyzing results [5, 115]. We measure fuzzing outcomes across two axes: *bug finding* and *coverage profiles* (the two most common metrics used to analyze the results of a fuzzing campaign, per Section 2.5.1). We adopt the following approaches (and terminology) to ensure statistical significance when analyzing results across these two axes (summarized in Table 2.2).

2.6.1 Repeated, Independent Trials

We run one fuzzing *campaign* per {target $\times$ fuzzer $\times y$} combination, where y is a *dependent variable*[4] that changes depending on the given evaluation (e.g., in Chapter 4 we examine the effect different seed selection strategies have on fuzzing outcomes; here, y is the set of seed selection strategies). Each campaign consists of N independent *trials* of length T hours.

2.6.2 Bug Survival Time

We statistically analyze and compare time-to-bug based on the recommendations of Böhme and Falk [19]. Following prior work [5, 211], we apply *survival analysis* [116] to time-to-bug events. This allows us to handle *censoring*: individual trials where a given bug is not found.

For each campaign (i.e., set of N repeated T hour trials) we model a bug's *survival function*—the probability of a bug being found over time—using the Kaplan-Meier estimator [110]. Integrating this survival function with upper-bound T gives a bug's

[4] An evaluation may have more than one dependent variable.

RMST for a given campaign. Shorter bug survival times indicate better performance. We report the RMST and 95 % confidence interval (CI) across each campaign.

We also use the *log-rank test* [140] to statistically compare bug survival times. The log-rank test is computed under the null hypothesis that two {target $\times$ fuzzer $\times y$} combinations share the same survival function. Thus, we consider two fuzzing campaigns to have statistically-equivalent bug survival times if the log-rank test's p-value > 0.05. We use the log-rank test because it is non-parametric and thus makes no assumptions about the underlying distribution of bugs.

2.6.3 Summary Statistics

We use both the arithmetic mean and 95 % CI to summarize non-bug-related fuzzing results (e.g., coverage profiles generated by different fuzzers). We use *bootstrapping* [57] to compute CIs, again because it is a non-parametric technique.

2.6.4 Statistical Significance

We use the non-parametric Mann-Whitney U-test [139] to compare and determine the ranking of A and B, where A and B are non-bug-related fuzzing results (e.g., coverage profiles). A p-value > 0.05 implies a statistically-equivalent ranking.

2.7 Chapter Summary

This chapter gives an overview of fuzzing, including both fundamental concepts and the state-of-the-art. We discussed coverage-guided greybox fuzzers in depth because they are the most pervasive type of fuzzer and the focus of this dissertation.

In the next chapter, we delve deeper into the coverage metrics fuzzers use to measure state space exploration. After this, we focus on our contributions to improve fuzzers' state space exploration: (i) how we bootstrap the fuzzing process to lower the cost of the initial state space search (Chapter 4); (ii) a data-flow-based coverage metric, providing an alternative view of a program's state space (Chapter 5); (iii) an investigation of techniques for measuring a fuzzer's state space search (Chapter 6); and (iv) a comprehensive evaluation of the coverage metrics discussed throughout this dissertation (Chapter 7).

Chapter 3

Coverage Metrics

Greybox fuzzers use one or more *coverage metrics* to abstract and approximate a target's state space. In doing so, the fuzzer maintains a notion of "progress": expanding coverage implies exploring new program behaviors. In this chapter, we survey and analyze widely-used coverage metrics to understand their advantages and disadvantages.

3.1 Introduction

A fuzzer finds bugs by maximizing its coverage of the target's state space. Greybox fuzzers are guided by a feedback loop based on one or more *coverage metrics*. Per Algorithm 1 (Chapter 2), the coverage measured (in $\mathbb{C}_s$) tells the fuzzer whether to keep or discard a seed. An effective greybox fuzzer requires coverage metrics that are: (i) **accurate**: program behaviors are accurately approximated; and (ii) **performant**: run-time overheads are minimal. Accurate metrics lead to more-informed decisions in `IsInteresting` (line 16 in Algorithm 1), guiding the fuzzer to explore more of the target's state space (and increasing the likelihood of finding bugs). In contrast, performant metrics ensure that the fuzzer spends most cycles executing inputs. There is a clear tension between these requirements: more accurate metrics require more intrusive instrumentation to capture subtle changes in program state. But these introduce higher run-time overheads.

Requirement

The *accuracy* of a fuzzer's coverage metric(s) must be balanced with *performance*, ensuring available cycles are spent finding as many bugs as possible.

Chapter outline. Keeping these accuracy and performance requirements in mind, the remainder of this chapter analyzes the coverage metrics used in 79 coverage-guided greybox fuzzers published from 2013–2022.

3.2 Coverage Metrics in Theory and Practice

Figure 3.1 shows the genealogy of 79 fuzzers published from 2013–2022 and the coverage metrics they use. To realize a coverage metric in practice, fuzzers introduce inaccuracy to reduce run-time overheads (in both time and space). In the remainder of this chapter we describe ideal coverage metrics and how they are realized in practice.

3.2.1 Basic Block

Basic-block ("block") coverage is the simplest form of control-flow coverage,[1] relying exclusively on block labels. There is no universal approach for labeling a block: UnTracer [151] uses static binary analysis to identify each block in the target and assign it an integer label, while VUzzer [176] uses a block's run-time address. The simplicity of block coverage means that it can be universally applied with low run-time overhead. However, this simplicity comes at a cost: fuzzers cannot distinguish block sequences and simply count how many times they execute each block. This limits the fuzzer's ability to differentiate between a loop's forward and backward edges.

3.2.2 Edge

Edge coverage allows the fuzzer to differentiate between different orderings of the same two blocks. An idealized edge can be described by pairing the previously-executed block label l_{prev} with the current block label l, capturing both intra- and inter-procedural edges. Recording edge coverage this way requires double the memory needed to record the same trace using block coverage. This, combined with the fact that a target typically has many more edges than blocks, means that *approximate* edge coverage is used in practice. We categorize techniques for approximating edge coverage as one of: (i) coarse-grained probabilistic; (ii) coarse-grained link-time optimization (LTO); and (iii) fine-grained control-flow graph (CFG) transformation.

Coarse-grained probabilistic. Randomly-assigned block labels are hashed together to identify an edge between blocks. American Fuzzy Lop (AFL) [231]—perhaps the

[1] Function level coverage is even simpler but is too coarse-grained for fuzzers and is never used in practice.

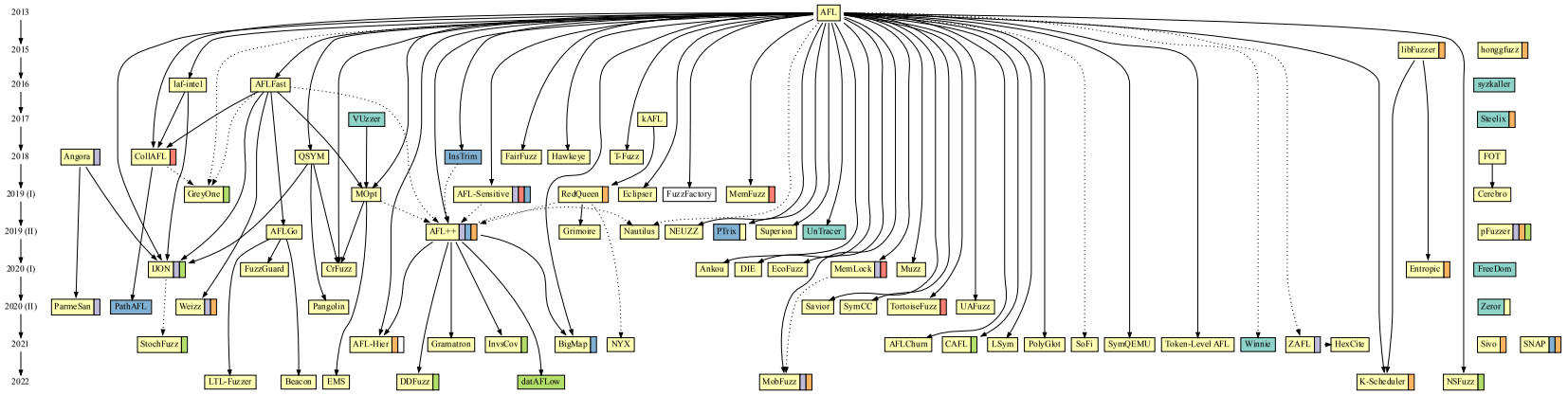


FIGURE 3.1: Greybox fuzzer genealogy for 79 fuzzers published over the last ten years. Coverage metrics are colored: basic block, edge, calling context, memory access, path, comparison operator, and program variable. The size of a rectangle within a node differentiates primary and secondary metrics. Solid edges indicate the fuzzer is built on top of the parent fuzzer, while dotted edges indicate the fuzzer reimplements a feature from the parent fuzzer. FuzzFactory is left uncolored because it allows for user-specified coverage feedback [164], while AFL-HIER [213] uses function coverage, which we do not include in our metrics.

most widely-used greybox fuzzer—uses the following hash algorithm:

$$\begin{aligned} p &\leftarrow ((l \gg x) \oplus (l_{\text{prev}} + z)) \\ l_{\text{prev}} &\leftarrow l \gg y \end{aligned} \tag{3.1}$$

Where l is the randomly-generated block identifier (assigned at compile time), l_{prev} is the identifier of the previously-executed block, $x = z = 0$, and $y = 1$. The result p is used as an index into $\mathbb{C}$. Right-shifting l allows AFL to differentiate between different orders of two blocks. While performant, this approach is prone to hash collisions, resulting in miscounted edges and thus reduced precision [2, 67, 127]. Notably, this problem is not unique to AFL: all fuzzers built on top of AFL [22, 24, 121, 170, 184, 216] or inspired by AFL [8, 37, 163] share the same implementation decision, and thus share the same loss in precision.

Coarse-grained LTO. CollAFL [67] (and fuzzers based on it; e.g., GREYONE [68], PathAFL [227]) uses Eq. (3.1) by selecting x, y, and z at link-time so that edges have unique hashes. Importantly, this approach only ensures the elimination of collisions for known edges; run-time collisions may still exist.

Fine-grained CFG transformation. Hash collisions can be eliminated by breaking *critical edges* in the CFG. Breaking critical edges (i.e., edges whose start/end blocks have multiple outgoing/incoming edges, respectively) allows block coverage to *precisely* imply edge coverage (assuming the number of blocks is fewer then the size of $\mathbb{C}$) at the cost of increased compile-time complexity. AFL++ [63], honggfuzz [207], and libFuzzer [188] use LLVM's SanitizerCoverage (SANCOV) instrumentation [208] to achieve this.

3.2.3 Path

While edge coverage improves on basic-block coverage by differentiating between forward and backward edges in a loop, it is unable to reason about the complete ordering of *all* blocks in an execution trace. Path coverage resolves this issue by tracking a complete execution trace—consisting of the sequence of executed blocks—through the target. However, using this definition of path coverage is infeasible: the number of paths (which can be infinite) and the memory overhead associated with maintaining this coverage information is prohibitive [67].

AFL-Sensitive [214] introduced *n-gram coverage* (later adopted by AFL++), where a history of n edges is used to approximate a path. These n edges are xor-ed together to generate an index into $\mathbb{C}$. In contrast, INSTRIM applies "CFG-aware" instrumentation to reduce the number of instrumented blocks (improving run-time performance) while ensuring different execution paths remain distinguishable [97]. Finally, PTRIX [40] uses Intel Processor Tracing (PT)—a low-overhead execution tracing feature available in modern Intel CPUs—to record program paths while fuzzing. PTRIX proposes an *elastic* approach to measure code coverage: encoded PT packets are concatenated and then hashed to achieve low-overhead path coverage. Like the coarse-grained probabilistic edge coverage metric, these approaches are lossy and collisions may occur.

3.2.4 Calling Context

Function call information allows a fuzzer to track context-sensitive coverage information (per Section 1.2, this was how Ormandy [162] discovered CVE-2021-43527). However, efficiently computing accurate calling context is difficult [25, 206]. For example, each stack frame requires tracking: (i) the current function; (ii) the current block; (iii) the set of local variables and their values; and (iv) allocated memory. This is (a) computationally expensive to convert into a single index into $\mathbb{C}$, and (b) memory-intensive for efficient access (i.e., without walking the stack) [25]. Unsurprisingly, this leads fuzzers to approximate calling context in practice.

AFL-Sensitive [214], Angora [37], ParmeSan [163], and AFL++ [63] approximate calling context by maintaining a rolling xor hash of call site identifiers (similar to the n-gram approach described in Section 3.2.3, but limited to function call/return sites). AFL++ also supports *k-call-site-sensitivity*, where the last k call sites are recorded in the hash. This hash is combined with edge coverage to provide *context-sensitive edge coverage* (again, with the possibility of hash collisions). While xor has minimal overhead, it reduces precision because it does not differentiate between recursive calls of arbitrary depth. Other approaches (e.g., those proposed by Bond and McKinley [25] and Sumner et al. [206]) are needed if differentiating between recursive function calls of arbitrary depth is required.

3.2.5 Memory Access

Memory-safety vulnerabilities remain one of the most common bug classes [147]. Consequently, some fuzzers incorporate *memory accesses* into their coverage metrics; inputs that lead to new memory access patterns (that may subsequently lead to memory-safety errors) are prioritized. This requires encoding memory reads/writes

in $\mathbb{C}$. When this encoding is code-dependent (e.g., encodes a memory access location or static address), then collected coverage approximates *control flow*. However, when it is data-dependent (e.g., encodes an input-controlled value or address), collected coverage approximates *data flow*. Therefore, $\mathbb{C}$ may record information about the location of the memory access (the code location) or the target of the memory address (the data location). Similarly, the value read from/written to a particular memory address may (or may not) be included in the coverage metric.

AFL-Sensitive proposes *memory-access-aware* edge coverage, where both edge coverage and memory reads/writes are combined in the same $\mathbb{C}$ (although reads and writes are distinguished by allocating their hash to different regions of $\mathbb{C}$). MEMFUZZ [49] adopts a similar approach, except (a) a lightweight static analysis limits instrumentation to "*interesting memory accesses*", and (b) edge coverage and memory accesses are disjoint in $\mathbb{C}$ (allowing for more fine-grained checks for new program behaviors). Both AFL-Sensitive and MEMFUZZ are prone to hash collisions in $\mathbb{C}$, reducing precision. Finally, CollAFL proposes a seed-selection policy that prioritizes inputs with memory access operations. Unlike the previous approaches, CollAFL counts memory accesses *post hoc*: after executing a testcase s, edges covered by s are retrieved from $\mathbb{C}_s$ and the number of memory accesses made per covered block is determined. This count is combined with $\mathbb{C}_s$ to weigh s.

3.2.6 Comparison Operator

Instrumented comparison operators offer a form of lightweight data flow. Both honggfuzz and libFuzzer capture comparisons containing constant operands, feeding these constants into `MutateInput` (Algorithm 1). Steelix [123] instruments comparisons to measure "comparison progress". Comparison progress—which measures how many consecutive bytes are matched—is stored with block coverage in $\mathbb{C}$. Similar to MEMFUZZ's memory access instrumentation, Steelix limits its comparison instrumentation to "interesting" comparisons; one byte and return value comparisons are ignored.

3.2.7 Program Variable

The previously-discussed coverage metrics are primarily control-flow based. While pervasive, control-flow-based coverage metrics lose important information about program behavior. For example, greybox fuzzers rely on coverage information to decide which input mutations lead to new program behaviors. However, the process for uncovering new behaviors can be highly inefficient, because a fuzzer guided by control-flow coverage alone cannot identify *which* mutated input bytes led to new

program behavior. Differences in data access and manipulation within a single code path are lost. Data-flow-based metrics attempt to solve this problem.

INVSCOV [61] augments coarse-grained probabilistic edge coverage with values and relationships among program variables. However, rather than looking at all program variables (resulting in state explosion), INVSCOV only considers "*likely invariants*": constraints on the values and relationships of variables learned by dynamically tracing a seed set. Invariant violations are encoded in $\mathbb{C}$, signaling to the fuzzer (via `IsInteresting`) that a new program state has been reached.

`DDFuzz` [141] also augments coarse-grained probabilistic edge coverage with data flows between program variables. Here, data flows are derived from the target's data dependency graph (DDG). DDGs describe the data flows between instructions in a program and are traditionally used by optimizing compilers [60]. Like INVSCOV, `DDFuzz` only considers a subset of program variables (again, to prevent state explosion): variable definition sites are restricted to `load` and `alloca` instructions in the LLVM intermediate representation (IR), while variable uses are restricted to `store` and `call` instructions. Importantly, the DDG is not used to expand coverage. Instead, the DDG is used to revisit particular program locations via new paths involving different variable dependencies (thus exposing program states not visible in the CFG). Consequently, further filtering is applied to discard data flows subsumed by edge coverage. Both INVSCOV and `DDFuzz` use heuristics to decide which program variables to track. This reduces run-time overhead, but may mean important variables are missed.

Directed greybox fuzzers (e.g., AFLGo [24]) aim to reach a target location in a codebase. They typically use a control-flow-based distance metric to determine how close a given test case is to this location. CAFL [121] extends this idea by incorporating "*data conditions*" into the distance metric. Satisfying data conditions requires (a) capturing relevant variables, and (b) computing distances[2] between these variables' values. `IsInteresting` uses these distances to determine if the fuzzer is making progress toward the target location.

3.3 Chapter Summary

In this chapter, we analyzed the coverage metrics used by 79 fuzzers released from 2013–2022. In doing so, we identified and surveyed three primary metrics (basic block, edge, and path coverage) and four secondary metrics (calling context, memory access, comparison operator, and program variable). Among these seven metrics,

[2] This calculation depends on the comparison operator used in the condition.

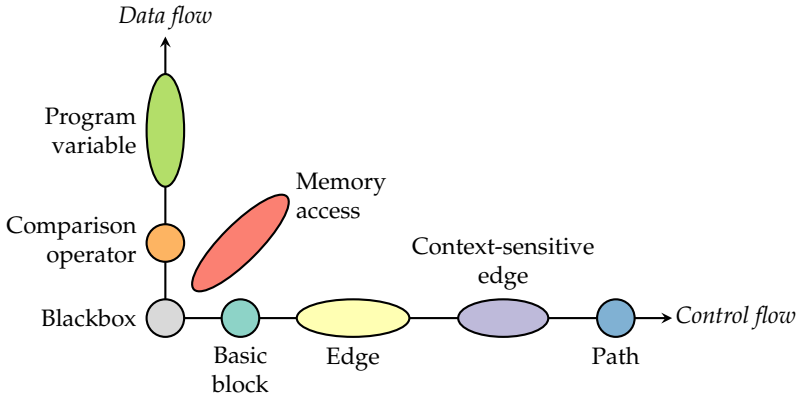


FIGURE 3.2: Qualitative comparison of greybox fuzzers' coverage metrics. Precision increases from left to right (along the x-axis) and bottom to top (along the y-axis). Most fuzzers rely exclusively on control-flow coverage (along the x-axis). However, these metrics can be augmented with data-flow-based metrics (along the y-axis). Some metrics (e.g., edge) vary in implementation, leading to varying levels of precision.

there exist a wide variety of design and implementation choices (edge coverage alone has been implemented in several different ways according to precision/performance tradeoffs).

Inspired by the hierarchy of control- and data-flow coverage metrics proposed by Horgan, London, and Lyu [94], Fig. 3.2 qualitatively summarizes and compares the coverage metrics we discussed in this chapter. Blackbox fuzzers appear in the bottom-left corner: they have no feedback mechanism and hence are the least precise. Most greybox fuzzers lie on the x-axis, as the majority use some form of edge coverage. Implementation decisions affect the precision of edge, memory access, and program variable metrics. Similarly, memory access coverage can be both control- and data-flow-based depending on implementation tradeoffs. A third axis—coverage of *input structure*—could also be included on this graph. This is particularly relevant to grammar-based fuzzers, where (a) the structure of the input format is known, and (b) maximizing the generation of syntactic input features will lead to increased coverage of the target's state space [83]. However, we omit this axis for readability and because our focus is on a target's state space, not its input space.

This chapter sets the scene for the remainder of this dissertation: how do we use these (sometimes incompatible) opinions about coverage to measure and improve a fuzzer's state space exploration. In the next chapter we return to the start of a fuzzing campaign (① in Fig. 1.1, Chapter 1); how to bootstrap the fuzzer to simplify its state space search.

Chapter 4

Seed Selection

Mutational fuzzers require an initial set of non-crashing inputs (a "corpus") to bootstrap their state space search. These inputs—or *seeds*—should exercise different parts of the target's state space. By mutating these seeds, the fuzzer aims to reach new, potentially bug-inducing states. In this chapter, we present: (i) an optimal approach for constructing corpora from a large collection of potential inputs; and (ii) a comparative evaluation of how corpus selection affects fuzzing outcomes.

4.1 Introduction

"Garbage in, garbage out" is the idea that the quality of input data determines the quality of outputs. It applies to many fields of computing, including fuzzing. We argue that fuzzers more efficiently explore a target's state space when bootstrapped with "high quality" inputs. What determines the quality of these inputs? We answer this question here.

Chapter outline. We begin by systematically examining seed selection practices used in prior work (Section 4.2). Then, we focus on the problem of *corpus minimization* (Section 4.3): given a (potentially large) corpus of initial seeds, how to best select a subset of these seeds (e.g., to remove redundancy) to bootstrap a fuzzer? Here we also introduce OPTIMIN, our corpus minimization technique. OPTIMIN is unique in that—unlike other corpus minimization techniques—it produces an *optimal minimum* corpus. Finally, we perform a comprehensive empirical evaluation (33 CPU-yr) to understand how seed choice—including various corpus minimization techniques—affects the fuzzing process (Section 4.4).

4.2 Seed Selection Practices

The process for selecting the initial corpus of seeds varies wildly across fuzzer evaluations and deployments. We systematically review this variation in the following sections, first considering experimental evaluations, followed by industrial-scale deployments.

4.2.1 An Ideal Initial Corpus

Ideally, an initial fuzzing corpus should contain inputs that exercise a diverse set of states in the target's state space. A diverse set of states means that a fuzzer's mutations is more likely to "nudge" an input into a new, potentially buggy, state. In contrast, having only a small number of states covered by the initial corpus means the fuzzer must waste time attempting to reach valid states, rather than focusing on exploring "corner cases" in the target's state space.

Seeds in a fuzzer's initial corpus should also be fast to execute, so that a fuzzer's iteration rate is not negatively impacted. Thus, large seeds are discouraged.

Finally, there should be minimal redundancy in the corpus. That is, seeds that cover the same states should be removed from the corpus. This is analogous to *test suite minimization*, where redundant tests should be removed from the test suite to reduce the time required to run the test suite [86, 98, 126, 220].

4.2.2 In Experimental Evaluation

Remarkably, Klees et al. [115] found "*most papers treated the choice of seed casually, apparently assuming that any seed would work equally well, without providing particulars*". In particular, of the 32 papers they surveyed (authored between 2012 and 2018): ten used non-empty seed(s), but it was not clear whether these seeds were valid inputs; nine assumed the existence of valid seed(s), but did not report how these were obtained; five used a random sampling of seed(s); four used manually constructed seed(s); and two used empty seed(s). Additionally, six papers used a combination of seed selection techniques. Klees et al. [115] conclude "*it is clear that a fuzzer's performance on the same program can be very different depending on what seed is used*" and recommend "*papers should be specific about how seeds are collected*".

We examine an additional 74 papers published since 2018[1] to see if these recommendations have been adopted. Table 4.1 summarizes our findings.

[1] Published in top security and software testing venues: ACSAC, ASIA CCS, CCS, EURO S&P, NDSS, RAID, S&P, SEC, ASE, ESEC/FSE, ICSE, and ISSTA.

TABLE 4.1: Summary of past fuzzing evaluation, focusing on seed selection. We adopt the categories and notation used by Klees et al. [115]: **R** means randomly sampled seeds; **M** means manually constructed seeds; **G** means automatically generated seed; **N** means non-empty seed(s) with unknown validity; **V** means the paper assumes the existence of valid seed(s), but with unknown provenance; **E** means empty seeds; **/** means different seeds were used in different programs, but only one kind of seeds in one program; and an empty cell means that the paper's evaluation did not mention seed selection. We introduce an additional category: **V*** means valid seed(s) with known provenance. We also indicate whether the evaluation is reproducible with the same seeds.

Year	Paper	Seed	Reproducible
2018	CollAFL [67]		✗
	Hawkeye [34]	E/V*	✗
	QSYM [230]	M, V*	✓
2019	AFL-Sensitive [214]	E/V*	✓
	CEREBRO [122]		✗
	Eclipser [46]	M/V*	✓
	FUZZFACTORY [164]	M, V*	✓
	GRIMOIRE [16]	M	✓
	MEMFUZZ [49]	E, V	✗
	MOPT [134]	R	✓
	NAUTILUS [8]	G, M	✓
	NEUZZ [193]	G, V*	✓
	PFUZZER [142]	E	✓
	PTRIX [40]	V*	✓
	REDQUEEN [9]	M	✓
	Superion [216]	R, V	✗
	UnTracer [151]	V	✓
	Zest [165]	V	✓
2020	[19]	E	✓
	Ankou [137]	V*	✓
	CRFUZZ [198]		✗
	CUPID [78]	E/V*	✓
	DIE [168]	V*	✓
	EcoFuzz [229]	V*	✓
	ENTROPIC [21]	E/V*	✓
	FuZZan [104]	E/V*, R	✓
	FuzzGen [103]		✗
	FuzzGuard [240]	E/V	✗
	GREYONE [68]	R	✗
	IJON [7]	M/R/V	✗
	Magma [84]	V*	✓
	MEMLOCK [219]	V	✓
	MEUZZ [39]	V*	✓
	MTFuzz [191]	G, V*	✗
	MUZZ [35]	V	✗
	PANGOLIN [100]	V*	✓
	ParmeSan [163]	E	✗
	PathAFL [227]	E/R/V	✗
	SAVIOR [41]	V*	✓
	SYMCC [173]		✗
	TortoiseFuzz [217]	R	✗
	UAFUZZ [156]	E/V*	✗
	WEIZZ [62]	M/V/V*	✗
	ZEROR [237]	E/V*	✗
2021	[20]	V*	✓
	AFLCHURN [239]	V*	✓
	AFL-HIER [213]	M/V*	✓
	CAFL [121]	V*	✗
	Controlled Compilation [195]		✗
	Gramatron [199]	E	✓
	HEXCITE [153]		✗
	INVSCOV [61]	V	✗
	LSym [145]	V	✗
	POLYGLOT [43]	V*	✗
	SIVO [157]	R	✗
	SNAP [51]	V	✗
	SoFi [85]	R, V, V*	✗
	STOCHFUZZ [236]	V/V*	✓
	SYMQEMU [174]	V*	✓
	Token-Level AFL [184]	V*	✗
	UNIFUZZ [124]	R	✓
	ZAFL [152]	E, V*	✓
2022	[23]	V*	✓
	[222]	G	✗
	[223]	V*	✗
	BEACON [99]	V*	✗
	DDFuzz [141]	R	✗
	EMS [133]	R/V	✗
	JIGSAW [36]	V*	✓
	K-Scheduler [190]	V*	✓
	LTL-Fuzzer [143]	R/V*	✓
	MobFuzz [232]	V*	✗
	PATA [125]	E/R/V*	✗
	TRUZZ [233]	V*	✓

Unreported seeds. Seven papers make no mention of their seed selection procedure. One, FuzzGen, explicitly mentions "*standardized common seeds*" [103] are key to a valid comparison, yet does not mention the seeds used.

Benchmark and fuzzer-provided seeds. Eight papers (Hawkeye, FUZZFACTORY, CUPID, ENTROPIC, FuZZan, ZEROR, STOCHFUZZ, and PATA) evaluate fuzzers on the Fuzzer Test Suite (FTS) [75], which provides seeds for 14 of its 24 targets (commit `6955fc9`). Of these seed sets, eight contain only one or two seeds. When no seed is provided, three papers (CUPID, FuZZan, and PATA) default to the empty seed. It is unclear what seeds the remaining five papers (Hawkeye, FUZZFACTORY, ENTROPIC, ZEROR, and STOCHFUZZ) default to. When evaluating on other fuzzer benchmarks—including FUZZBENCH [144], Magma [84], and LAVA-M [55]—the provided seeds are used. Finally, seven papers (AFL-Sensitive, PTRIX, EcoFuzz, PANGOLIN, SAVIOR, Wu et al. [223], and MobFuzz) used the singleton seed sets provided by American Fuzzy Lop (AFL).

Manually-constructed seeds. Two papers (REDQUEEN and GRIMOIRE) used "*an uninformed, generic seed consisting of different characters from the printable ASCII set*" [9]. However, the authors do not justify (a) why this specific singleton corpus was chosen, and (b) what impact this choice has on the authors' real-world results, particularly when fuzzing `binutils`, where most of the targets accept non-ASCII, binary file formats (e.g., `readelf`, `objdump`). Similarly, IJON used "*a single uninformative seed containing only the character 'a'*" [7], QSYM used "*a dummy ASCII file containing 256 'A's*" [230], and Eclipser used "*a dummy seed composed of 16 consecutive NULL bytes*" [46]. Finally, AFL-HIER started fuzzing the CGC binaries with a single seed containing the string "`123\n456\n789\n`" [213] (rather than the provided seeds).

Random seeds. Thirteen papers (MOPT, Superion, FuZZan, GREYONE, IJON, PathAFL, TortoiseFuzz, SoFi, UNIFUZZ, `DDFuzz`, EMS, `LTL-Fuzzer`, and PATA) randomly select seeds from either (a) a larger corpus of seeds provided by developers of a particular target, or (b) by crawling the Internet. For example, `DDFuzz` selects a small subset of seeds "*from the `test` directories... in the repositories of the projects*" [141]: these subsets range from three seeds (*bison*) to 65 seeds (*mir*, which contains over 900 test inputs). Which seeds are selected and why is not explained.[2] Of these papers, three (Superion, GREYONE, and SoFi) specifically mention using `afl-cmin`, AFL's corpus minimization tool (discussed further in Section 4.3), to remove duplicate seeds from the random seed set.

[2]The authors were happy to share their starting seeds when contacted.

Empty seeds. Sixteen papers use an empty seed to bootstrap the fuzzing process. In particular, both Böhme, Manès, and Cha [21] and Böhme and Falk [19] explicitly removed the corpora provided by OSS-Fuzz (discussed further in Section 4.2.3) and used the empty seed because they found "*initial seed corpora…are often for saturation: feature discovery has effectively stopped shortly after the beginning of the campaign*".

A reproduction experiment: REDQUEEN. To demonstrate the importance of seed selection, we reproduce an experiment from the REDQUEEN evaluation. Aschermann et al. [9] fuzz several `binutils`' programs, bootstrapping each trial with an "uninformed, generic seed" (discussed previously). Their `readelf` results are particularly striking: notably, AFLFast and honggfuzz cover little code. We repeat this experiment but use a variety of initial seeds, including: (i) the original, uninformed seed; (ii) a single, valid ELF file (from AFL's seed set); and (iii) a collection of ELF files sourced from the ALLSTAR [200] and Malpedia [172] datasets (reduced from 104,737 to 366 seeds using `afl-cmin`). In place of the original REDQUEEN (which we were unable to build and reproduce) we use AFL++ [63] with "CmpLog" instrumentation enabled; this reimplements REDQUEEN's "input-to-state correspondence".

Our results appear in Fig. 4.1 and clearly show seed choice's impact on code coverage. Similar to the results of Aschermann et al. [9], AFLFast bootstrapped with the uninformed seed explores very little of `readelf`'s code: less than 1 %. However, this increases to ∼38 % for AFLFast bootstrapped with the valid ELF file, making it much more competitive against both honggfuzz and AFL++ (although AFL++ still outperforms them both by ∼15 %). Finally, while the `afl-cmin` corpus has a negligible impact on AFLFast and honggfuzz, it results in a significant improvement when fuzzing with AFL++, increasing coverage to ∼60 %.

Requirement

At a minimum, fuzzer evaluations must report the seed set used to bootstrap the fuzzing process. To ensure reproducibility, artifacts must provide the initial seed set (since results vary wildly depending on the seeds used). Ideally, fuzzer evaluations should experiment with different initial seed corpora to see how varying initial seeds affect fuzzing outcomes.

4.2.3 In Deployment

In addition to being an active research topic, fuzzers are frequently deployed to find bugs in real-world software [3, 33, 150, 159]. Notably, security professionals

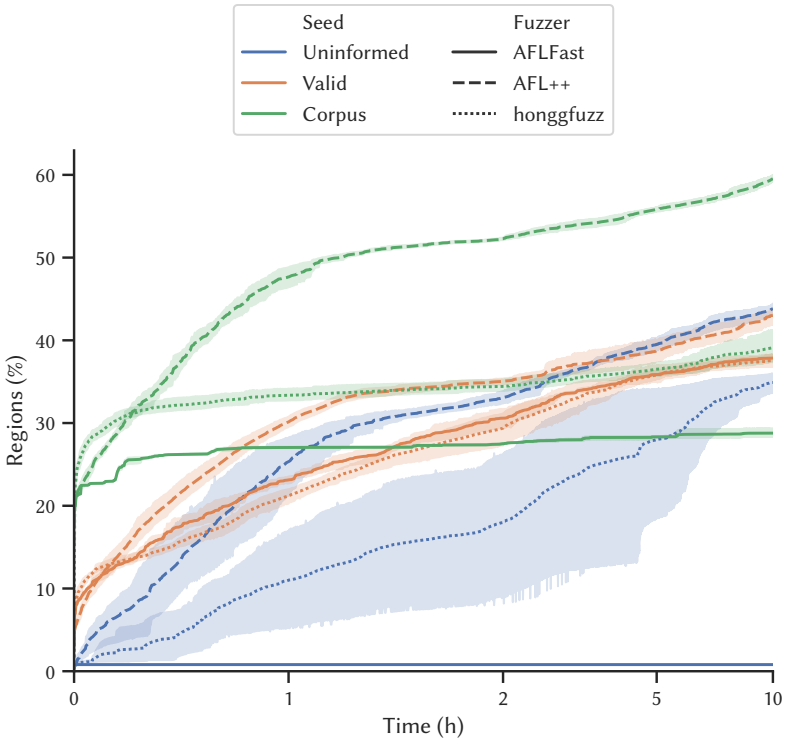


FIGURE 4.1: Code coverage of `readelf` with different initial seed sets. The mean coverage (using Clang's source-based region coverage metric) and 95 % bootstrap confidence interval over five repeated 10 h trials is shown. The x-axis uses a log scale.

also recognize the importance of seed selection. For example, the developers of the Mozilla Firefox browser remark [149]:

> *Mutation-based strategies are typically superior to others if the original samples* [i.e., seeds] *are of good quality because the originals carry a lot of semantics that the fuzzer does not have to know about or implement. However, success here really stands and falls with the quality of the samples. If the originals don't cover certain parts of the implementation, then the fuzzer will also have to do more work to get there.*

In contrast to the evaluation practices described in Section 4.2.2, "industrial fuzzing" eschews small seed sets and the empty seed in favor of large corpora. For example, seed corpora in Google's continuous fuzzing service for open-source software, OSS-Fuzz (commit `0deeef6`), ranges in size from a single seed (e.g., OpenThread, ICU) to 62,726 seeds (Suricata). Of the 363 OSSFuzz projects, 135 projects supply an initial corpus (37 %) for 706 fuzzable targets. The mean corpus size is 1,083 seeds and the median is 36 seeds. More than half of the 135 projects include more than 100 seeds.

We examined the Suricata corpus in more detail because it provided the largest number of seeds (62,726). We found large redundancy in these 62,726 seeds: we reduced the corpus to 31,234 seeds (∼50 % decrease) by discarding seeds with an identical MD5 hash. We were able to reduce the size of the corpus further (down to 145 seeds, a 99 % reduction) by again applying `afl-cmin` (similarly to the `readelf` reproduction experiment described in Section 4.2.2). This redundancy is wasteful, as it leads to seeds clogging the fuzzing queue, which hinders and delays the mutation of more promising seeds. We discuss corpus minimization in the following section.

Requirement

When deploying fuzzers at an industrial scale, it is imperative that seeds exhibiting redundant behavior be removed from the fuzzing queue, as they will lead to wasted cycles.

4.3 Corpus Minimization

Orthogonal to the seed selection practices examined in Section 4.2, many popular fuzzers (e.g., AFL [231], libFuzzer [188], honggfuzz [207]) provide *corpus minimization* (sometimes called *distillation*) tools. Corpus minimization assumes a large corpus of seeds already exists, and thus a corpus minimization tool *reduces* this large corpus to

a subset of seeds that are then used to bootstrap the fuzzing process. How are these seeds selected?

Abdelnur et al. [1] first formalized this problem as an instance of the *minimum set cover problem* (MSCP). The MSCP states that given a set of elements U (the universe) and a collection of N sets $S = s_1, s_2, \ldots, s_N$ whose union equals U, what is the *smallest* subset of S whose union still equals U. This smallest subset $\mathcal{C} \subseteq S$ is known as the *minimum set cover*. Moreover, each $s_i \in S$ may be associated with a weight w_i. In this case, the *weighted* MSCP (WMSCP) attempts to minimize the total cost of elements in $\mathcal{C}$.

(W)MSCP is NP-complete [111], so Abdelnur et al. [1] used a *greedy* algorithm to solve the unweighted MSCP. Here, U consists of code coverage information for the set of seeds in the original collection corpus. Subsequently, code coverage (e.g., basic blocks, edges) continues to be used to characterize seeds in a fuzzing corpus due to the strong positive correlation between code coverage and bug finding [23, 76, 117]. Computing $\mathcal{C}$ is therefore equivalent to finding the minimum set of seeds that still maintains the code coverage observed in the collection corpus.

Several corpus minimization techniques have been proposed since the work of Abdelnur et al. [1]. We discuss prior work in Section 4.3.1, and introduce our own minimization technique, OPTIMIN, in Section 4.3.2.

4.3.1 Prior Work

Peach Minset. The Peach fuzzer [71] provides a corpus minimization tool, Peach Minset. Despite its name, Rebert et al. [177] found that Peach Minset does not in fact calculate $\mathcal{C}$, "*nor a proven competitive approximation thereof*". Peach Minset uses basic block coverage.

MINSET. Rebert et al. [177] extended the work of Abdelnur et al. [1] by also computing $\mathcal{C}$ weighted by execution time or file size. They designed six corpus minimization techniques, simulating and empirically evaluating these techniques over several fuzzing campaigns using the Basic Fuzzing Framework (BFF). These techniques use basic block coverage. Rebert et al. [177] found that UNWEIGHTED MINSET—an unweighted greedy-reduced minimization—performed best in terms of minimization ability, and that the PEACH SET algorithm (based on the previously-discussed Peach Minset) found the highest number of bugs. We extend the work of Rebert et al. [177] in Section 4.4 with an evaluation based on modern coverage-guided greybox fuzzing.

LibFuzzer. LLVM's libFuzzer [188] provides a corpus "merge" feature for minimizing a large corpus "*while still preserving the full coverage [of the collection corpus]*" [130]. Unlike MINSET and Peach Minset, which use basic block coverage, libFuzzer uses *feature* coverage. Here, a feature combines the edge executed and a hit count [19]. This means two seeds s_i and s_j exercising the same edges will have $\mathbb{C}_{s_i} \neq \mathbb{C}_{s_j}$ if one seed executes an edge more often [21]. LibFuzzer's merge algorithm is similar to MINSET: seeds are sorted by size (with smaller seeds preferred) and the number of covered features (with higher hit counts preferred). Once sorted, seeds are greedily added to $\mathcal{C}$ whenever a new feature is found to be covered.

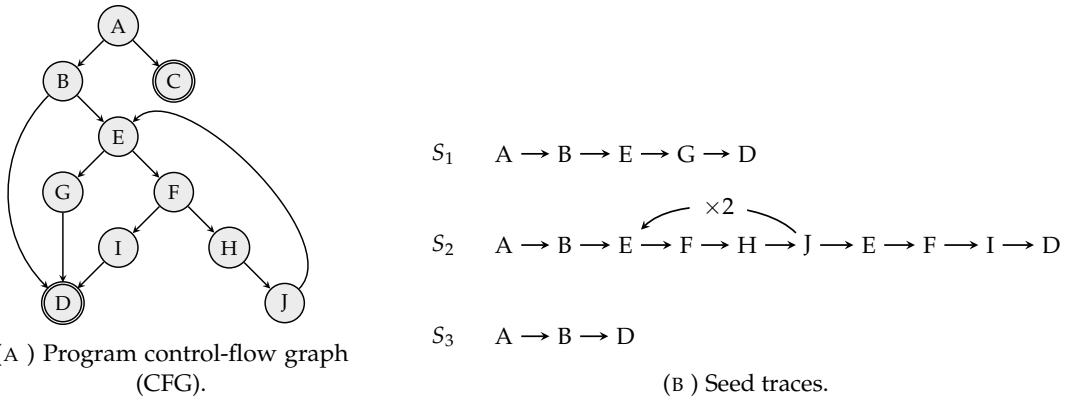
MoonShine. MoonShine [166] is a corpus minimization tool for operating system (OS) fuzzers. OS fuzzers typically test the system-call interface between the OS kernel and user-space applications. As such, the seeds minimized by MoonShine are a list of system calls gathered from program traces.

SmartSeed. SmartSeed [135] takes a different approach to those previously described. Rather than minimizing a corpus of seeds, SmartSeed instead uses a machine learning model to generate "valuable" seeds, where a seed is considered valuable if it uncovers new code or produces a crash.

`afl-cmin`. Due to AFL's popularity, `afl-cmin` [231] is perhaps the most widely-used corpus minimization tool. It implements a greedy minimization algorithm but has a unique approach to coverage. In particular, `afl-cmin` reuses AFL's implementation of edge coverage to categorize seeds at minimization time, recording an approximation of edge *frequency count*, not just whether the edge has been taken. Moreover, `afl-cmin` *bins* edge counts such that changes in edge frequency counts within a single bin are ignored, while transitions from one bin to another are "*flagged as an interesting change in program control flow*" [231]. When minimizing, `afl-cmin` chooses the smallest seed in the collection corpus that covers a given edge count and then performs a greedy, weighted minimization. We consider `afl-cmin` and MINSET as representatives of the state-of-the-art in corpus minimization tools and include both in our evaluation.

4.3.2 OPTIMIN

The corpus minimization techniques described in Section 4.3.1 all employ heuristic algorithms to approximate $\mathcal{C}$. This is because the underlying problem, the (W)MSCP, is NP-complete. However, in the case of corpus minimization, we found exact solutions were nonetheless computable in reasonable time. We achieved this by encoding the problem as a maximum satisfiability (MaxSAT) problem and using



(A) Program control-flow graph (CFG).

(B) Seed traces.

FIGURE 4.2: An example program and set of traces for illustrating corpus minimization. The program in Fig. 4.2a is executed with seeds S_1, S_2, and S_3, producing the traces in Fig. 4.2b .

an off-the-shelf MaxSAT solver. Thus, we implement OPTIMIN, an optimal corpus minimization tool for AFL.

OPTIMIN[3] uses the EvalMaxSat solver [10] to pose and solve corpus minimization as a MaxSAT problem. Unlike the boolean satisfiability (SAT) problem—which determines whether the variables in a given Boolean formula can be assigned values to make the formula evaluate to true—the MaxSAT problem divides constraints into hard and soft constraints with the aim to satisfy all hard constraints and maximize the total number (or weighted total) of satisfied soft constraints. OPTIMIN considers edge coverage as a hard constraint, while seed exclusion is considered a soft constraint. This approach ensures the solution $\mathcal{C}$ covers all edges with the minimal number of seeds, and is optimal in the sense that $\mathcal{C}$ is guaranteed to be exact (rather than an approximation).

We describe our approach using the example in Fig. 4.2. The program in Fig. 4.2a is executed with three seeds, producing the traces in Fig. 4.2b . To obtain $\mathcal{C}$, OPTIMIN:

1. Encodes each edge as a disjunction of seeds covering that edge. For example, the edge (A, B) is encoded as $S_1 \vee S_2 \vee S_3$ because it is covered by all seeds. Similarly, the edge (B, E) is encoded as $S_1 \vee S_2$ because it is only covered by seeds S_1 and S_2. Intuitively, this encoding tells the solver to select S_1 *or* S_2 to ensure the edge (B, E) is represented in $\mathcal{C}$.

2. Produces a conjunction of seed disjunctions. This conjunction is the *hard* constraint (thus assigned the maximum weight $\top$), guaranteeing all edges are covered in $\mathcal{C}$ (i.e., no coverage is "lost" during minimization).

[3]The "optimal minimizer". OPTIMIN is available at https://github.com/HexHive/fuzzing-seed-selection.

3. Encodes each seed S_i as a *soft* constraint with a given weight W_i. In a MaxSAT solver, minimization is performed via negation. Intuitively, this tells the solver to exclude a seed from $\mathcal{C}$. If the solver must choose between a set of seeds (to satisfy the hard constraint), then the one with a lower weight is preferred. For unweighted minimizations (i.e., when only the number of seeds matters, while their features—such as size—do not), $W_i = 1$. In the weighted case, W_i is S_i's file size (leading OPTIMIN to preference smaller seeds).

4. Encodes the hard and soft constraints in weighted conjunctive normal form (CNF) [12]. These constraints are passed to EvalMaxSat, which produces $\mathcal{C}$.

We present the complete set of constraints for this example in Table 4.2.

OPTIMIN is not the first tool to generate optimal solutions to minimization problems in software testing. For example, MINTS [98] and Nemo [126] both use integer linear programming (ILP) solvers to perform *test-suite minimization*; i.e., eliminate redundant test cases from a test suite "*based on any number of criteria [e.g., statement coverage, time-to-run, setup effort]*" [98]. When developing OPTIMIN, we explored the use of mixed integer programming solvers (which are more general than ILP solvers), but found them to be *orders-of-magnitude slower* than EvalMaxSat.

4.4 Evaluation

We perform a large-scale evaluation (33 CPU-yr of fuzzing) to understand the impact of seed selection on fuzzing's ultimate goal: finding bugs in real-world software. We aim to answer the following research questions:

RQ 1 How effective are corpus *minimization* tools at producing a minimal corpus? (Section 4.4.2)

RQ 2 What effect does seed selection have on a fuzzer's *bug finding* ability? Do fuzzers perform better when bootstrapped with (a) a small seed set (e.g., empty or singleton set), or (b) a large corpus of seeds, derived from an even larger collection corpus after applying a corpus minimization tool? (Section 4.4.3)

RQ 3 How does seed selection affect *state space exploration*? Does starting from a corpus that executes more instrumentation data points result in greater code coverage, or does a fuzzer's mutation engine naturally achieve the same coverage (e.g., when starting from an empty seed)? (Section 4.4.4)

We find that while corpus minimization greatly impacts fuzzing campaigns, the underlying minimization tool is less important, as long as *some* form of minimization

TABLE 4.2: OPTIMIN constraints for the example in Fig. 4.2. Edges are encoded as a disjunction of seeds covering that edge. The hard constraint—a conjunction of seed disjunctions—ensures all edges are covered in $\mathcal{C}$. Soft constraints—weighted by W_i—minimize the number of seeds selected.

(A) Weighted minimization (WOPT in Section 4.4.1).

Coverage.

Edges	Constraint
(A, B)	$S_1 \vee S_2 \vee S_3$
(B, E)	$S_1 \vee S_2$
(B, D)	S_3
(E, F), (F, H), (F, I), (H, J), (I, D), and (J, E)	S_2
(E, G), and (G, D)	S_1

Hard constraints.

Constraint	Weight
$(S_1 \vee S_2 \vee S_3) \wedge (S_1 \vee S_2) \wedge S_3 \wedge S_2 \wedge S_1$	$\top$

Soft constraints.

Constraint	Weight
$\neg S_1$	W_1
$\neg S_2$	W_2
$\neg S_3$	W_3

(B) Weighted minimization accounting for edge frequencies (WMOPT in Section 4.4.1). The loop backedge (J, E) is split into two edges (the number of times the back edge was executed).

Coverage.

Edges	Constraint
(A, B)	$S_1 \vee S_2 \vee S_3$
(B, E)	$S_1 \vee S_2$
(B, D)	S_3
(E, F), (F, H), (F, I), (H, J), (I, D), $(J, E)_1$, and $(J, E)_2$	S_2
(E, G), and (G, D)	S_1

Hard constraints.

Constraint	Weight
$(S_1 \vee S_2 \vee S_3) \wedge (S_1 \vee S_2) \wedge S_3 \wedge S_2 \wedge S_1$	$\top$

Soft constraints.

Constraint	Weight
$\neg S_1$	W_1
$\neg S_2$	W_2
$\neg S_3$	W_3

occurs. All experimental data is available at https://datacommons.anu.edu.au/DataCommons/rest/display/anudc:6106.

4.4.1 Methodology

Fuzzer Selection

Our real-world fuzzing campaigns use AFL (v2.52b) in greybox mode, while our Magma and FTS campaigns also include AFL++ with (a) CmpLog instrumentation enabled, and (b) a 250 KiB coverage map. We configure both fuzzers for single-system parallel execution with one main and one secondary node; the primary node focuses on deterministic checks, while the secondary node proceeds skips these checks and proceeds directly to random mutation.

For the FTS and the real-world targets we compile using AFL's LLVM [120] (v8) instrumentation for 32-bit `x86` and AddressSanitizer (ASAN) [187]. We chose LLVM instrumentation over AFL's assembler-based instrumentation because LLVM's offers the best level of interoperability with ASAN . We compile Magma targets using their default build configuration (i.e., 64-bit `x64` without ASAN).

We tune AFL's timeout and memory parameters for each target to enable effective fuzzing.[4] When fuzzing the FTS we configure the target process to respawn after *every* iteration (due to stability issues encountered when fuzzing in parallel execution mode). All other parameters are left at their default values.

Target Selection

We use targets from the Magma [84] and FTS [75] benchmarks, and six popular open-source programs (spanning 14 different file formats) to test different seed selection strategies. Table 4.3 summarizes these targets.

We exclude 14 of the 24 FTS targets, because: (i) they contain only memory leaks (e.g., *proj4-2017-08-14*), which are not detected by AFL by default; or (ii) we were unable to find a suitably-large collection corpus for a particular file type (e.g., ICC files for *lcms-2017-03-21*). Similarly, two Magma targets were excluded (*openssl* and *sqlite3*) because we were unable to find a suitably large corpus, leaving us with five targets. The six real-world targets are popular programs that are both (a) commonly fuzzed, and (b) operate on a diverse range of file formats (e.g., images, audio, documents).

[4] Per-target settings are available at https://datacommons.anu.edu.au/DataCommons/rest/display/anudc:6106.

TABLE 4.3: Fuzzing targets. The FTS provides drivers.

	Target	Driver	Command-line	Version	File type
Magma	*libpng*	`libpng_read_fuzzer`			PNG
	libtiff	`tiff_read_rgba_fuzzer`			TIFF
	libxml2	`libxml2_xml_reader_for_file_fuzzer`			XML
	php-exif	`exif`			JPEG
	php-json	`json`			JSON
	php-parser	`parser`			PHP
	poppler	`pdf_fuzzer`			PDF
FTS	*freetype2*			2017	TTF
	guetzli			2017-3-30	JPEG
	json			2017-02-12	JSON
	libarchive			2017-01-04	GZIP
	libjpeg-turbo			07-2017	JPEG
	libpng			1.2.56	PNG
	libxml2			2.9.2	XML
	pcre2			10.00	Regex
	re2			2014-12-09	Regex
	vorbis			2017-12-11	OGG
Real-world	*freetype2*	`char2svg`	`@@ @`	2.5.3	TTF
	librsvg	`rsvg-convert`	`-o /dev/null @@`	2.40.20	SVG
	libtiff	`tiff2pdf`	`-o /dev/null @@`	4.0.9	TIFF
	libxml2	`xmllint`	`-o /dev/null @@`	2.9.0	XML
	poppler	`pdftotext`	`@@ /dev/null`	0.64.0	PDF
	sox-mp3	`sox`	`-single-threaded @@ -b 16 -t aiff /dev/null channels 1 rate 16k fade 3 norm`	14.4.2	MP3
	sox-wav	`sox`	`-single-threaded @@ -b 16 -t aiff /dev/null channels 1 rate 16k fade 3 norm`	14.4.2	WAV

Sample Collection

For each file type in Section 4.4.1, we built a Web crawler using Scrapy [241] to crawl the Internet for 72 h to create the collection corpus. For image files, crawling started with Google search results and the Wikimedia Commons repository. For media and document files, crawling started from the Internet Archive and Creative Commons collections. We used the regular expressions from RegExLib [178], and sourced OGG files from old video games [112, 148, 161] (in addition to the Internet Archive). We sourced PHP files from test suites for popular PHP interpreters (e.g., Facebook's HipHop Virtual Machine [58]) and from popular GitHub repositories (e.g., WordPress [221]). Finally, we found TIFF files to be relatively rare, so we generated 40 % of the TIFF seeds by converting other image types, such as JPEG and BMP, using ImageMagick (v6.9.7).

We preprocessed each collection corpus to remove (a) duplicates identified by MD5 checksum, and (b) files larger than 300 KiB. The cutoff file size of 300 KiB is our best effort to conform to the AFL authors' suggestions regarding seed size, while still having enough eligible seeds in the preprocessed corpora. We split audio files larger than 1 MiB into smaller files using FFmpeg (v3.4.8). In total, we collected 2,899,208 seeds across 14 different file formats. After preprocessing our collection corpus, we were left with a total of 1,019,688 seeds.

Experimental Setup

We run the Magma experiments on a cluster of AWS EC2 machines with 36-core Intel® Xeon® E5-2666 v3 2.9 GHz CPUs and 60 GiB of RAM. We conduct the FTS and real-world experiments on a Dell PowerEdge server with a 48-core Intel® Xeon® Gold 5118 2.30 GHz CPU, 512 GiB of RAM, and running Ubuntu 18.04.

Following Section 2.6, we run one fuzzing campaign per {program × fuzzer × initial corpus} combination (where a *program* is an executable driver for a particular *target*). Each fuzzing campaign consists of thirty independent 18 h trials.

Experiment

We evaluate the following six seed selection strategies against the previously-described targets and fuzzers:

FULL The collection corpus without minimization, preprocessed to remove duplicates and filtered for size (per Section 4.4.1).

EMPTY A per-target corpus comprising a single "empty" seed. Per Klees et al. [115], "*the empty seed should be considered [when evaluating fuzzers], despite its use contravening conventional wisdom.*" This seed is a zero-length file for six file types (JSON, MP3, REGEX, TIFF, TTF, and XML). For the remaining eight file types, the seed is a small file handcrafted to contain the bytes necessary to satisfy file header checks. This follows from the `readelf` experiments in Section 4.2.2, which demonstrates how poorly AFL performs when these header checks are not satisfied by the initial corpus. These files range in size from 11 B (SVG) to 13 KiB (OGG), with a median size of 51 B.

PROV The corpus provided with the benchmark (if any). This strategy is only applicable for the two fuzzer benchmarks (Magma and FTS).

MSET The corpus obtained using the UNWEIGHTED MINSET tool. We present UNWEIGHTED MINSET (rather than TIME or SIZE MINSET) because it finds more bugs than other MINSET configurations [177].

CMIN The corpus produced using AFL's `afl-cmin` tool.

WOPT The *optimal minimum* corpus weighted by *file size*.

WMOPT The *weighted optimal minimum* corpus that takes into account edge frequencies. WMOPT attempts to *minimize* file sizes while *maximizing* an edge's frequency count. We tried to implement an "optimal `afl-cmin`" (i.e., minimizing file size while treating the same edge with different hit counts as distinct

constraints), but EvalMaxSat was unable to find a solution (after 6 h) for many targets. Maximizing the total hit count for a given edge is a compromise and one that we hypothesize results in *deeper* state space exploration.

We exclude REDQUEEN's uninformed, generic seed due to its poor performance in Section 4.2.2. We also explored an unweighted optimal minimum corpus, but found that EvalMaxSat produced the same corpora as WOPT for all but three targets (Magma's *libpng*, the FTS' *libarchive*, and the real-world *poppler* target). Thus, we also exclude unweighted minimal corpora from our results.

We compare the performance of each seed selection strategy across three measures:

Bug count. The ultimate aim of fuzzing is to find bugs in software. Thus, we use a direct bug count to answer RQ 2 and compare fuzzer effectiveness. We perform a manual triage of *all* crashes in the real-world target set. This is in contrast to many prior works [115, 122, 177, 214], which uses stack-hash deduplication to determine unique bugs from crashes—a technique known to both over- *and* under-count bugs [84, 108, 115].

Bug survival. We use survival analysis and the log-rank test as described in Section 2.6.2. When computing the restricted mean survival time (RMST) we use $N = 30$ repeated trials and an upper bound $T = 18\,\text{h}$. The log-rank test is computed under the null hypothesis that two corpora share the same survival function. Thus, we consider two corpora to have statistically-equivalent bug survival times if the log-rank test's p-value > 0.05.

Code coverage. Coverage is often used to measure fuzzing effectiveness [23, 115, 144]. Like FUZZBENCH [144], we use Clang's source-based region coverage metric to answer RQ 3 and compare coverage across fuzzers using different coverage maps. This is achieved by replaying each trial's fuzzing queue through a version of the target compiled with Clang's source-based coverage [209].

We report both the mean and 95 % bootstrap confidence interval (CI) of the percentage of regions executed across each campaign. Per Section 2.6.3, region coverage is compared across corpora using the Mann-Whitney U-test; a p-value > 0.05 implies a {program $\times$ fuzzer $\times$ corpus} combination yields a statistically-equivalent result compared to another.

TABLE 4.4: Corpora sizes. Each corpus is summarized by the number of files contained within ("#") and total size ("S", in MiB unless stated otherwise). The smallest *minimized* corpus for both "#" and "S" are highlighted in green and blue, respectively.

		FULL		PROV		MSET		CMIN		WOPT		WMOPT	
	Target	#	S	#	S	#	S	#	S	#	S	#	S
Magma	libpng	66,512	7,773.60	4	1.22 KiB	36	2.69	172	9.61	33	2.01	51	5.37
	libtiff	99,955	446.63	21	0.20	35	0.14	115	0.39	33	0.13	55	0.23
	libxml2	79,032	205.64	1,268	3.90	42	0.38	132	0.82	42	0.40	56	0.51
	php-exif	120,000	222.86	60	0.26	2	0.01	2	0.01	2	0.01	2	0.01
	php-json	19,978	76.46	55	4.23 KiB	17	0.72	212	4.29	17	0.85	30	1.91
	php-parser	75,777	224.51	2,934	0.92	606	1.94	2,229	11.43	569	1.71	1,187	7.86
	poppler	99,986	6,085.07	392	18.39	237	28.11	2,273	119.27	222	26.70	510	66.89
Google FTS	freetype2	466	35.50	2	2.83 KiB	43	5.40	246	20.92	37	5.04	53	6.95
	guetzli	120,000	222.86	2	544 B	17	0.04	463	0.60	13	0.03	51	0.10
	json	19,978	76.46	1	14 B	17	0.95	149	2.56	16	1.21	27	1.77
	libarchive	108,558	850.64	1	500 B	41	1.05	180	2.80	40	0.94	57	1.52
	libjpeg-turbo	120,000	222.86	1	413 B	3	0.01	93	0.11	3	0.01	13	0.02
	libpng	66,512	7,773.60	1	1.23 KiB	22	1.91	107	4.05	19	1.71	28	2.85
	libxml2	79,032	205.64	0	–	97	2.23	440	7.71	89	1.40	175	4.50
	pcre2	4,520	0.46	0	–	183	0.04	691	0.13	175	0.03	321	0.09
	re2	4,520	0.46	0	–	56	0.01	155	0.01	55	0.01	84	0.01
	vorbis	99,450	8,902.70	1	2.54 KiB	8	0.33	237	12.06	8	0.27	20	1.88
Real-world	freetype2	466	35.50	–	–	23	3.04	73	8.68	23	3.02	33	4.70
	librsvg	71,763	744.59	–	–	173	4.34	881	17.05	159	3.80	333	10.47
	libtiff	99,955	446.63	–	–	23	0.10	67	0.27	23	0.10	33	0.14
	libxml2	79,032	205.64	–	–	103	1.67	505	9.04	95	1.60	196	6.70
	poppler	99,986	6,085.07	–	–	189	22.70	1,318	121.90	177	22.04	381	50.30
	sox-mp3	99,691	4,094.22	–	–	9	0.17	137	3.75	6	0.30	15	0.64
	sox-wav	74,000	2,490.61	–	–	10	0.39	68	1.65	9	0.27	14	0.49

4.4.2 Minimization (RQ 1)

Table 4.4 shows the sizes of 14 collection corpora minimized across 21 target programs based on the code coverage measured by AFL. We reapplied the four corpus minimizers when fuzzing Magma with AFL++, as AFL++'s larger coverage map (250 KiB, compared to AFL's 64 KiB) theoretically results in a more fine-grained coverage view. Indeed, we saw small variations (up to 10 %) between the AFL and AFL++ minimized corpora, due to both the different-sized coverage maps and hash collisions inherent to AFL's method for computing edges.

Across both fuzzers, corpora produced by CMIN are significantly larger than that produced by MSET (mean 8× larger), WOPT (mean 9× larger), and WMOPT (mean 4× larger). We attribute this to CMIN distinguishing seeds with different edge frequency counts: MSET and WOPT only look at edges executed, ignoring the number of times these edges are executed, while WMOPT maximizes an edge's frequency count. In comparison, MSET was only at most 37 seeds larger than WOPT (*php*'s `parser`), and on average only five seeds larger than WOPT. WMOPT corpora were (on average) twice as large as WOPT corpora. Finally, the minimized *php*'s `exif` corpora are notable because they discard 99 % of the full JPEG corpus, due to a lack of diverse EXIF data. The small minimized `exif` corpora demonstrate the importance of selecting a

diverse range of initial inputs and minimizing large corpora.

In addition to generating smaller corpora, W[M]OPT incur lower run-time costs during minimization (compared to CMIN). We exclude the time required to trace the target and collect coverage data for each seed. WOPT's minimization times range from 12 ms (*freetype2*) to 23 min (*libjpeg-turbo*), with a mean time of 141 s. WMOPT takes a similar amount of time: between 31 ms (*freetype2*) and 24 min (*php*'s `exif`), with a mean time of 143 s. In comparison, CMIN's minimization times range from 12 s to 130 min, with a mean time of 25 min. Despite CMIN's significantly slower minimization times, recall that (a) `afl-cmin` is a BASH script, while we wrote our optimal solver in C++, and (b) corpus minimization is a one-time upfront cost.

What ultimately matters is if the minimized corpora lead to better fuzzing outcomes. To this end, the following section discusses the bug-finding ability of the different corpus minimization techniques across our three benchmark suites (Magma, FTS, and a set of real-world targets) and two fuzzers (AFL and AFL++). We analyze these results with respect to the performance measures outlined in Section 4.4.1.

Finding

OPTIMIN produces significantly smaller corpora compared to existing state-of-the-art corpus minimization tools, while also incurring lower run-time costs.

4.4.3 Bug Finding (RQ 2)

Table 4.5 summarizes the bugs found in our Magma and FTS campaigns. For our real-world campaigns, we summarize the seven Common Vulnerabilities and Exposures (CVEs) assigned to us (for bugs found in these campaigns) in Table 4.6. In total, 77 (25 Magma, 15 FTS, and 33 real-world) bugs were found.

FTS Coverage Benchmarks

While is section's main focus is on the bug-finding ability of each corpus, we first discuss six FTS "bugs" that are not actual bugs, but are instead code locations that the fuzzer must reach (marked with † in Table 4.5b). Notably, two of these locations are reached instantaneously (i.e., seeds in the fuzzing corpora reach the particular line of code without requiring any fuzzing) by most corpora *except* EMPTY. Naturally, EMPTY takes some time to reach the target locations, as AFL must construct valid inputs from "nothing". Nevertheless, EMPTY reaches four of the six target locations within two hours (on average). A *libpng* location and the *freetype2* locations are never reached by EMPTY, because: (i) *freetype2* requires a valid composite glyph, which

TABLE 4.5: Bug-finding results, presented as the RMST in hours with 95 % bootstrap CI. Bugs never found have an RMST of ⊤ (to distinguish bugs with an RMST of 18 h). The best performing corpus (corpora if the bug survival times are statistically equivalent per the log-rank test) for each target is highlighted in green (smaller is better).

(A) Magma bugs found by AFL and AFL++. We only report the RMST for bugs *triggered*. Bugs not triggered by any corpus are omitted (irrespective of whether the bug was reached or not). The *php* `json` and `parser` targets are omitted because no bugs were found.

Target	Bug	FULL		EMPTY		PROV		MSET		CMIN		WOPT		WMOPT	
		AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++
libpng	PNG001	⊤	⊤	16.83 ± 4.30	⊤	16.96 ± 3.82	⊤	17.95 ± 0.33	⊤	⊤	⊤	17.54 ± 2.83	⊤	17.47 ± 3.25	⊤
	PNG003	1.93 ± 0.07	8.67 ± 3.45	⊤	0.12 ± 0.05	0.0042 ± 0.01	0.0050 ± 0.01	0.01 ± 0.01	0.01 ± 0.01	0.02 ± 0.01	0.10 ± 0.08	0.01 ± 0.01	0.04 ± 0.01	0.01 ± 0.01	0.34 ± 0.13
	PNG006	⊤	⊤	⊤	17.96 ± 0.26	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	PNG007	17.04 ± 3.53	⊤	⊤	17.41 ± 3.61	6.85 ± 2.53	17.56 ± 2.69	11.03 ± 3.46	⊤	12.26 ± 3.68	⊤	10.65 ± 3.26	⊤	10.04 ± 3.54	⊤
libtiff	TIF001	17.77 ± 0.96	⊤	⊤	⊤	⊤	⊤	15.98 ± 3.23	⊤	15.81 ± 2.85	⊤	13.93 ± 3.64	⊤	15.20 ± 2.55	⊤
	TIF002	16.85 ± 2.97	16.48 ± 4.03	17.65 ± 1.17	⊤	15.29 ± 2.77	17.21 ± 2.93	16.39 ± 2.73	17.20 ± 3.10	12.81 ± 2.73	17.46 ± 3.27	14.09 ± 3.32	16.40 ± 4.25	14.50 ± 2.84	⊤
	TIF007	0.34 ± 0.07	7.33 ± 4.14	1.27 ± 0.66	0.42 ± 0.25	0.05 ± 0.02	0.05 ± 0.03	0.03 ± 0.01	0.01 ± 0.01	0.03 ± 0.01	0.13 ± 0.01	0.03 ± 0.01	0.01 ± 0.01	0.04 ± 0.01	0.02 ± 0.01
	TIF008	16.96 ± 1.07	⊤	17.02 ± 2.94	⊤	16.23 ± 2.81	⊤	12.53 ± 2.23	13.36 ± 3.85	7.91 ± 2.24	17.43 ± 3.50	8.68 ± 2.29	⊤	9.39 ± 2.18	⊤
	TIF012	3.64 ± 1.81	12.24 ± 3.66	2.45 ± 1.24	9.04 ± 3.93	0.47 ± 0.17	7.65 ± 2.79	0.65 ± 0.21	7.84 ± 3.03	0.53 ± 0.12	9.85 ± 3.31	0.68 ± 0.18	9.33 ± 3.04	0.70 ± 0.14	9.83 ± 3.11
	TIF014	0.76 ± 0.15	11.34 ± 3.97	1.87 ± 1.19	0.73 ± 0.55	0.88 ± 0.36	6.46 ± 3.23	1.38 ± 0.78	3.08 ± 1.68	1.56 ± 0.72	5.16 ± 2.31	1.45 ± 0.35	2.60 ± 1.16	1.66 ± 0.51	1.39 ± 1.13
libxml2	XML001	17.87 ± 0.82	⊤	⊤	⊤	17.42 ± 2.49	⊤	⊤	⊤	17.25 ± 2.10	⊤	⊤	⊤	⊤	⊤
	XML003	8.69 ± 2.00	14.94 ± 2.89	⊤	⊤	14.93 ± 4.32	14.74 ± 3.69	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	XML004	0.69 ± 0.12	0.85 ± 0.08	0.01 ± 0.01	0.02 ± 0.01	0.05 ± 0.01	0.02 ± 0.01	0.77 ± 0.30	0.08 ± 0.05	0.61 ± 0.29	0.15 ± 0.08	1.21 ± 0.41	0.77 ± 0.26	0.63 ± 0.27	0.80 ± 0.34
	XML009	15.99 ± 3.58	16.42 ± 3.26	17.94 ± 0.36	15.00 ± 2.75	16.44 ± 3.64	17.09 ± 2.93	17.55 ± 1.69	17.50 ± 2.19	15.33 ± 3.22	17.07 ± 2.58	15.51 ± 3.55	16.91 ± 2.39	14.31 ± 3.16	16.89 ± 2.01
	XML017	0.63 ± 0.12	1.32 ± 0.11	16.88 ± 2.66	2.60 ± 1.22	0.06 ± 0.02	0.38 ± 0.15	0.09 ± 0.01	0.09 ± 0.01	0.11 ± 0.03	0.07 ± 0.01	0.11 ± 0.01	0.07 ± 0.01	0.09 ± 0.01	0.08 ± 0.01
php-exif	PHP004	⊤	⊤	⊤	⊤	0.00 ± 0.01	16.20 ± 5.28	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	PHP009	⊤	⊤	⊤	⊤	0.00 ± 0.01	0.00 ± 0.01	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	PHP011	11.79 ± 2.09	7.76 ± 3.78	8.68 ± 2.32	0.05 ± 0.01	0.00 ± 0.01	13.74 ± 2.40	9.66 ± 2.67	0.05 ± 0.01	7.40 ± 2.16	0.06 ± 0.01	8.43 ± 1.97	0.06 ± 0.02	9.76 ± 2.77	0.05 ± 0.01
poppler	PDF002	⊤	⊤	⊤	⊤	17.43 ± 3.45	⊤	17.86 ± 0.82	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	PDF005	⊤	⊤	⊤	⊤	⊤	⊤	17.46 ± 3.28	⊤	17.18 ± 3.01	⊤	16.93 ± 2.92	⊤	15.40 ± 3.38	⊤
	PDF010	⊤	⊤	⊤	⊤	0.83 ± 0.64	7.66 ± 4.15	4.06 ± 0.74	17.42 ± 3.54	5.54 ± 0.62	17.65 ± 1.13	4.51 ± 0.48	⊤	8.07 ± 1.49	⊤
	PDF016	⊤	⊤	0.0042 ± 0.01	0.0066 ± 0.01	0.08 ± 0.01	0.09 ± 0.02	0.14 ± 0.01	2.19 ± 0.14	1.11 ± 0.04	3.13 ± 0.17	0.14 ± 0.01	2.48 ± 0.17	0.30 ± 0.01	2.51 ± 0.16
	PDF018	⊤	⊤	⊤	⊤	6.17 ± 2.62	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	PDF021	⊤	⊤	⊤	⊤	⊤	⊤	17.61 ± 2.37	⊤	17.67 ± 2.00	⊤	⊤	⊤	⊤	⊤
	PDF022	⊤	⊤	⊤	⊤	⊤	⊤	17.79 ± 0.57	⊤	⊤	⊤	17.90 ± 0.48	⊤	⊤	⊤

(B) Bug-finding results (cont.). FTS bugs found by AFL and AFL++. IDs are derived from the order in which the bugs are presented in the target's `README` (from the FTS repo). Bugs marked with † denote benchmarks that attempt to verify the fuzzer can reach a known location. Results with "–" indicate the FTS does not contain seeds for that target (see Table 4.4), and so we ignore it. The *vorbis* target is omitted because none of its three bugs were found.

Target	Bug	FULL		EMPTY		PROV		MSET		CMIN		WOPT		WMOPT	
		AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++
`freetype2`	A†	0.00 ± 0.00	0.00 ± 0.00	⊤	⊤	6.78 ± 2.12	6.10 ± 2.73	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
`guetzli`	A	⊤	⊤	⊤	⊤	10.25 ± 2.33	17.48 ± 1.38	15.88 ± 2.09	17.45 ± 3.37	17.78 ± 1.07	⊤	15.45 ± 2.82	⊤	16.92 ± 2.83	⊤
`json`	A	⊤	⊤	⊤	16.41 ± 4.24	0.09 ± 0.08	0.28 ± 0.27	17.64 ± 2.17	⊤	16.93 ± 3.92	17.57 ± 2.60	17.68 ± 1.43	⊤	17.90 ± 0.62	⊤
`libarchive`	A	⊤	⊤	⊤	15.31 ± 1.76	⊤	9.08 ± 0.90	4.44 ± 0.21	11.86 ± 1.74	9.39 ± 1.42	11.97 ± 1.90	12.50 ± 0.50	6.78 ± 0.44	7.38 ± 0.05	13.50 ± 1.75
`libjpeg-turbo`	A†	17.19 ± 2.37	⊤	1.92 ± 0.45	10.43 ± 2.73	3.00 ± 0.95	14.14 ± 2.47	3.36 ± 0.98	16.57 ± 2.70	3.82 ± 1.19	15.62 ± 2.75	4.71 ± 1.52	16.91 ± 2.70	3.68 ± 0.93	15.70 ± 2.11
`libpng`	A†	2.41 ± 0.01	0.0043 ± 0.002,0	0.03 ± 0.01	0.11 ± 0.02	0.08 ± 0.08	0.21 ± 0.03	0.0051 ± 0.002,9	0.09 ± 0.01	0.0072 ± 0.003,6	0.0041 ± 0.002,7	0.0025 ± 0.001,7	0.0049 ± 0.003,2	0.0038 ± 0.002,5	0.08 ± 0.01
	B†	0.00 ± 0.00	0.00 ± 0.00	⊤	0.33 ± 0.04	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00	0.00 ± 0.00
	C†	2.42 ± 0.02	0.0089 ± 0.009,2	0.0003 ± 0.000,4	0.07 ± 0.14	0.0008 ± 0.000,7	2.47 ± 0.85	0.0046 ± 0.002,9	0.0049 ± 0.002,5	0.0070 ± 0.003,5	0.0054 ± 0.003,3	0.0023 ± 0.001,7	0.0062 ± 0.003,4	0.0036 ± 0.002,2	0.0040 ± 0.003,2
`libxml2`	A	4.46 ± 0.32	17.84 ± 0.58	⊤	16.46 ± 2.48	–	–	0.54 ± 0.06	3.10 ± 0.09	0.79 ± 0.12	4.72 ± 0.96	0.66 ± 0.07	4.70 ± 0.51	0.85 ± 0.23	7.81 ± 2.23
	B	16.46 ± 1.27	⊤	5.32 ± 1.22	13.43 ± 2.26	–	–	13.35 ± 1.95	⊤	10.57 ± 2.20	17.85 ± 0.91	8.93 ± 1.81	17.95 ± 0.28	14.46 ± 2.09	⊤
	C	⊤	⊤	⊤	⊤	–	–	⊤	⊤	17.53 ± 2.89	⊤	⊤	⊤	⊤	⊤
`pcre2`	A	2.57 ± 0.39	4.56 ± 0.36	1.88 ± 0.25	2.70 ± 0.44	–	–	2.15 ± 0.24	5.43 ± 0.74	2.07 ± 0.23	3.10 ± 0.46	1.59 ± 0.16	4.83 ± 0.47	1.39 ± 0.18	3.34 ± 0.44
	B	2.04 ± 0.73	5.34 ± 1.58	3.25 ± 0.70	5.71 ± 0.88	–	–	2.01 ± 0.68	3.83 ± 0.73	2.29 ± 0.65	3.98 ± 1.36	1.42 ± 0.42	3.47 ± 0.96	1.61 ± 0.72	3.39 ± 1.22
`re2`	A†	0.82 ± 0.16	2.91 ± 0.52	1.72 ± 0.31	9.72 ± 1.61	–	–	0.74 ± 0.52	3.30 ± 0.53	0.52 ± 0.12	3.85 ± 0.70	0.50 ± 0.16	7.52 ± 2.88	0.42 ± 0.09	3.35 ± 1.14
	B	16.20 ± 3.83	16.41 ± 2.49	17.66 ± 1.48	17.69 ± 1.87	–	–	11.52 ± 2.70	15.69 ± 2.53	11.85 ± 3.22	16.97 ± 2.35	12.19 ± 3.07	17.08 ± 2.62	12.36 ± 3.18	15.91 ± 2.55

TABLE 4.6: Real-world targets assigned a CVE.

Target	CVE	Description
`libtiff`	2019-14973	Elision of integer overflow check by compiler
`poppler`	2019-12293	Heap buffer overread
`sox`	2019-8354	Integer overflow causes improper heap allocation
	2019-8355	Integer overflow causes improper heap allocation
	2019-8356	Stack buffer bounds violation
	2019-8357	Integer overflow causes failed memory allocation
	2019-13590	Integer overflow causes failed memory allocation

EMPTY never synthesizes in the given timeframe, and (ii) *libpng* requires a specific chunk type (sRGB), which is difficult to synthesize without any knowledge of the PNG file format.

The only coverage benchmark that is not reached within minutes, *libjpeg-turbo*, is reliably reached by all corpora *except* FULL within the first five hours (on average) of each trial. The FULL corpus is highly unreliable on this target: it only reaches the target location in 10 % of trials, and when it does, it takes double the time of the other corpora. This results in a high survival time of 17.19 h.

The EMPTY Seed

Of the 41 (14 Magma, 8 FTS, and 19 real-world) bugs EMPTY was able to find, it was the (equal) fastest to do so for 21 of these (5 Magma, 3 FTS, and 13 real-world). This result is particularly striking on *sox* (both MP3 and WAV), where EMPTY found the most bugs with the lowest RMSTs (including three of the CVEs in Table 4.6). However, EMPTY also suffers from the highest "false negative" rate: it is the most likely corpus to miss a bug when one exists (as evident from the number of $\top$ entries in Table 4.5; the most of any corpus).

We hypothesize this low RMST is due to the reduced search space when mutating the empty seed, but that the mutation engine is less likely to "get lucky" in generating a bug-inducing input when starting from nothing. Indeed, for the three FTS bugs where EMPTY statistically outperforms the other corpora (*libjpeg-turbo*, *libpng*'s bug C, and *libxml2*'s bug B), EMPTY finds the bug with the lowest number of mutations (on average, half the number of mutations compared to the other corpora on these three bugs) while also achieving a comparable (and sometimes slightly slower) iteration rate than the other corpora (in particular, WOPT achieves a higher iteration rate than EMPTY on these three targets).

PROVided Seeds

The FTS PROV seeds—half of which are *singleton seeds*—are selected (by the FTS developers) based on their ability to trigger the target bug(s) within a few hours. For example, the FTS *json* bug is "*usually found in about 5 minutes using the provided seed*" [75] (confirmed by our results). While serving as a useful regression test (i.e., ensuring a new fuzzer does not perform worse than an existing one), this is not indicative of typical fuzzing practices, as (a) the location of bugs is unknown *a priori*, and (b) large seed sets are used in practice (per Section 4.2.3). Given the former, it is notable that the minimized corpora (CMIN, MSET, WOPT, and WMOPT) also

successfully found the same FTS bugs found by PROV, and even outperform the PROV corpus in half of these targets (*freetype2*, *libarchive*, and *libpng*).

The PROV corpus is the best performer at finding bugs in Magma: it triggers the most bugs—21 of the 25 bugs found by all fuzzers—and achieves the (equal) lowest RMSTfor 15 of these bugs. Similarly to FTS's *freetype2* and *libpng*, all three *php* `exif` bugs were found without any mutation of the PROV seeds. Closer inspection of this corpus reveals why: PROV contains images that serve as regression tests for each of these three bugs (`bug77753.tiff`, `bug77563.jpg`, `bug77950.jpg`, corresponding to bugs PHP004, PHP009, and PHP011, respectively). These regression tests immediately trigger their respective bugs, but with Magma's *ideal sanitization*—"*in which triggering a bug immediately results in a crash*" [84]—disabled by default, these seeds do not cause a crash and hence are not excluded by AFL.[5]

Iteration Rates

Low iteration rates coupled with large corpora have a detrimental effect on a fuzzer's ability to find bugs. For example, with FULL, *guetzli* achieves mean iteration rates of 0.84 and 0.74 execs/s for AFL and AFL++, respectively. At the other end of the spectrum, EMPTY achieves mean iteration rates of 229.23 and 167 execs/s (for AFL and AFL++, respectively), while the minimized corpora achieve iteration rates between 2 and 5 execs/s. FULL's low iteration rate has a severe impact: both AFL and AFL++ fail to complete an initial pass over the 120,000 seeds in this corpus (in an 18 h trial), let alone perform any mutations and discover the bug. In comparison, the *guetzli* bug is found by all minimized corpora (CMIN, MSET, WOPT, and WMOPT) and PROV. We encounter similar results with *poppler*, where again neither AFL nor AFL++ complete a full pass over the collection corpus (resulting in no bugs triggered).

We find iteration rates vary significantly between fuzzers. For example, fuzzing Magma's *libpng* with AFL and EMPTY achieves a mean iteration rate of 693 execs/s, while AFL++ achieves a mean iteration rate of 2,508 execs/s. Conversely, fuzzing the same target with AFL and WOPT achieves a mean iteration rate of 4,575 execs/s, compared to AFL++ at 351 execs/s. These results correlate with the bug survival times in Table 4.5a (where AFL++ outperforms AFL on the EMPTY seed, while AFL outperforms AFL++ with the WOPT corpus), suggesting higher iteration rates contribute to a fuzzer's bug-finding ability (and at the very least, allow a fuzzer to more-quickly discard inputs that are not worth exploring).

[5] At the time of writing, this is a known issue flagged by the Magma developers, per https://github.com/HexHive/magma/issues/54.

When fuzzing with non-empty corpora, AFL achieves a higher iteration rate than AFL++. This is likely due to a combination of (a) more complex target instrumentation (where more of this instrumentation is being exercised with valid inputs), and (b) a coverage map $\sim 4\times$ larger than AFL's (which has L2 cache implications). These results further reinforce the need for corpus minimization when starting with a large collection corpus, particularly as fuzzer complexity increases.

Comparison to Previous Magma Evaluations

Our results improve on those originally presented by Hazimeh, Herrera, and Payer [84]. Nine of the bugs triggered during our 18 h trials (PNG006, TIF001, XML001, XML003, PDF002, PDF005, PDF018, PDF021, and PDF022) were *never* triggered by AFL/AFL++ in the original 24 h campaigns. Furthermore, two of these bugs (*poppler*'s PDF005 and PDF022) were never found by *any* of the seven fuzzers originally evaluated by Hazimeh, Herrera, and Payer [84]. These two bugs were only found by the distilled corpora fuzzed with AFL and never with FULL, EMPTY, PROV, or AFL++. This is significant because over 200,000 CPU-hr were spent fuzzing Magma targets (across many 24 h and 7 d trials).

CVEs

Our real-world fuzzing campaigns led to the assignment of seven CVEs across three targets, summarized in Table 4.6 (our campaigns uncovered another 26 bugs, but these were already under investigation). Of these bugs, *libtiff*'s CVE-2019-14973 is particularly interesting: discovered by all corpora, but found the fastest and most reliably by EMPTY (with an RMST of 6.02 h), this bug is only evident because we build our real-world targets for 32-bit `x86`. The *libtiff* maintainers report that the undefined behavior at the root of this bug does not manifest on 64-bit targets.

Our campaigns also reproduced an uncontrolled resource consumption bug in *libtiff* (CVE-2018-5784). This bug is caused by an infinite loop in the TIFF image directory linked list and is again found most frequently by EMPTY (11 out of 30 trials, resulting in an RMST of 12.37 h). In comparison, this bug is never found by FULL and MSET and only once by the other corpora. Notably, the initial EMPTY seed does not contain any image file directories, while all of the TIFF files in our minimized corpora do. We suspect AFL's mutations break existing directory structures (leading to parser failures), whereas EMPTY can construct a (malformed) directory list from scratch. These mutations eventually lead to a loop in the list, causing uncontrolled resource consumption.

Finding

Seed selection significantly impacts a fuzzer's ability to find bugs. Both AFL and AFL++ perform better when bootstrapped with a minimized corpus, although the exact minimization tool is inconsequential. While both AFL and AFL++ find a similar number of bugs, AFL is generally faster to do so (and with less variance in bug-finding times).

4.4.4 Code Coverage (RQ 3)

Table 4.7 summarizes the coverage achieved overall Magma and FTS campaigns. For most targets, EMPTY explores less code than the other corpora: on average, half as much code (although this difference slightly decreases when fuzzing with AFL++). In particular, on targets accepting highly-structured input formats (e.g., *libxml2*, *poppler*), EMPTY explores less than half as much of the program's code compared to the four minimized corpora. EMPTY's results slightly improve when fuzzing with AFL++, likely due to the additional CmpLog instrumentation (reflecting our `readelf` results in Section 4.2.2). However, despite this improvement, EMPTY's performance remains inferior to any other seed selection strategies.

After an 18 h trial, little distinguishes the code coverage achieved by the non-EMPTY corpora, and once again the four minimized corpora with AFL produced the best results. Curiously, the coverage gains AFL++ saw when fuzzing `readelf` with CMIN (Section 4.2.2) did not manifest in any of the five Magma targets: both AFL and AFL++ achieved similar levels of code coverage.

Finding

Seed selection has a significant impact on a fuzzer's ability to expand code coverage. When fuzzing with the empty seed, more advanced fuzzers (such as AFL++) can cover more code. However, this advantage all but disappears when bootstrapping the fuzzer with a minimized corpus, as faster iteration rates become more critical. The exact minimization tool remains inconsequential.

4.4.5 Discussion

Selecting a corpus minimization tool. We evaluated three corpus minimization tools: MINSET, `afl-cmin`, and our OPTIMIN. Our results do not reveal a "best" minimization tool; while minimized corpora sizes varied markedly between tools, stochastic fuzzing variability means this ultimately has no statistically significant

TABLE 4.7: Code coverage, expressed as the mean region coverage (as a percentage of total program regions) with 95 % bootstrap CI. The best performing corpus (corpora if the code coverages are statistically equivalent per the Mann-Whitney *U*-test) for each target is highlighted in green (larger is better).

(A) Magma code coverage with AFL and AFL++.

	FULL		EMPTY		PROV		MSET		CMIN		WOPT		WMOPT	
Target	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++
libpng	28.71 ± 0.09	25.56 ± 1.51	15.26 ± 0.06	19.10 ± 0.97	25.19 ± 0.29	26.55 ± 0.74	30.00 ± 0.05	29.53 ± 0.06	29.98 ± 0.05	29.60 ± 0.06	30.03 ± 0.04	29.39 ± 0.12	29.08 ± 0.05	28.81 ± 0.14
libtiff	44.37 ± 0.39	44.76 ± 0.30	35.10 ± 2.23	27.41 ± 2.05	44.77 ± 0.25	41.72 ± 0.43	45.40 ± 0.23	44.45 ± 0.22	46.09 ± 0.23	44.38 ± 0.39	45.70 ± 0.26	44.38 ± 0.26	45.86 ± 0.29	43.90 ± 0.32
libxml2	21.10 ± 0.29	20.39 ± 0.31	10.60 ± 0.29	14.99 ± 0.44	22.47 ± 0.22	22.74 ± 0.37	19.47 ± 0.14	19.84 ± 0.20	20.75 ± 0.21	20.64 ± 0.32	19.53 ± 0.22	19.14 ± 0.10	20.06 ± 0.14	19.07 ± 0.07
php-exif	2.24 ± 0.04	2.34 ± 0.01	2.28 ± 0.04	2.36 ± 0.00	2.37 ± 0.00	2.37 ± 0.00	2.25 ± 0.04	2.36 ± 0.00	2.30 ± 0.03	2.36 ± 0.00	2.29 ± 0.03	2.36 ± 0.00	2.26 ± 0.04	2.36 ± 0.00
poppler	35.96 ± 0.00	35.95 ± 0.00	1.49 ± 0.00	1.76 ± 0.01	41.12 ± 0.05	38.35 ± 0.06	41.40 ± 0.06	36.74 ± 0.30	41.13 ± 0.06	37.92 ± 0.32	41.30 ± 0.07	36.69 ± 0.27	41.44 ± 0.07	36.85 ± 0.33

(B) FTS code coverage with AFL and AFL++.

	FULL		EMPTY		PROV		MSET		CMIN		WOPT		WMOPT	
Target	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++	AFL	AFL++
freetype2	48.01 ± 0.12	44.65 ± 0.41	17.31 ± 0.16	33.85 ± 0.78	34.99 ± 0.17	45.41 ± 0.59	47.58 ± 0.14	44.84 ± 0.34	47.84 ± 0.15	44.11 ± 0.42	47.82 ± 0.20	43.24 ± 0.24	47.73 ± 0.16	45.12 ± 0.21
guetzli	67.34 ± 0.00	67.34 ± 0.00	31.05 ± 0.14	30.31 ± 0.14	75.55 ± 0.12	73.61 ± 0.08	72.95 ± 0.14	69.84 ± 0.10	73.12 ± 0.11	70.76 ± 0.07	72.93 ± 0.10	70.16 ± 0.07	72.79 ± 0.11	69.92 ± 0.08
json	90.29 ± 0.08	90.87 ± 0.11	90.00 ± 0.34	90.18 ± 0.31	90.32 ± 0.26	90.74 ± 0.22	90.60 ± 0.06	90.96 ± 0.06	90.62 ± 0.07	91.14 ± 0.08	90.82 ± 0.10	90.96 ± 0.07	90.79 ± 0.08	91.05 ± 0.06
libarchive	17.14 ± 0.12	16.67 ± 0.47	17.84 ± 0.46	23.09 ± 0.77	18.04 ± 0.63	24.63 ± 0.68	18.51 ± 0.19	20.14 ± 0.39	18.54 ± 0.22	22.18 ± 0.24	18.65 ± 0.10	21.55 ± 0.21	18.36 ± 0.20	21.61 ± 0.27
libjeg-turbo	16.31 ± 0.30	15.38 ± 0.08	18.56 ± 0.09	18.01 ± 0.07	18.84 ± 0.24	18.13 ± 0.07	20.53 ± 0.11	19.24 ± 0.32	20.54 ± 0.11	19.50 ± 0.24	20.47 ± 0.12	19.25 ± 0.30	20.41 ± 0.11	19.01 ± 0.38
libpng	32.81 ± 0.12	34.87 ± 0.10	19.30 ± 0.00	29.66 ± 0.23	25.40 ± 0.00	33.83 ± 0.21	34.92 ± 0.09	36.62 ± 0.18	35.09 ± 0.06	36.82 ± 0.13	34.97 ± 0.06	36.95 ± 0.15	34.94 ± 0.07	36.63 ± 0.12
libxml2	14.90 ± 0.09	14.49 ± 0.03	6.77 ± 0.09	8.12 ± 0.49	–	–	15.81 ± 0.15	15.08 ± 0.12	15.91 ± 0.13	14.95 ± 0.03	16.01 ± 0.02	15.06 ± 0.15	15.76 ± 0.15	15.29 ± 0.18
pcre2	60.81 ± 0.16	63.97 ± 0.14	59.65 ± 0.15	63.13 ± 0.20	–	–	60.94 ± 0.16	63.21 ± 0.22	61.04 ± 0.22	63.64 ± 0.24	61.09 ± 0.23	63.03 ± 0.16	61.00 ± 0.16	62.92 ± 0.22
re2	59.00 ± 0.07	58.96 ± 0.06	59.26 ± 0.16	58.13 ± 0.53	–	–	59.05 ± 0.06	59.03 ± 0.05	59.08 ± 0.04	59.00 ± 0.06	59.10 ± 0.05	57.40 ± 0.83	59.05 ± 0.08	59.00 ± 0.07

impact on fuzzing outcomes (with respect to both bug finding and code coverage). However, a minimized corpus is always better due to the faster iteration rate. While our results show that this may not necessarily find more bugs in a given trial, the fuzzer can more quickly discard inputs not worth exploring. We, therefore, recommend the adoption of OPTIMIN, given the considerably smaller corpora it produces.

When to use the empty seed. While our results demonstrate that corpus minimization achieves the best results, there were nine occasions (three in each of the Magma, FTS, and real-world benchmarks) where EMPTY performed as well as or better than the minimized corpora. These occasions correspond to when coverage is at its lowest, suggesting these are shallow bugs. Thus, we recommend an additional campaign with the empty seed to quickly weed out shallow bugs where possible.

Corpus minimization as lossy compression. Prior work [183, 214] shows how different coverage metrics can affect fuzzing results in practice. Similarly, corpus minimization can also use coverage metrics not solely based on code coverage (or approximate edge coverage in AFL's case). Corpus minimization based solely on code coverage is effectively a form of *lossy compression* [59]: program states may be discarded if they do not expand code coverage. Indeed, we saw this in Section 4.4.2, where the AFL/AFL++ corpus sizes differed due to different-sized coverage maps. We leave it to future work to explore how corpus minimization generalizes to other coverage metrics.

Generalizing to other fuzzers. We limit our experiments to two coverage-guided, mutational greybox fuzzers: AFL and AFL++. We selected AFL because it is widely evaluated and deployed, while AFL++ is an updated and maintained version of AFL incorporating broad improvements from recent fuzzing research and regularly outperforms *all* other fuzzers on Google's FUZZBENCH [144]. While it is unclear how our results might generalize to other fuzzers (e.g., honggfuzz [207] and libFuzzer [188], both of which provide corpus minimization capabilities), we believe our AFL++ results—which demonstrate how seed selection practices impact a range of recent advances in fuzzing research—are generalizable to other mutation-based greybox fuzzers. We leave it to future work to confirm this.

4.5 Chapter Summary

In this chapter we presented, to the best of our knowledge, the first in-depth analysis of the impact seed selection has on mutation-based coverage-guided greybox fuzzing.

We argue that the choice of fuzzing corpus is a critical decision—often overlooked—made before a fuzzing campaign begins. Our results provide ample confirmation of this criticality. In particular, we demonstrate how fuzzing outcomes vary significantly depending on the initial seeds used to bootstrap the fuzzer; especially with respect to state space exploration.

Now that the fuzzer is bootstrapped with an appropriate seed corpus, we turn our attention to running the fuzzer and the instrumented target (② and ③ in Fig. 1.1, Chapter 1). In particular, in the next chapter we introduce a new coverage metric: *def-use* chains.

Chapter 5

Data-Flow-Guided Fuzzing

In Chapter 3 we surveyed the coverage metrics fuzzers use to abstract a target's state space and measure state space search. We found that most fuzzers use coverage metrics derived from *control-flow* features (e.g., basic block, edge). However, in some targets, control flow offers only a coarse-grained approximation of program behavior; instead, *data flow* provides a more fine-grained view.

5.1 Introduction

Exploiting dynamic information drives fuzzer efficiency. Tracking code executed in a target allows a fuzzer to focus its mutations on inputs that reach new code. However, per Section 1.2, tracking code coverage (i.e., control-flow coverage) only covers one dimension of a program's state space. Is this sufficient? We answer this question here.

Chapter outline. We begin by motivating the need for data-flow-guided fuzzers (Section 5.2). Then, we introduce our approach for tracking run-time data flows with low overhead: the DATAFLOW fuzzer (Section 5.3). Inspired by AFL-Sensitive [214], DATAFLOW provides a tunable sensitivity range, balancing the computational cost of exploration with precision. We discuss this in Section 5.3.1, followed by a discussion of DATAFLOW's implementation in Section 5.4. Finally, we evaluate DATAFLOW in Section 5.5. This evaluation compares DATAFLOW to state-of-the-art fuzzers driven by control flow, taint analysis, and data flow.

5.2 Motivation

In some targets, control flow offers only a coarse-grained approximation of program behavior. This includes targets whose control structure is decoupled from its semantics (e.g., LR parsers generated by `yacc` [224]). Such targets require *data-flow*

coverage [65, 88, 101, 175, 202, 224] to accurately capture program behavior. Whereas control flow focuses on the order of operations in a program (i.e., branch and loop structures), data flow instead focuses on how variables (i.e., data) are defined and used [175]; indeed, there may be no control dependence between variable definition and use sites.

5.3 Design of a Data-Flow-Guided Fuzzer

A greybox fuzzer should maintain *accurate* coverage information without negatively impacting *performance*. These requirements exist irrespective of the coverage metric used. With this in mind, we describe: (i) a theoretical foundation for constructing data-flow-based coverage metrics; (ii) how DATAFLOW incorporates these observations; and (iii) the implementation of a DATAFLOW prototype.

5.3.1 Coverage Sensitivity

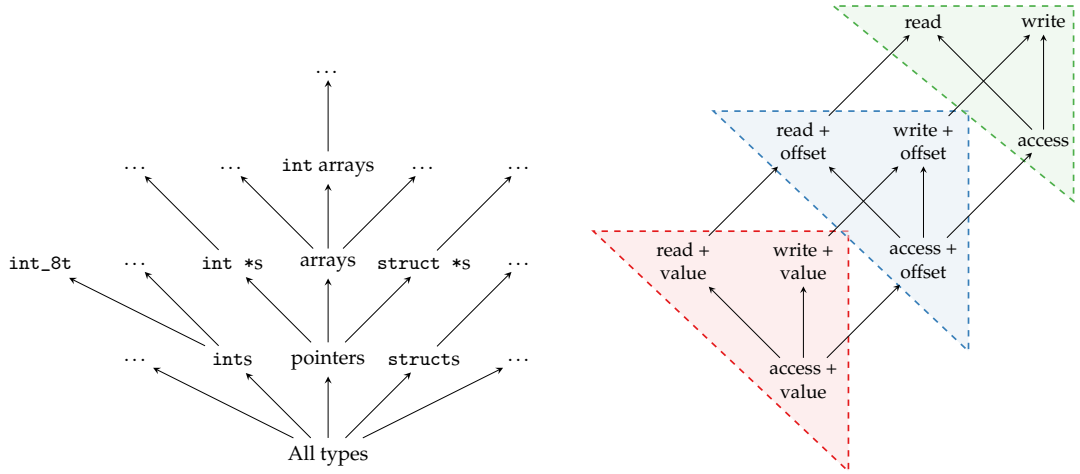
We define data-flow coverage as follows:

Definition

Data-flow coverage is the tracking of *def-use* chains executed at run time.

This definition allows us to explore data-flow-based coverage metrics with different *sensitivities* [183, 214]. We follow the program analysis literature and define sensitivity as a coverage metric's ability to discriminate between a set of program behaviors [114]. In fuzzing, a coverage metric's sensitivity is its ability to preserve a chain of mutated test cases until they trigger a bug [214]. Different sensitivities allow us to balance efficacy and performance: more sensitive metrics incur higher performance penalties. For example, edge coverage sensitivity is increased by incorporating function call context [37]. However, this requires additional instrumentation, increasing run-time overhead [183].

Like traditional data-flow analysis, our data-flow coverage metric requires identifying variable *def* and *use* sites. Following Horgan and London [93], we define a data-flow variable *def* site as a name referring to storage allocated statically (e.g., storage class static, global) or automatically (i.e., local to a procedure). We deviate from this definition by: (i) including calls to dynamic memory allocation routines (e.g., `malloc`); and (ii) excluding reallocations/reassignments that would traditionally *kill* a definition. Instead, *def*s are only killed when they (a) go out of scope (e.g., a local variable in a returning procedure), or (b) are explicitly deallocated (e.g., via `free`).



(A) *Def* site sensitivity lattice. Variables are partitioned based on their type.

(B) *Use* site sensitivity lattice. Variables are partitioned based on their access (e.g., read, write) and what information about the access is recorded (e.g., offset, value).

FIGURE 5.1: *Def* and *use* site sensitivity lattices. The sensitivity of coverage metrics increases towards the bottom.

Consequently, a *use* site includes both reads/writes from/to a *def* site. We deviate from the classic definition to ensure scalability: the difficulties of scaling data-flow analyses on real-world programs are well known [29, 88, 202]. We believe reducing precision by not killing definitions (when assigning a new value to a variable) is a suitable trade-off to maintain scalability.

Once we identify *def* and *use* sites, DATAFLOW instruments these sites (using compiler-based instrumentation, discussed in Section 5.4) so *def*-*use* chains can be tracked at run time. However, exactly which *def*-*use* sites are instrumented (and hence which are tracked) depends on the required sensitivity. Inspired by Wang et al. [214], this leads us to define a pair of sensitivity lattices—one for *def* sites and another for *use* sites, in Fig. 5.1—that can be composed to achieve the desired overall sensitivity (we discuss related limitations in Section 5.4.4). The remainder of this section reuses the code from Fig. 1.2a , which we reproduce in Fig. 5.2 for convenience.

Def Site Sensitivity

Complete data-flow coverage requires identifying and instrumenting *all* variable *def* sites. Unfortunately, the overhead to achieve this level of sensitivity is prohibitively expensive [28]. Therefore, a method for identifying (and hence instrumenting) a subset of important program variables is required. Ideally, this would be an (almost entirely) automated process, reducing the developer burden on the user.

```c
static char buf[128];

void foo(int a, size_t b) {   memset(buf, a, b); }
void bar(int a, size_t b) {   foo('A', sizeof(buf)); }
void baz(int a, size_t b) {   bar(a, sizeof(buf)); }

const struct handler_t {
  char code;
  void (*handler)(int, size_t);
} handlers[] = {{0x66, foo}, {0x61, bar}, {0x7a, baz}};

int sig_cmp(void *data, size_t start, size_t len) {
  if (start + 3 > len)
    return 1;

  static const char sig[3] = {0xa, 0xb, 0xc};
  return memcmp(&data[start], sig, 3);
}

int main(int argc, char *argv[]) {
  struct stat st;

  int fd = open(argv[1], O_RDONLY);
  fstat(fd, &st);
  size_t size = st.st_size;
  char *data = malloc(size);
  read(fd, data, size);

  if (sig_cmp(data, 0, size))
    return 1;

  for (unsigned i = 4; (i + 3) <= size; i += 3)
    for (unsigned j = 0; j < 3; ++j)
      if (data[i] == handlers[j].code)
        handlers[j].handler(data[i + 1], data[i + 2]);

  free(data);
  close(fd);

  return 0;
}
```

FIGURE 5.2: Source code for the simple dispatch table from Fig. 1.2a .

One approach is to partition *def* sites by *type* and restrict instrumentation to *def* sites of a given type (or type set). Figure 5.1a shows the sensitivity lattice for this type-based partitioning.

Partitioning *def* sites by type has several advantages. For example, instrumenting array variables focuses the fuzzer on memory-safety vulnerabilities. Similarly, tracking the data flow of structs may allow for the discovery of type confusion vulnerabilities [105, 197]. Type-based partitioning requires some upfront knowledge of the target to ensure meaningful variables are tracked at run time. For example, the fuzzer may miss important program behaviors (and hence bugs) if "uninteresting" variables are tracked (e.g., `st` at line 21 in Fig. 5.2).

Requirement

Tracking *all* data flows is prohibitively expensive. Identification (and instrumentation) of only important variables is required.

Use Site Sensitivity

Figure 5.1b shows the *use* site sensitivity lattice. Variables are either read from or written to (i.e., "accessed"). Variable accesses are strictly more sensitive than writes or reads on their own. The simplest and least sensitive metrics only track when a variable is accessed (shown at the top of the lattice).

Conversely, the most sensitive data-flow coverage metrics are ones that track not only *when* a particular variable is accessed, but the *value* of that variable when accessed. For example, considering line 17 in Fig. 5.2, this is the difference between reading from `data` and reading the value `0xa` from `data`. The latter is akin to traditional data-flow testing, which focuses on the values that variables take at run time [175, 202], and is similar to GREYONE, which monitors (a subset of) program variables and their values to infer taint [68]. Depending on the *def* site sensitivity, this approach will quickly saturate the fuzzer's coverage map (due to the path collision problem [67]); a middle ground between this overly sensitive approach and simple accesses is required.

We achieve this middle ground by incorporating more fine-grained spatial information into a variable's *use*. This is particularly useful when *def* sites include arrays and/or structs (e.g., `handlers` in Fig. 5.2), as *def-use* chains are now differentiated by the *offset* at which an array/struct is accessed (analogous to a field-sensitive static analysis).

Requirement

Information at different granularities is recorded at *use* sites. Care is required when recording more precise information to ensure the coverage map does not saturate, clogging the fuzzing queue.

Composing Sensitivity Lattices

Different *def-use* sensitivities can be composed to track data flow at different granularities. We reuse the code in Fig. 1.2a —reproducing it in Fig. 5.2 for convenience—to illustrate this. For example, given the *def* sensitivity lattice in Fig. 5.1a , the following may be tracked: (i) all variables; (ii) integer variables (e.g., `fd`, `i`); (iii) arrays (e.g., `handlers`); or (iv) each `struct` in `handlers`. To simplify our presentation, we restrict *def* site instrumentation (and hence *def-use* chain tracking) to the `handlers` array. This leads to varying *def-use* chains depending on the *use* site sensitivity.

Simple access. The green region in Fig. 5.1b . Tracks when `handlers` is accessed (lines 34 and 35 in Fig. 5.2). This results in two *def-use* chains: line 10 $\rightsquigarrow$ line 34, and line 10 $\rightsquigarrow$ line 35. However, this provides a poor approximation of program behavior: information about the specific `handler` executed is lost.

Access with offset. The blue region in Fig. 5.1b . Tracks when `handlers` is accessed along with the offsets where `handlers` is accessed (at index `j`). This provides a more complete view of how `handlers` is used with negligible overhead. This is similar to MEMFUZZ's approach, which incorporates memory accesses into code coverage [49]. This results in 2×3 *def-use* chains: one for each read at each index `j`.

Access with value. The red region in Fig. 5.1b . Tracks when `handlers` is accessed along with the values (being read) during these accesses. This is the most sensitive *use* site coverage metric and achieves the goal of traditional data-flow coverage: associate values with variables, and how these associations can affect the execution of the target [175]. This is similar to GREYONE's "taint inference", which looks at the value of variables used in path constraints [68].

Again, this level of sensitivity results in six *def-use* chains. Here, `handlers`' value range is fully deterministic. However, in general these values will depend on user input, resulting in rapid saturation of the fuzzer's coverage map.

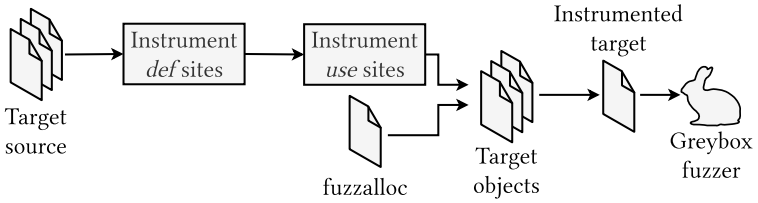


FIGURE 5.3: High-level overview of DATAFLOW.

Requirement

Sensitivity lattice composition must balance efficacy and performance: too precise, and the fuzzer's coverage map will saturate, reducing throughput.

By composing *def* and *use* sensitivity lattices, we realize a variety of data-flow-based coverage metrics. We do this in our fuzzer, DATAFLOW, described in the following sections.

5.4 DATAFLOW Implementation

Figure 5.3 depicts DATAFLOW's high-level architecture, including: (i) compiler instrumentation (built on LLVM v12) for capturing *def-use* sites at the desired sensitivity (Sections 5.4.1 and 5.4.2); and (ii) a runtime library for feeding data-flow information to the fuzzing engine (Section 5.4.3).

Our architecture is agnostic to the underlying fuzzer; the instrumented target produced by the compiler (and linked with the `fuzzalloc` runtime library) can be executed by any American Fuzzy Lop (AFL)-based fuzzer (i.e., any fuzzer using an AFL-style coverage map). However, instead of recording and tracking control-flow coverage, the fuzzer's coverage map tracks data-flow coverage. DATAFLOW is available at `https://github.com/HexHive/datAFLow`.

5.4.1 *Def-use* Site Identification

We must first identify *def* and *use* sites so that data flows between these sites can be tracked. Per Section 5.3.1, *def* site selection impacts coverage sensitivity: more instrumented *def* sites leads to more complete data-flow coverage. We implement several *def* site instrumentation schemes based on the type-based partitioning described in Section 5.3.1.

We make the following assumptions during *def-use* site identification. First, we assume debug metadata is available in the LLVM intermediate representation (IR).

We use this metadata to identify and limit variable *def* sites to source-level variables. Second, we assume tracked variables are accessed via memory references (i.e., `load`/`store` instructions), rather than registers. This is automatic for most composite types (e.g., arrays). For primitive types (e.g., integers), this requires demoting registers to memory references (via LLVM's `reg2mem` pass).

The first assumption reduces the number of potential data flows and is adopted from prior work [61, 141]. The second assumption limits *use* sites to memory access instructions, simplifying instrumentation. We apply existing LLVM transforms to limit *use* sites to two instructions: `loads` and `stores`.[1] Exactly which instructions we instrument depends on the *use* sensitivity required (configured at compile time). We describe our instrumentation in Section 5.4.2.

5.4.2 *Def-use* Tracking

We reduce the run-time tracking of *def-use* chains to a metadata management problem. Here, *def* site identifiers are the metadata requiring efficient retrieval at *use* sites. Inspired by AFL's approach for tracking edge coverage—where basic blocks (in the LLVM IR) are statically assigned a random 16-bit integer—we statically "tag" *def* sites (again, in the LLVM IR) with a random 16-bit integer (Section 5.4.2). This tag is then propagated to *use* sites, where it is retrieved and used to construct a *def-use* chain (Section 5.4.3).

Def Site Instrumentation

We adopt Padding Area MetaData (PAMD) [128] for tracking *def-use* chains. PAMD extends *baggy bounds checking*, a technique proposed by Ding et al. [50] for protecting C and C++ code against buffer overruns. PAMD attaches inline metadata to memory objects (hence our assumption that tracked variables are accessed via memory references; Section 5.4.1) and provides constant-time lookup of this metadata. This lookup occurs via the "baggy bounds table", which stores the binary logarithm of an object's size and alignment (denoted e). Once e is retrieved from the baggy bounds table, the base and size of an object pointed to by p is computed using:

$$\text{base} = p \,\&\, {\sim}(2^e - 1) \tag{5.1}$$

$$\text{size} = 2^e \tag{5.2}$$

Equations (5.1) and (5.2) require an object's size and alignment to be a power-of-two. To meet this requirement, PAMD pads static objects (i.e., stack and global variables)

[1] We lower atomic memory intrinsics and expand `llvm.mem*` intrinsics so we can focus on `load`/`store` instructions (both of which are trivial to identify and hence instrument).

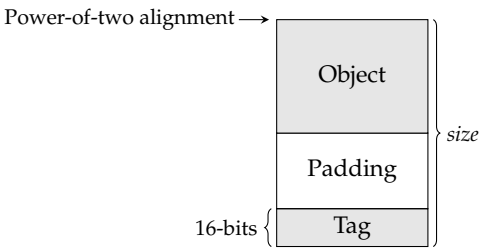


FIGURE 5.4: PAMD's approach for inline metadata. "Object" is aligned to a power-of-two boundary and "padding" is inserted to ensure *size* is a power-of-two.

before attaching the *def* site tag. Figure 5.4 illustrates this process. For example, given a 4-byte object, then $size = 8$, $e = 3$ (the binary logarithm of *size*), and two bytes of padding is inserted before the tag.

Objects whose padding or overall size becomes too large for static allocation are "*heapified*" (i.e., move to the heap). We adopt CCured's [154] approach to heapify objects. For heap-allocated objects (including heapified objects), calls to `malloc`, `calloc`, and `realloc` are replaced with tagged versions (e.g., `__bb_malloc`) accepting the 16-bit tag as an additional argument. Figure 5.5 demonstrates our *def* site instrumentation on a static global variable.

Figure 5.5 shows a snippet of the (un)instrumented LLVM IR for the simple dispatch table in Fig. 1.2 (Chapter 1). We focus our *def* site instrumentation on the global `handlers` array. During compilation, DATAFLOW resizes `handlers` to meet PAMD's object size requirement. Here, 14 B of padding are inserted before the two-byte tag (line 8 in Fig. 5.5b). DATAFLOW then tags this *def* site with the identifier `1337` (line 9). `Handlers` is a global variable, so registration in the baggy bounds table occurs via a constructor (line 13) inserted by the compiler.

Use Site Instrumentation

Per Section 5.4.1, *use* sites are limited to `load` and `store` instructions in the LLVM IR (e.g., line 12 in Fig. 5.6a). We instrument these instructions with a call to `__hash_def_use`, which retrieves the object's size from the baggy bounds table and uses this size to retrieve the *def* tag. The size is also used to determine the offset at which an object is accessed (enabling the *access with offset* sensitivity described in Section 5.3.1). Like *def* sites, *use* sites are tagged at compile time with a randomly-generated identifier (e.g., `4242` at line 12 in Fig. 5.6b). Finally, we leverage several techniques from AddressSanitizer (ASAN) [187] to limit the number of *use* instrumentation sites, thereby reducing overhead without sacrificing precision. We describe the

```
; const struct handler_t handlers[] =
;   {{'f', foo}, {'a', bar}, {'z', {baz}}}
@handlers = [3 x %struct.handler_t] [
  %struct.handler_t {i8 102, void (i32, i64)* @foo},
  %struct.handler_t {i8  97, void (i32, i64)* @bar},
  %struct.handler_t {i8 122, void (i32, i64)* @baz}
], align 16
```

(A) Original code.


```
; Assign handlers the tag "1337"
@handlers = <{ [3 x %struct.handler_t], [14 x i8], i16 }> <{
  [3 x %struct.handler_t] [
    %struct.handler_t {i8 102, void (i32, i64)* @foo},
    %struct.handler_t {i8  97, void (i32, i64)* @bar},
    %struct.handler_t {i8 122, void (i32, i64)* @baz}
  ],                          ; The original array
  [14 x i8] zeroinitializer, ; Padding
  i16 1337                    ; Tag
}>, align 64

; Register the allocation in the baggy bounds table
define void @fuzzalloc.ctor() {
entry:
  %cast = bitcast @handlers to i8*
  call void @__bb_register(i8* %cast, i64 64)
  ret void
}
```

(B) Instrumented code.

FIGURE 5.5: Example *def* site instrumentation of the simple dispatch table in Fig. 1.2a . DATAFLOW replaces the `handlers` global array with a tagged version (tag = 1337) and registers this allocation in the baggy bounds table (in the `fuzzalloc.ctor`).

```
; data[i] == handlers[j].code

; Load data[i]
%data_idx = getelementptr inbounds i8*, i8* @data, i64 %i
%0 = load i8, i8* %data_idx

; Load handlers[j].code
%handlers_idx = getelementptr inbounds [3 x %struct.handler_t],
  [3 x %struct.handler_t]* @handlers, i64 0, i64 %j
%code = getelementptr inbounds %struct.handler_t,
  %struct.handler_t* %handlers_idx, i32 0, i32 0
%1 = load i8, i8* %code

; Check equality
%cmp =icmp i8 %0, %1
```

(A) Original code.


```
; Instrument handlers' use with a call to __hash_def_use

%data_idx = getelementptr inbounds i8*, i8* @data, i64 %i
%0 = load i8, i8* %data_idx

; Load handlers[j].code and instrument use
%handlers_idx = getelementptr inbounds [3 x %struct.handler_t],
  [3 x %struct.handler_t]* @handlers, i64 0, i64 %j
%code = getelementptr inbounds %struct.handler_t,
  %struct.handler_t* %handlers_idx, i32 0, i32 0
%1 = load i8, i8* %code
call void @__hash_def_use(i16 4242i8* %code, i64 1)

%cmp =icmp i8 %0, %1
```

(B) Instrumented code.

FIGURE 5.6: Example *use* site instrumentation of the simple dispatch table in Fig. 1.2a . DATAFLOW tags (tag = 4242) and instruments the *use* site with a call to `__hash_def_use`.

internals of `__hash_def_use`, and how it integrates with the fuzzer, in the following section.

5.4.3 Fuzzer Integration

The `__hash_def_use` function constructs a *def-use* chain by hashing together the *def* and *use* sites. This hash is used as a lookup into the fuzzer's coverage map to guide the fuzzer toward discovering new data flows. This is analogous to AFL tracing edges to discover new control flow paths. Consequently, we leverage techniques used by traditional greybox fuzzers (e.g., compact bitmaps) to efficiently record data-flow coverage [138].

In particular, we use *coarse* data-flow coverage metrics—*def-use* chain hit counts stored in a compact bitmap—to achieve efficient fuzzing. While these techniques result in path collisions [67], we are willing to tolerate such imprecision to limit overhead costs. Coarse coverage metrics also lower implementation costs, enabling the reuse of existing fuzzing engines (here, AFL++ [63]).

We adopt AFL's hashing process (Eq. (3.1) in Section 3.2.2) for looking up data flows in the fuzzer's coverage map. *What* is hashed and *how* it is hashed varies according to the desired sensitivity (Section 5.3.1):

Simple access. Xor of the *def* and *use* site tags:

$$p \leftarrow def \oplus use \tag{5.3}$$

Access with offset. The *def* site tag, *use* site tag, and the offset being accessed. The offset (e.g., array index, struct offset) is computed by subtracting the base address—found using Eq. (5.1)—from pointer p. We compute the hash as:

$$p \leftarrow def \oplus (use + \text{offset}) \tag{5.4}$$

Access with value. The *def* site tag, *use* site tag, the offset, and the value accessed. The *def* / *use* tags and offset are left-shifted to allow room for the value hash and reduce collisions. The accessed value is divided into single-byte chunks $\{v_0, v_1, \ldots\}$ that are hashed into the *def-use* chain:

$$p \leftarrow (def \oplus (use + \text{offset}) \ll 2) \oplus (v_0 \oplus v_1 \oplus \ldots) \tag{5.5}$$

This is implemented as a loop, resulting in a double-load of the accessed object (in addition to the `load` in the original code). We implement the `__hash_def_use`

function so that uninstrumented data flows (i.e., those without an entry in the baggy bounds table) are bucketed in their own coverage map entry.

5.4.4 Limitations

Def Site Selection

Our *def* site selection approach (Section 5.3.1) is incomplete: important data flows may be missed if the appropriate *def* sites are not instrumented. Per our *def* site sensitivity lattice, our prototype focuses on composite types (i.e., arrays and structs) and eschews instrumenting primitive types (e.g., integers). While this approach may miss important data flows, we accept this trade-off, given (a) memory safety remains a key concern [147], and (b) the prohibitive run-time overheads when tracking all *def* sites.

Custom Memory Allocators

Identifying *def* sites is complicated because many applications do not directly call the standard allocation routines (e.g., `malloc`), but indirectly through a custom memory allocator. For example, standard memory allocation routines may be wrapped in other functions. These functions may then be indirectly called via global variables/aliases, stored and passed around in structs, or used as function arguments.

To address the challenge imposed by custom memory allocators and memory allocation patterns, DATAFLOW allows the user to specify wrapper functions to tag (in addition to the standard allocation routines). While DATAFLOW requires the user to find these wrappers manually, existing techniques [38] could assist in this process. We wrap these memory allocation routines within *trampoline* functions when their address is taken (e.g., stored in a global variable). Rather than a compile-time *def* site tag (which may not be statically computable), these trampolines revert to using the lower 16-bits of the PC as the *def* site tag. This approach avoids the need for expensive and imprecise static analysis (e.g., to track the access of memory allocators through global variables).

C++ Dynamic Memory Allocation

To simplify our instrumentation, we rewrite C++ `new` calls as `malloc` calls. However, this prevents us from handling any `std::bad_alloc` exceptions, meaning any failed allocations will cause a program crash (irrespective of any exception handlers in place). Such false negatives are removed by replaying crashing inputs through the original target.

Coverage Imprecision

Storing coarse coverage information in a compact bitmap is inherently inaccurate and incomplete [67]. While this may limit DATAFLOW's ability to discover and explore data flows, this limitation is not unique to DATAFLOW, and affects many greybox fuzzers [7, 8, 37, 49, 61, 68, 97, 134, 141, 214, 217, 231].

5.5 Evaluation

We perform an extensive evaluation (over 3 CPU-yr of fuzzing) to answer the following research questions:

RQ 1 Is data-flow-guided fuzzing viable with minimal run-time overheads? (Section 5.5.2)

RQ 2 Does data-flow-guided fuzzing find more or different bugs? (Section 5.5.3)

RQ 3 Does data-flow-guided fuzzing expand more coverage? (Section 5.5.4)

RQ 4 Can we predict *a priori* the targets most amenable to data-flow-guided fuzzing? (Section 5.5.5)

5.5.1 Methodology

Fuzzer Selection

Our evaluation compares the performance of fuzzers using: (i) pure control-flow coverage; (ii) pure data-flow coverage; and (iii) exact and approximate dynamic taint analysis (DTA), combining control-flow coverage with data-flow tracking.

We select AFL++ (commit `3e2986d`) as the pure control-flow-guided fuzzer because it is the current state-of-the-art coverage-guided greybox fuzzer. We configure AFL++ with: (i) link-time optimization (LTO) instrumentation, eliminating hash collisions; and (ii) with and without "CmpLog" instrumentation. CmpLog—inspired by REDQUEEN's input-to-state correspondence [9]—approximates DTA by capturing comparison operands. Similarly, we select Angora as an alternative control-flow-guided fuzzer (using context-sensitive edge coverage) that also incorporates exact DTA. Finally, we select `DDFuzz` as an alternative data-flow-guided fuzzer.[2]

We configure DATAFLOW with: (i) two *def* site sensitivities: arrays only ("A") and arrays + structs ("A+S"); and (ii) three *use* site sensitivities: simple access ("A"), accessed offset ("O"), and accessed value ("V"). We use the notation "X/Y to refer to

[2] `DDFuzz` was published concurrently with DATAFLOW.

TABLE 5.1: Evaluated fuzzer configurations. Angora and DDFuzz use their default map sizes. AFL++'s LTO instrumentation does not require a fixed-size map.

Name	Map size (KiB)	Description
A_{LTO}	–	AFL++ with LTO instrumentation
A_{CL}	–	AFL++ with LTO and CmpLog instrumentation
An	1,024	Angora
DD	64	DDFuzz
$D_{A/A}$	1,024	DATAFLOW with array *def*s with accessed *use*s
$D_{A/A+O}$	1,024	DATAFLOW with array *def*s with accessed offset *use*s
$D_{A/A+V}$	1,024	DATAFLOW with array *def*s with accessed value *use*s
$D_{A+S/A}$	1,024	DATAFLOW with array & struct *def*s with accessed *use*s
$D_{A+S/A+O}$	1,024	DATAFLOW with array & struct *def*s with accessed offset *use*s
$D_{A+S/A+V}$	1,024	DATAFLOW with array & struct *def*s with accessed value *use*s

the composition of *X def* and *Y use* site sensitivities; e.g., "A/A" refers to array *def* and access *use* sites; "A+S/O" refers to arrays + structs *def* and accessed offset *use* sites. The evaluated fuzzers are summarized in Table 5.1.

Target Selection

We evaluate the ten fuzzers in Table 5.1 on the following targets. We fuzz 20 target programs in total.

SPEC CPU2006. The SPEC CPU benchmark suite [89] is an industry-standardized, CPU-intensive benchmark suite for stress-testing a system's processor, memory subsystem, and compiler. We use SPEC CPU2006 to answer RQ 1.

Magma. Unlike other fuzzing benchmarks (e.g., UNIFUZZ [124]), Magma [84] contains ground-truth bug knowledge. We exclude the *php* target because it failed to build with AFL++'s CmpLog instrumentation (failing with a segmentation fault). We use 15 Magma targets to answer RQ 2.

DDFuzz **target dataset.** Mantovani, Fioraldi, and Balzarotti [141] select five targets—*bison*, *pcre2*, *mir*, *qbe*, and *faust*—they believe to contain a large number of data dependencies, and hence are amenable to data-flow-guided fuzzing. We use newer versions of these targets (because some did not compile on Ubuntu 20.04), shown in Table 5.2. We use these targets to answer RQ 3.

Experimental Setup

We conduct all experiments on an Ubuntu 20.04 AWS EC2 instance with a 48-core Intel® Xeon® Platinum 8275CL 3.0 GHz CPU and 92 GiB of RAM. Each fuzz run

TABLE 5.2: DDFuzz target dataset.

Target	Driver	Command line	Commit
bison	`bison`	`@@ -o /dev/null`	`5555f4d`
pcre2	`pcre2test`	`@@ /dev/null`	`db53e40`
mir	`c2m`	`@@`	`852b1f2`
qbe	`qbe`	`@@`	`c8cd282`
faust	`faust`	`@@`	`13def69`

was conducted for 24 h and repeated five times (ensuring statistically sound results). All targets were bootstrapped with their provided seeds.[3] Finally, we (a) manually located and specified memory allocation functions for DATAFLOW to tag, and (b) used Angora's default behavior to discard taint when calling an external library.

5.5.2 Run-time Overheads (RQ 1)

Conventional wisdom assumes data-flow-based coverage metrics are too heavy-weight, adversely affecting a fuzzer's performance by reducing its iteration rate. We investigate the extent to which this assumption is true by isolating the effects of instrumentation overhead *outside* of a fuzzing environment. Per Section 5.5.1, we measure performance overheads on SPEC CPU2006.

Table 5.3 shows the overhead of all ten evaluated fuzzers on all 19 C and C++ targets in the SPEC CPU2006 v1.0 benchmark suite. We compare these measurements against a baseline without instrumentation (`clang` v12), calculating the geometric mean ("geomean") and 95 % bootstrap confidence interval (CI) over three repeated iterations. The following results are omitted because they failed to build or run: AFL++ (LTO) *445.gobmk* triggered a run-time assertion; DATAFLOW (all configurations) *429.mcf* crashed with a run-time segmentation fault; and Angora *447.dealII*, *471.omentpp*, *473.astar*, and *483.xalancbmk* failed to link with DFSAN's runtime library.

Angora has a geomean overhead of 32.79×. This is particularly notable because previous work has found DFSAN—the framework upon which Angora's taint tracking mode is built—to be one of the more performant DTA frameworks [192]. However, while this overhead is significantly higher than AFL++ (LTO) and AFL++ (CmpLog)—which have geomean overheads of 1.19× and 2.80×, respectively—it is important to recall Angora amortizes this cost over the lifetime of a fuzzing campaign by only tracking taint once on a given input over many mutations.

Of the six DATAFLOW configurations, A/A has the lowest overhead (10.69×), while A + S/V has the highest (15.01×). This is unsurprising, given the rolling hash

[3] We contacted Mantovani, Fioraldi, and Balzarotti [141] to obtain their initial seed sets.

TABLE 5.3: SPEC CPU2006 overheads. Computed as the geomean (over three repeated iterations) relative to an uninstrumented benchmark (compiled with `clang` v12). The 95 % bootstrap CI is reported for the geomean across all targets (for a given fuzzer). The bootstrap CI is zero for individual targets and hence is omitted. A ✗ indicates the target failed to compile or run.

Target	Fuzzer (×)									
	A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
400.perlbench	1.27	3.86	141.85	21.81	12.04	12.75	16.34	12.51	13.21	16.79
401.bzip2	1.26	2.17	25.83	2.54	7.75	8.53	11.12	7.69	8.49	11.09
403.gcc	1.30	3.40	21.19	3.45	19.53	21.22	26.07	19.58	21.18	26.20
429.mcf	1.12	2.46	12.08	1.52	✗	✗	✗	✗	✗	✗
445.gobmk	✗	2.48	23.41	5.26	6.99	7.51	9.48	6.92	7.44	9.73
456.hmmer	1.12	3.08	60.41	1.56	13.47	15.07	21.61	13.60	14.95	21.62
458.sjeng	1.21	4.36	29.69	4.44	7.57	8.13	10.05	7.54	8.01	10.32
462.libquantum	1.20	2.40	27.09	1.61	3.81	4.04	6.97	3.70	4.14	6.97
464.h264ref	1.19	1.82	41.01	1.88	100.63	109.40	134.10	100.96	109.68	134.58
471.omnetpp	1.06	2.02	✗	2.05	6.58	6.34	6.82	6.15	6.34	7.56
473.astar	1.13	2.19	✗	1.59	5.53	5.83	6.80	5.60	5.96	7.28
483.xalancbmk	1.29	5.04	✗	3.48	11.44	12.25	15.67	11.66	12.43	15.93
Geomean	1.19 ± 0.00	2.80 ± 0.01	32.79 ± 0.34	2.91 ± 0.03	10.69 ± 0.13	11.41 ± 0.18	14.64 ± 0.22	10.65 ± 0.16	11.47 ± 0.21	15.01 ± 0.21

approach used for the "access with value" *use* sensitivity (Section 5.4.3). Performance improvements are possible by specializing the hash function based on the *type* of value accessed (e.g., hashing a `uint64_t` or `float` value directly, rather than dividing it into single-byte chunks). Increasing the *def* site sensitivity to include structs added minimal overhead. However, this is target specific: the median number of tracked arrays (across the 12 SPEC CPU2006 targets) is 51, compared to 33 structs. This result may not generalize across targets where structs outnumber arrays.

Our results reflect those presented by Liu and Criswell [128] (e.g., *464.h264ref* has the highest run-time overhead in both the original and our work). However, there is a significant increase in our run-time overheads compared to the original PAMD implementation [128]. To validate our PAMD (re)implementation we evaluated a version of DATAFLOW that *only* performed metadata lookup in the baggy bounds table (i.e., it did not construct *def-use* chains nor update the fuzzer's coverage map). This version of DATAFLOW has a geomean overhead of 3.97×. *Def-use* chain construction is a simple xor operation (Section 5.4.3), so we attribute this dramatic increase in run-time overhead to the interaction of the baggy bounds table and coverage map. In particular, cache effects associated with reading from/writing to these two tables.

Finding

Despite building on PAMD—an efficient metadata encoding scheme—DATAFLOW remains impaired by high run-time overheads. Maximizing fuzzer iteration rates (e.g., by lowering run-time instrumentation costs) is crucial to maximizing fuzzing outcomes.

5.5.3 Bug Finding (RQ 2)

Following Sections 2.6.2 and 4.4.3, we again use survival analysis to summarize our bug-finding results. When computing the restricted mean survival time (RMST) we use $N = 5$ repeated trials and an upper bound $T = 24\,\text{h}$. The log-rank test is again used to compute the statistical significance of our bug-finding results; two fuzzers have statistically-equivalent bug survival times if the log-rank test's p-value > 0.05.

We present our bug-finding results in Table 5.4. Based on raw bug counts, AFL++ was the best performing fuzzer, triggering 60 bugs. The two data-flow-driven fuzzers followed this; `DDFuzz` (44 bugs) and DATAFLOW (41 bugs). Angora was the worst-performing fuzzer, triggering only 24 bugs.

DATAFLOW with "simple access" *use* sensitivity ($\text{DF}_{\text{A/A}}$ and $\text{DF}_{\text{A+S/A}}$) was the best-performing version of DATAFLOW (39 bugs). This was followed by $\text{DF}_{\text{A+S/O}}$ (31 bugs). DATAFLOW was "accessed value" *use* sensitivity was the worst performer. This suggests incorporating variable values at *use* sites is not worth the increased run-time cost; simply tracking the existence of *def-use* chains is "good enough" (for discovering bugs).

AFL++ remains the best-performing fuzzer when accounting for RMSTs (i.e., it triggers bugs fastest), outperforming the data-flow-guided fuzzers for the majority of bugs triggered (60 %). However, this result is reversed (i.e., the data-flow-guided fuzzers outperform AFL++) for 14 % of the triggered bugs. Notably, DATAFLOW was the only fuzzer to trigger LUA003 (not previously triggered by any fuzzer in any prior Magma evaluation), while DATAFLOW and `DDFuzz` triggered XML001 (`xmllint`) and LUA004 orders-of-magnitude faster than AFL++. `DDFuzz` was the only fuzzer to trigger PDF008. However, this bug was only triggered once (over five trials) and towards the end of the trial (after 20 h). This suggests that the bug is difficult to find and `DDFuzz` may have just "got lucky". Finally, AFL++ either failed to trigger or was orders-of-magnitude slower at triggering SSL009 (`x509`) and PDF003 (`pdfimages`). *Do these bugs share properties that make them amenable to discovery via data-flow-guided fuzzing?* To answer this question, we examine the two `lua` bugs in greater depth.

TABLE 5.4: Magma bugs, presented as the RMST (in hours) with 95 % bootstrap CI. Bugs never found by a particular fuzzer have an RMST of ⊤ (to distinguish bugs with a 24 h RMST). We only report the RMST for bugs triggered; bugs not triggered by any fuzzer are omitted. The best performing fuzzer (fuzzers if the bug survival times are statistically equivalent per the log-rank test) for each bug is highlighted in green (smaller is better).

Target	Driver	Bug	Fuzzer									
			A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
libpng	`read_fuzzer`	PNG003	0.01 ± 0.01	0.01 ± 0.01	0.01 ± 0.01	0.05 ± 0.01	0.03 ± 0.02	0.06 ± 0.01	0.24 ± 0.15	0.04 ± 0.02	0.03 ± 0.03	0.99 ± 0.44
		PNG006	⊤	0.02 ± 0.02	0.07 ± 0.03	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PNG007	7.47 ± 6.93	17.54 ± 4.15	⊤	19.49 ± 7.37	23.38 ± 1.39	⊤	⊤	19.49 ± 10.21	19.94 ± 9.19	⊤
libsndfile	`sndfile_fuzzer`	SND001	0.43 ± 0.27	0.72 ± 0.29	–	1.75 ± 0.55	0.78 ± 0.20	19.97 ± 6.74	⊤	0.77 ± 0.32	20.44 ± 8.07	⊤
		SND005	0.55 ± 0.21	0.62 ± 0.43	–	2.05 ± 0.79	5.63 ± 2.25	23.25 ± 1.69	⊤	5.63 ± 1.55	15.32 ± 7.81	20.49 ± 7.96
		SND006	0.36 ± 0.23	0.26 ± 0.16	–	1.06 ± 0.21	6.54 ± 5.97	⊤	19.52 ± 10.15	3.05 ± 2.43	⊤	18.96 ± 8.45
		SND007	0.64 ± 0.27	0.27 ± 0.16	–	2.53 ± 0.54	1.39 ± 0.73	⊤	⊤	3.33 ± 2.21	21.12 ± 6.52	22.57 ± 3.24
		SND017	0.43 ± 0.28	0.06 ± 0.02	–	0.00 ± 0.01	0.01 ± 0.02	0.74 ± 1.21	0.04 ± 0.04	0.02 ± 0.02	0.01 ± 0.01	14.42 ± 11.50
		SND020	0.71 ± 0.30	0.49 ± 0.14	–	0.60 ± 0.11	0.30 ± 0.20	4.76 ± 2.72	1.71 ± 0.57	0.61 ± 0.36	5.53 ± 7.39	15.15 ± 10.63
		SND024	0.31 ± 0.24	0.26 ± 0.16	–	1.03 ± 0.21	0.50 ± 0.30	21.61 ± 3.20	19.38 ± 10.45	0.60 ± 0.26	21.03 ± 6.73	15.07 ± 10.72
libtiff	`read_rgba_fuzzer`	TIF002	18.44 ± 6.92	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		TIF007	0.01 ± 0.01	0.02 ± 0.01	0.81 ± 0.40	0.27 ± 0.11	0.17 ± 0.07	0.95 ± 0.63	19.62 ± 9.92	0.31 ± 0.18	2.20 ± 1.85	14.93 ± 10.89
		TIF008	19.97 ± 5.16	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		TIF012	0.16 ± 0.05	0.99 ± 0.47	6.09 ± 7.19	10.32 ± 6.45	3.71 ± 1.45	14.83 ± 11.01	⊤	4.56 ± 3.05	19.50 ± 10.17	⊤
		TIF014	0.60 ± 0.20	1.26 ± 0.37	⊤	15.46 ± 10.33	14.15 ± 7.31	20.12 ± 8.79	⊤	20.07 ± 8.90	19.45 ± 10.29	⊤
	`tiffcp`	TIF005	⊤	⊤	14.91 ± 10.91	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		TIF006	7.52 ± 4.11	15.40 ± 8.28	10.14 ± 6.14	⊤	22.86 ± 2.59	⊤	⊤	18.12 ± 7.14	⊤	⊤
		TIF007	0.03 ± 0.02	0.02 ± 0.02	0.73 ± 0.66	0.29 ± 0.09	0.26 ± 0.08	6.26 ± 7.22	1.68 ± 0.45	0.38 ± 0.20	0.66 ± 0.16	10.57 ± 9.61
		TIF008	⊤	22.17 ± 4.13	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		TIF009	10.33 ± 6.86	13.30 ± 7.97	⊤	⊤	17.70 ± 7.79	⊤	⊤	⊤	⊤	⊤
		TIF012	0.26 ± 0.13	0.71 ± 0.27	11.57 ± 9.09	11.14 ± 7.90	4.99 ± 1.97	20.42 ± 8.11	⊤	7.02 ± 6.80	⊤	⊤
		TIF014	0.50 ± 0.21	1.60 ± 0.70	14.23 ± 7.56	8.79 ± 6.65	12.87 ± 8.05	20.31 ± 8.35	⊤	13.78 ± 7.91	14.22 ± 6.59	⊤

TABLE 5.4: Magma bugs (continued).

Target	Driver	Bug	Fuzzer									
			A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
libxml2	`read_memory_fuzzer`	XML001	⊤	⊤	10.16 ± 3.03	13.11 ± 6.48	23.23 ± 1.74	⊤	⊤	19.99 ± 4.84	⊤	⊤
		XML002	19.51 ± 10.15	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		XML003	4.00 ± 3.45	0.78 ± 0.61	0.01 ± 0.01	0.03 ± 0.03	0.04 ± 0.05	0.05 ± 0.00	0.05 ± 0.04	0.05 ± 0.05	0.04 ± 0.05	0.05 ± 0.06
		XML009	1.07 ± 0.35	0.95 ± 0.29	⊤	14.59 ± 6.94	3.38 ± 3.11	15.31 ± 10.48	⊤	6.70 ± 6.33	15.72 ± 10.08	⊤
		XML017	0.01 ± 0.01	0.01 ± 0.01	0.01 ± 0.01	0.01 ± 0.01	0.07 ± 0.00	0.08 ± 0.00	0.07 ± 0.00	0.07 ± 0.00	0.07 ± 0.00	0.07 ± 0.00
	`xmllint`	XML001	22.88 ± 2.53	⊤	⊤	0.01 ± 0.01	0.06 ± 0.00	0.08 ± 0.00	0.07 ± 0.00	0.06 ± 0.00	0.07 ± 0.00	0.07 ± 0.00
		XML002	22.83 ± 2.64	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		XML009	0.59 ± 0.24	1.14 ± 0.61	4.29 ± 0.12	7.16 ± 3.70	1.57 ± 0.74	18.53 ± 8.70	⊤	1.18 ± 0.65	15.65 ± 10.03	⊤
		XML012	20.61 ± 4.99	22.30 ± 3.84	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		XML017	0.02 ± 0.02	0.02 ± 0.02	0.01 ± 0.01	0.02 ± 0.02	0.05 ± 0.04	0.06 ± 0.04	0.05 ± 0.00	0.05 ± 0.05	0.05 ± 0.00	0.05 ± 0.00
lua	`lua`	LUA003	⊤	⊤	⊤	⊤	⊤	⊤	⊤	19.81 ± 9.47	⊤	⊤
		LUA004	6.06 ± 3.88	10.38 ± 5.67	⊤	0.22 ± 0.37	0.02 ± 0.02	0.01 ± 0.02	0.02 ± 0.01	0.01 ± 0.02	0.21 ± 0.37	0.01 ± 0.02
openssl	`asn1`	SSL001	3.53 ± 3.38	8.34 ± 4.86	⊤	21.83 ± 3.63	⊤	⊤	⊤	⊤	⊤	⊤
		SSL003	0.05 ± 0.07	0.03 ± 0.03	0.00 ± 0.01	0.01 ± 0.01	0.01 ± 0.02	0.01 ± 0.02	0.02 ± 0.03	0.01 ± 0.02	0.01 ± 0.01	0.01 0.02
	`client`	SSL002	0.06 ± 0.00	0.06 ± 0.00	0.01 ± 0.01	0.05 ± 0.05	0.02 ± 0.02	0.03 ± 0.03	0.03 ± 0.03	0.02 ± 0.03	4.82 ± 10.85	0.03 ± 0.02
	`server`	SSL002	0.08 ± 0.00	0.09 ± 0.01	0.18 ± 0.02	5.00 ± 7.60	0.52 ± 0.00	0.69 ± 0.16	0.69 ± 0.27	5.22 ± 7.51	0.39 ± 0.04	0.58 ± 0.02
		SSL020	20.78 ± 6.00	⊤	6.13 ± 0.14	21.90 ± 4.76	⊤	⊤	⊤	⊤	⊤	⊤
	`x509`	SSL009	⊤	⊤	0.01 ± 0.02	0.11 ± 0.00	0.02 ± 0.02	0.02 ± 0.02	0.02 ± 0.02	0.02 ± 0.02	0.01 ± 0.02	0.02 ± 0.02
poppler	`pdf_fuzzer`	PDF010	1.08 ± 0.54	1.47 ± 0.96	⊤	2.23 ± 1.33	11.01 ± 4.15	⊤	16.14 ± 9.47	7.91 ± 6.84	16.90 ± 5.17	22.22 ± 4.03
		PDF016	0.03 ± 0.01	0.02 ± 0.02	0.23 ± 0.00	0.16 ± 0.03	0.37 ± 0.09	0.31 ± 0.01	0.27 ± 0.01	0.31 ± 0.02	0.25 ± 0.01	0.49 ± 0.03
		PDF018	17.60 ± 7.22	15.86 ± 4.14	⊤	17.35 ± 8.27	20.09 ± 5.16	⊤	⊤	⊤	⊤	⊤
		PDF019	23.14 ± 1.94	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PDF021	23.52 ± 1.09	⊤	⊤	22.30 ± 3.86	⊤	⊤	⊤	⊤	⊤	⊤

TABLE 5.4: Magma bugs (continued).

Target	Driver	Bug	Fuzzer									
			A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
poppler	`pdfimages`	PDF002	⊤	21.31 ± 6.08	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PDF003	6.89 ± 3.26	13.29 ± 3.79	0.01 ± 0.01	0.00 ± 0.01	0.00 ± 0.01	0.00 ± 0.01	0.00 ± 0.01	0.00 ± 0.01	4.80 ± 10.86	0.00 ± 0.01
		PDF011	⊤	19.88 ± 9.33	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PDF016	0.02 ± 0.01	0.01 ± 0.01	0.20 ± 0.06	0.11 ± 0.04	0.20 ± 0.08	0.13 ± 0.03	0.19 ± 0.05	0.15 ± 0.03	4.91 ± 7.64	0.16 ± 0.02
		PDF018	3.68 ± 1.61	9.13 ± 6.32	⊤	14.63 ± 6.86	20.66 ± 7.57	⊤	⊤	6.56 ± 7.09	⊤	⊤
		PDF019	21.99 ± 4.54	14.37 ± 7.73	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PDF021	⊤	22.58 ± 3.22	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
	`pdftoppm`	PDF006	20.72 ± 7.43	⊤	⊤	22.86 ± 2.57	⊤	⊤	⊤	⊤	⊤	⊤
		PDF008	⊤	⊤	⊤	23.49 ± 1.16	⊤	⊤	⊤	⊤	⊤	⊤
		PDF010	2.20 ± 1.15	1.78 ± 0.56	12.52 ± 7.98	13.13 ± 7.80	3.70 ± 0.71	⊤	19.50 ± 10.18	3.66 ± 1.30	19.54 ± 10.10	17.32 ± 8.99
		PDF011	21.29 ± 6.14	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		PDF016	0.05 ± 0.05	0.02 ± 0.02	1.29 ± 0.46	0.19 ± 0.07	0.30 ± 0.06	0.23 ± 0.02	0.30 ± 0.06	0.42 ± 0.14	0.17 ± 0.02	0.37 ± 0.02
		PDF018	15.87 ± 5.87	15.34 ± 4.72	⊤	16.95 ± 6.77	15.80 ± 9.85	⊤	⊤	18.77 ± 6.28	⊤	⊤
		PDF021	⊤	17.10 ± 8.38	⊤	20.74 ± 7.38	⊤	⊤	⊤	⊤	⊤	⊤
sqlite3	`sqlite3_fuzz`	SQL002	0.72 ± 0.21	1.59 ± 0.84	⊤	11.67 ± 4.01	15.07 ± 10.75	21.30 ± 6.12	⊤	19.40 ± 10.40	⊤	⊤
		SQL012	22.93 ± 2.41	22.74 ± 2.86	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		SQL013	⊤	23.49 ± 1.15	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		SQL014	4.38 ± 2.57	5.60 ± 4.45	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤
		SQL018	1.74 ± 0.87	2.74 ± 1.37	⊤	17.17 ± 8.20	⊤	21.49 ± 5.68	⊤	20.03 ± 9.00	23.75 ± 0.56	⊤
		SQL020	17.70 ± 8.30	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤	⊤

```
#define l_checkmodep(m) \
  ((m[0] == 'r' || m[0] == 'w') && m[1] == '\0')
```

FIGURE 5.7: LUA003 missing popen check.

LUA003. This bug is caused by a missing check of the "mode" argument to popen. The check is shown in Fig. 5.7. While the check is quickly reached by DDFuzz (after ∼4 h) and all six DATAFLOW variations (on average, after ∼60 s), the exact trigger conditions were only met once by $DF_{A+S/A}$. Upon examining the compiled binary, we found the second check (`m[1] == '\0'`) was optimized to a *branchless* operation (i.e., it did not contain conditional control flow). This effectively makes the program state where `m[1] != '\0'` invisible to a control-flow-guided fuzzer (in particular, there is no explicit edge for AFL++ to instrument). This state is explicitly visible to DATAFLOW, which reaches it after ∼19 h of fuzzing.

LUA004. This is a logic bug, caused by a missing update to the interpreter's "old" program counter (occurring under particular conditions when tracing the execution of a Lua function). Again, there is no explicit "state" in the target's control-flow graph (CFG) for the fuzzer to reach. Rather, the bug is triggered when the `oldpc` field in the `lua_State` struct is not updated. This only happens under particular conditions, again depending on specific data values.

Finding

The control-flow-guided fuzzers (AFL++ and Angora) outperform the data-flow-guided fuzzers (DDFuzz and DATAFLOW) on 60 % of the triggered Magma bugs. However, the data-flow-guided fuzzers significantly outperform the control-flow-guided fuzzers (by orders-of-magnitude) on 11 % of the triggered bugs. These results suggest that fuzzers guided by control flow and data flow should be combined to maximize bug-finding potential.

5.5.4 Coverage Expansion (RQ 3)

Control-flow coverage is typically quantified by reasoning over the target's CFG (e.g., basic blocks, edges, lines of code). For example, FUZZBENCH replays the fuzzer's queue through an independent and precise (i.e., collision-free) coverage metric; specifically, Clang's source-based coverage [144, 209]. However, the equivalent process for quantifying *data-flow* coverage does not exist.

We quantify coverage expansion using both control-flow and data-flow metrics, using (a) static analyses to approximate an upper bound, and (b) dynamic analyses to quantify coverage expansion against this upper bound. The usual limitations of static analysis (e.g., undecidability) mean this upper bound may be larger than the set of executable coverage elements (e.g., a code region may not be reachable from the target's driver, or a pointer's points-to set may be over-approximated). We accept this imprecision for both metrics (because it is a comparative metric that does not rely on an absolute value). Per Section 2.6.4, we use the Mann-Whitney U-test [139] to statistically compare dynamic coverage across fuzzers: two fuzzers cover the same number of coverage elements if the Mann-Whitney U-test's p-value > 0.05.

Control-flow coverage. We use Clang's existing source-based coverage metric [209]. Specifically, we use *region coverage* (as used by FUZZBENCH), Clang's version of statement coverage. Like classic statement coverage, region coverage is more granular than function and line coverage [94]. Region information is embedded into the target during compilation and can be statically extracted using existing LLVM tooling (to obtain the upper bound).

Data-flow coverage. We build a static analysis on SVF [204] (commit `d6fe474`), a state-of-the-art value flow and pointer analysis framework. This static analysis computes the set of *def-use* chains in a target (for the set of tracked variables, as determined by the chosen *def* site sensitivity). This analysis leverages a flow- and context-insensitive interprocedural pointer analysis based on the Andersen algorithm [4].[4] For the dynamic analysis, we modify the PAMD metadata stored at each *def* site (Section 5.4.2) to store a tuple of ⟨*variable name*, *location*⟩, where *location* is another tuple ⟨*source filename*, *function name*, *line*, *column*⟩. Both tuples are constructed by extracting source-level information from the target's debug information. A *use* site (Section 5.4.2) is similarly labeled with a *location* tuple. Unlike the 16-bit tags used by DATAFLOW, this approach does not result in hash collisions and is precise (albeit with a higher run-time cost). Importantly, neither the static nor dynamic analysis takes into account *def-use* chain *values*. We also exclude dynamic memory allocations from these analyses (to simplify run-time *def-use* tracking when faced with custom memory allocators, per Section 5.4.4).

Table 5.5 and Figs. 5.8 and 5.9 summarize our coverage expansion results. Two targets, `bison` and `faust`, failed to build with AFL++'s CmpLog (again, due to a segmentation fault) and are excluded from our results.

[4] We experimented with SVF's flow-sensitive interprocedural analysis but found the run-time overheads prohibitively large.

TABLE 5.5: Coverage expansion (control and data flow) on DDFuzz targets, reported as the mean over five repeated trials with 95 % bootstrap CI. The "static" results give an approximate upper bound, while the "dynamic" results give the percentage of coverage elements covered at run time. The best performing fuzzer (fuzzers if the coverages are statistically equivalent per the Mann-Whitney U-test) for each target is highlighted in green (larger is better). A ✗ indicates the target failed to compile or run.

(A) Control-flow coverage. Quantified in terms of Clang's code region coverage.

Target	Static (#)	Dynamic (%)									
		A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
`bison`	35,476	33.47 ± 0.08	✗	30.84 ± 0.31	29.95 ± 3.21	32.47 ± 0.39	30.83 ± 0.11	27.95 ± 0.56	31.31 ± 0.12	29.59 ± 0.23	28.28 ± 0.27
`pcre2test`	36,973	56.35 ± 1.78	62.22 ± 2.25	23.01 ± 1.68	37.56 ± 0.30	36.21 ± 0.00	22.63 ± 1.31	21.51 ± 1.25	36.13 ± 0.32	21.13 ± 0.66	23.01 ± 1.98
`c2m`	43,765	45.35 ± 0.07	45.86 ± 0.08	43.81 ± 0.22	45.45 ± 0.07	44.22 ± 0.09	41.99 ± 0.15	41.95 ± 0.22	44.18 ± 0.08	42.02 ± 0.21	41.78 ± 0.17
`qbe`	5,400	76.15 ± 0.19	76.34 ± 0.21	73.94 ± 0.03	76.03 ± 0.15	74.17 ± 0.09	73.03 ± 0.03	73.84 ± 0.12	74.54 ± 0.11	73.20 ± 0.14	74.11 ± 0.07
`faust`	26,872	32.11 ± 0.32	✗	29.02 ± 0.09	32.07 ± 0.20	31.02 ± 0.08	30.17 ± 0.04	30.57 ± 0.04	31.13 ± 0.17	30.16 ± 0.03	30.31 ± 0.12

(B) Data-flow coverage. Quantified in terms of interprocedural *def-use* chains (scaled by $\times 10^{-3}$, due to the small percentage of *def-use* chains covered across all targets). The `c2m` target is excluded because it unexpectedly crashed with our precise data-flow tracking instrumentation.

Target	Static (#)	Dynamic ($\times 10^{-3}$ %)									
		A_{LTO}	A_{CL}	An	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
`bison`	9,923,196	11.60 ± 0.10	✗	9.94 ± 0.17	10.40 ± 0.17	11.27 ± 0.26	10.71 ± 0.10	9.57 ± 0.33	11.26 ± 0.16	10.60 ± 0.03	9.84 ± 0.28
`pcre2test`	1,401,778	181.03 ± 14.78	205.78 ± 16.14	78.24 ± 7.23	125.53 ± 1.53	122.72 ± 1.30	76.12 ± 4.87	69.51 ± 5.71	124.83 ± 1.65	69.73 ± 2.86	73.82 ± 5.99
`c2m`	25,462,192	–	–	–	–	–	–	–	–	–	–
`qbe`	450,407	381.97 ± 3.06	378.68 ± 4.46	178.28 ± 0.47	376.64 ± 3.62	363.01 ± 1.71	352.61 ± 0.36	356.21 ± 1.38	366.20 ± 0.40	353.24 ± 0.62	355.77 ± 4.22
`faust`	159,515,187	5.46 ± 0.06	✗	4.88 ± 0.07	5.55 ± 0.06	5.19 ± 0.01	4.81 ± 0.03	5.01 ± 0.05	5.34 ± 0.03	4.78 ± 0.02	4.89 ± 0.07

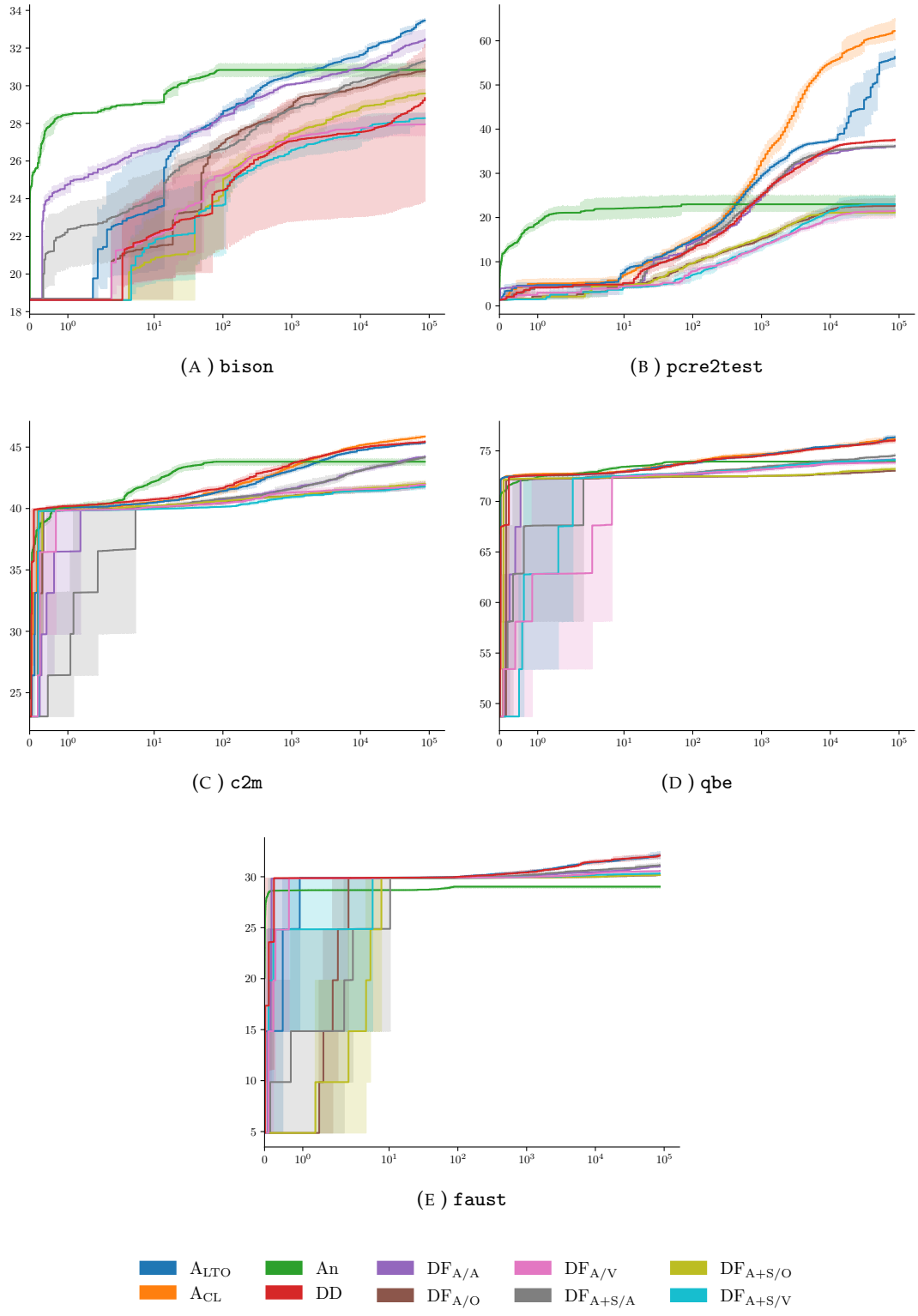


FIGURE 5.8: Control-flow coverage expansion over time. The x-axis is time in seconds (log scale), and the y-axis is the percentage of code regions expanded (against the static upper bound in Table 5.5a). The mean coverage (over five repeated trials) and 95 % bootstrap CI is shown.

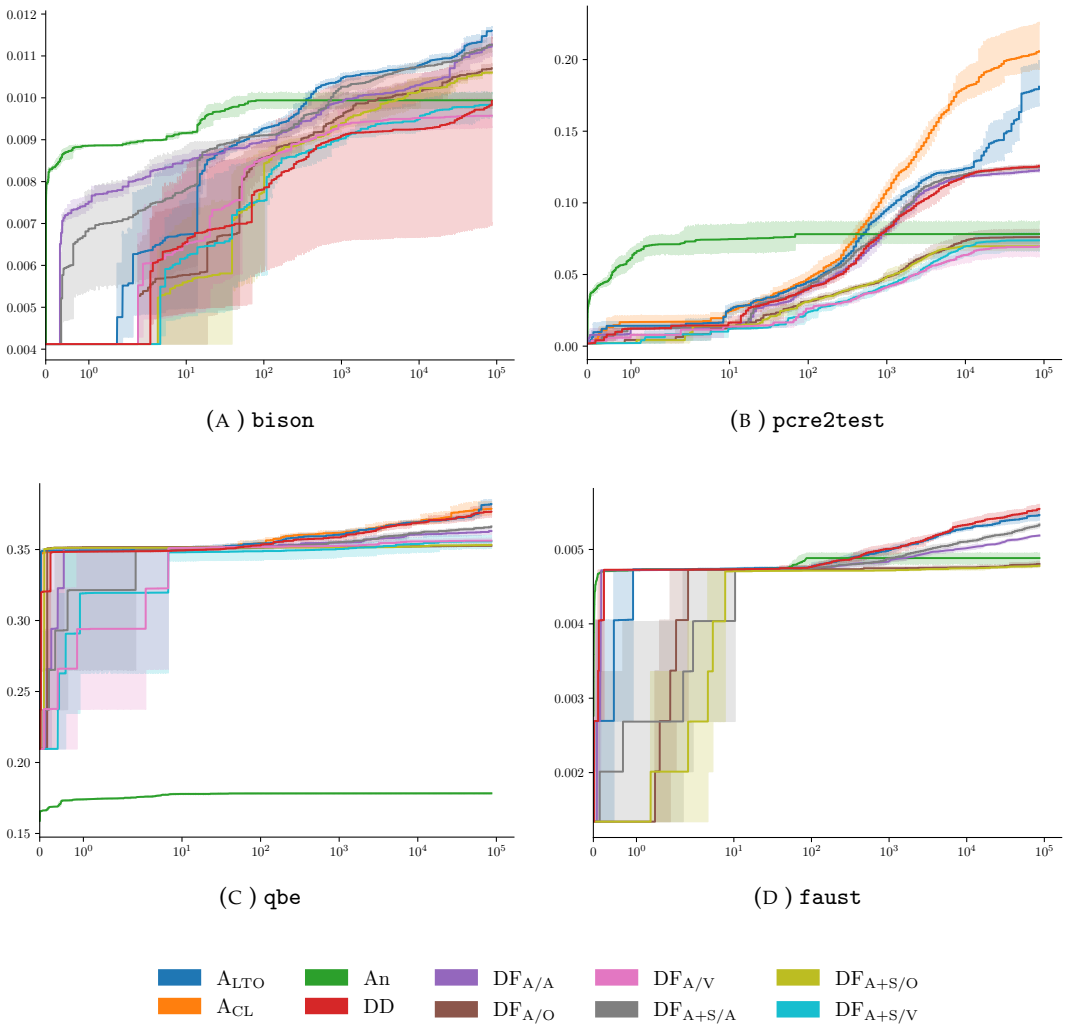


FIGURE 5.9: Data-flow coverage expansion over time. The *x*-axis is time in seconds (log scale), and the *y*-axis is the percentage of *def-use* chains expanded (against the static upper bound in Table 5.5b). The mean coverage (over five repeated trials) and 95 % bootstrap CI is shown.

AFL++ is again the best-performing fuzzer, achieving the highest control-flow (i.e., code region) coverage. CmpLog improves AFL++'s already-strong coverage expansion capabilities. These results are unsurprising, given that control-flow coverage (specifically, edge coverage) guides AFL++. Similarly, Angora again performs poorly, outperformed by both `DDFuzz` and DATAFLOW in maximizing both control- and data-flow coverage. Curiously, however, AFL++ also achieves the highest *data-flow* (i.e., *def-use* chain) coverage. This is despite DATAFLOW's data-flow guidance. We attribute this (surprising) result to the differences in *fuzzer iteration rates* (i.e., the number of inputs executed by the fuzzer per unit of time).

Accounting for Iteration Rates

AFL++ (LTO) achieves a mean iteration rate of 1,172 execs/s (median 347 execs/s). In contrast, `DDFuzz`, Angora, and DATAFLOW achieve mean iteration rates of 974, 616 and 270 execs/s, respectively (median 442, 249 and 144 execs/s). This dramatic decrease in iteration rates reflects our overhead results in Section 5.5.2.

To account for differences in iteration rates, rather than comparing coverage at the *end* of each fuzz run (i.e., after 24 h of fuzzing), we compare coverage at a given *execution* ("exec"). Specifically, we compare coverage at the last exec of the slowest fuzzer (i.e., with the lowest iteration rate). Intuitively, this places a "ceiling" on the coverage achieved by faster fuzzers (i.e., those able to execute more inputs within a single 24 h fuzz run). For example, $\text{DF}_{\text{A+S/V}}$ is the slowest fuzzer on `bison` (85 execs/s). Thus, we compare the coverage achieved at the last execution of $\text{DF}_{\text{A+S/V}}$ (exec = 7,358,231), implicitly ignoring any additional coverage expanded after this exec. Unfortunately, Angora does not provide the necessary information to map coverage to a particular exec, so we exclude it from our analysis (despite it being the slowest fuzzer on two targets: `bison` and `faust`).

We present coverage "normalized" against iteration rates in Table 5.6. DATAFLOW is now more competitive (against AFL++) on expanding data-flow coverage. It achieves the highest *def-use* chain coverage on `bison` and `faust`, and is only ∼4 % behind the number of *def-use* chains expanded by AFL++ on `qbe`. Again, increasing DATAFLOW's *use* sensitivity to include variable values fails to improve fuzzing outcomes. These results reinforce our belief that fuzzer iteration rates have a significant impact on fuzzing outcomes.

TABLE 5.6: Coverage expansion—control and data flow ("CF" and "DF", respectively)—on DDFuzz targets. In contrast to Table 5.5, which reports the (mean) coverage achieved at the *end of a 24 h fuzz run*, here we report the (mean) coverage at the *last exec of the slowest fuzzer* ("Exec" occurring at the given "Time"). Dynamic coverage is quantified in terms of the static analysis results in Table 5.5 (the DF results are again scaled by $\times 10^{-3}$). The best performing fuzzer (fuzzers if the coverage is statistically equivalent per the Mann-Whitney U-test) for each target is highlighted in green (larger is better).

Target	Exec (#)	Time (hr)	Metric	Dynamic (%, $\times 10^{-3}$ %)								
				A_{LTO}	A_{CL}	DD	$DF_{A/A}$	$DF_{A/O}$	$DF_{A/V}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
bison	7,358,231	13.74	CF	32.35 ± 0.09	✗	28.39 ± 3.71	32.17 ± 0.47	30.64 ± 0.11	27.89 ± 0.53	30.90 ± 0.20	29.54 ± 0.23	28.11 ± 0.51
			DF	10.95 ± 0.09	✗	9.56 ± 2.08	11.06 ± 0.24	10.59 ± 0.06	9.55 ± 0.34	11.03 ± 0.08	10.56 ± 0.08	9.79 ± 0.35
pcre2test	37,992,897	8.22	CF	37.51 ± 0.30	55.79 ± 1.09	36.66 ± 0.16	35.85 ± 0.00	22.56 ± 1.28	21.38 ± 1.15	35.82 ± 0.29	21.12 ± 0.66	23.01 ± 1.98
			DF	124.40 ± 1.69	182.84 ± 11.61	122.42 ± 0.77	121.02 ± 0.68	75.96 ± 4.72	69.08 ± 5.31	123.47 ± 1.68	69.73 ± 2.83	73.82 ± 5.99
c2m	516,654	5.27	CF	43.20 ± 0.13	43.44 ± 0.08	43.41 ± 0.27	42.84 ± 0.11	41.44 ± 0.24	41.67 ± 0.25	42.77 ± 0.22	41.39 ± 0.11	41.61 ± 0.19
			DF	–	–	–	–	–	–	–	–	–
qbe	12,589,430	15.78	CF	75.39 ± 0.07	75.27 ± 0.12	75.28 ± 0.10	74.11 ± 0.12	72.93 ± 0.06	73.82 ± 0.12	74.41 ± 0.10	73.02 ± 0.13	74.05 ± 0.09
			DF	368.69 ± 1.33	368.38 ± 1.73	368.33 ± 1.20	362.03 ± 1.89	352.53 ± 0.38	356.17 ± 1.33	364.83 ± 0.75	352.93 ± 0.67	355.68 ± 4.26
faust	1,033,846	23.97	CF	31.26 ± 0.04	✗	31.05 ± 0.08	30.80 ± 0.06	30.12 ± 0.04	30.49 ± 0.06	30.86 ± 0.08	30.15 ± 0.04	30.25 ± 0.12
			DF	5.25 ± 0.02	✗	5.22 ± 0.01	5.13 ± 0.01	4.79 ± 0.03	4.99 ± 0.05	5.22 ± 0.00	4.77 ± 0.01	4.87 ± 0.08

Finding

The control-flow-guided fuzzers (specifically, AFL++) achieve the highest control- *and* data-flow coverage. Data-flow-guided fuzzers require more complex instrumentation (compared to control-flow-guided fuzzers), impairing the fuzzers' iteration rates.

5.5.5 Characterizing Data-Flow (RQ 4)

The fuzzing community has largely settled on control-flow-based coverage metrics—in particular, edge coverage—to drive a fuzzer's exploration. While prior successes have largely validated this approach [52, 181, 189, 207, 231], we wish to understand what (if any) program characteristics lend themselves to data-flow-based coverage.

Mantovani, Fioraldi, and Balzarotti [141] propose the *DD ratio*—defined as the ratio between the number of basic blocks instrumented with data-dependency information over the total number of basic blocks in the target—to determine whether data-flow-based coverage—derived from the target's data dependency graph (DDG)—adds value (e.g., over edge coverage). A higher DD ratio suggests the target is

TABLE 5.7: Characterizing data flow using the data dependency ratio ("DD ratio") introduced by Mantovani, Fioraldi, and Balzarotti [141]. *Strongly* data-dependent targets (i.e., those with a DD ratio $\geq 10\,\%$) are highlighted in green.

(A) Magma.

Target	DD ratio (%)
`png_read_fuzzer`	13.40
`sndfile_fuzzer`	12.14
`tiff_read_rgba_fuzzer`	12.01
`tiffcp`	11.37
`xml_read_memory_fuzzer`	12.86
`xmllint`	13.03
`lua`	12.73
`asn1`	9.89
`client`	9.99
`server`	9.98
`x509`	9.98
`pdf_fuzzer`	11.62
`pdfimages`	9.33
`pdftoppm`	11.77
`sqlite3_fuzz`	12.80

(B) DDFuzz target dataset.

Target	DD ratio (%)
`bison`	6.59
`pcre2test`	22.60
`c2m`	21.82
`qbe`	12.45
`faust`	7.29

more amenable to data-flow-guided fuzzing; a target with a DD ratio above 10 % is considered *strongly* data dependent.

Table 5.7 summarizes the DD ratio of our 20 target programs.[5] Thirteen of these targets (65 %) have DD ratios $\geq 10\,\%$, indicating their suitability for data-flow-guided fuzzing. However, we found little correlation between a target's DD ratio and fuzzing outcomes (both bug finding and coverage expansion). For example, `png_read_fuzzer` had the highest DD ratio among the Magma targets (13.40 %), closely followed by `xmllint` (13.03 %). However, AFL++ outperformed the data-flow-guided fuzzers (DDFuzz and DATAFLOW) on both targets (across bug counts and survival times). Similarly, `pcre2test` and `c2m` had the highest DD ratios among the DDFuzz targets (22.60 and 21.82 %, respectively). Again, AFL++ outperformed the two data-flow-guided fuzzers (across both control- and data-flow coverage expansion).

Based on these results, we conclude that the DD ratio is not suitable for determining a target's suitability for data-flow-guided fuzzing. Instead, we propose using *subsumption*.

[5] These values differ from the original DDFuzz evaluation [141] because we use newer versions of the targets (per Section 5.5.1).

Finding

The "DD ratio" is not suitable for determining whether a target is amenable to data-flow-guided fuzzing. Alternative techniques—such as those based on subsumption—may yield more accurate results.

5.5.6 Discussion

Coverage sensitivity. In Section 5.3.1, we introduced a framework for reasoning about and constructing data-flow coverage metrics for greybox fuzzing. This framework allows the user to balance precision with performance. Our results suggest that fuzzing outcomes (i.e., bug finding and coverage expansion) fail to improve as precision increases. Notably, this finding also applies to Angora; Angora's exact DTA provided little benefit over the approximate DTA used by AFL++'s CmpLog mode. Our results reflect prior findings that demonstrate the importance of maximizing fuzzer iteration rates [9, 68, 92, 173, 225].

Bugs vs. coverage. Böhme, Szekeres, and Metzman [23] found the fuzzer best at maximizing coverage expansion may not be best at finding bugs. Our results reflect this finding; despite AFL++ outperforming DATAFLOW on coverage expansion (Section 5.5.4), DATAFLOW triggered bugs AFL++ failed to find (Section 5.5.3). Ultimately, fuzzers are deployed to find bugs and vulnerabilities; our findings reinforce the need for bug-based fuzzer evaluation [84, 115, 234] (not only a comparison of coverage profiles).

Computing coverage upper bounds with static analysis. In Section 5.5.4 we used static analysis to approximate a coverage upper bound (for both control- and data-flow coverage). In theory, this upper bound is useful for estimating the *residual risk* of ending a fuzz run before maximizing coverage (analogous to the residual risk of missing a bug [20]). In practice, static analysis of "real-world" programs is fraught; dynamically loaded, just in time (JIT), and inline assembly code all impact precision. Even specific command-line arguments influence the reachability of particular code regions. Thus, it is difficult to determine how realistic the upper bounds in Section 5.5.4 are. We return to this issue in Chapter 6.

Control- or data-flow? We hypothesized that data-flow-guided fuzzing offers superior performance on targets where control flow is decoupled from semantics. Our results lead us to reject this hypothesis. In most cases, control-flow-guided fuzzers

outperformed data-flow-guided fuzzers (across both bug-finding and coverage-expansion metrics, and on targets identified as being amenable to data-flow-guided fuzzing). However, we are not prepared to give up on data-flow-guided fuzzing; despite lower run-time costs than DTA, DATAFLOW's run-time costs remain high, negatively impacting coverage expansion. Despite this impediment, DATAFLOW discovers bugs control-flow-guided fuzzers do not. We believe reducing the run-time costs of data-flow-guided fuzzers will improve fuzzing outcomes.

5.6 Future Work

Our results highlight the opportunities for further work on data-flow-based coverage metrics. We propose ideas for further exploration here.

Reducing overheads. The significant run-time overheads remain the primary impediment to the adoption of data-flow-guided fuzzing (see Section 5.5.2). Liu and Criswell [128] propose using interprocedural optimizations to eliminate unnecessary object (de)allocation in the baggy bounds table, improving performance. Similarly, more sophisticated pointer analyses (e.g., those provided by SVF) could be used to eliminate unnecessary *def*/*use* site instrumentation (e.g., removing redundant instrumentation when *def*-*use* chains can be statically identified).

Reducing hash collisions. Per Section 5.4.3, DATAFLOW is prone to hash collisions. It is well known that hash collisions cause fuzzers to miss program behaviors [67]. While AFL++'s LTO mode solves the hash collision problem for edge coverage, we did not investigate a similar technique for *def*-*use* chain coverage. A hash-collision-free DATAFLOW may lead to improved coverage expansion.

Combining control- and data-flow coverage. Finally, DATAFLOW exclusively uses *def*-*use* chain coverage to drive exploration. In contrast, other data-flow-guided fuzzers (e.g., INVSCOV [61], DDFuzz [141]) combine data flow with control flow. Given our bug-finding results—i.e., those where DATAFLOW significantly outperformed AFL++ (e.g., LUA003, LUA004, SSL009, and PDF003)—combining DATAFLOW with edge coverage may provide a "best of both worlds" solution (echoing the conclusions reached by Salls et al. [183]). This combination of coverage metrics could be realized by combining control- and data-flow coverage in a single coverage map, maintaining separate coverage maps, or by dynamically switching between different instrumented targets.

Coverage metric subsumption. Our results in Section 5.5.5 led us to conclude that the DD ratio was not suitable for determining a target's suitability for data-flow-guided fuzzing. Prior work on characterizing programs for automated test suite generation is also unsuitable; e.g., the approaches proposed by Neelofar et al. [155] and Oliveira et al. [160] are specific to object-oriented software and focus on control-flow features. Instead, we propose *subsumption*.

We say that coverage metric $\mathcal{M}_1$ strictly subsumes metric $\mathcal{M}_2$ if covering all coverage elements in $\mathcal{M}_1$ also covers all elements in $\mathcal{M}_2$. For example, edge coverage strictly subsumes basic block coverage. Relaxing this definition of strict subsumption allows us to quantify the number of coverage elements in $\mathcal{M}_2$ not subsumed by $\mathcal{M}_1$. Intuitively, more elements in $\mathcal{M}_2$ *not* subsumed by $\mathcal{M}_1$ implies fuzzing with $\mathcal{M}_2$ will lead to behaviors not detectable by $\mathcal{M}_1$. Static data-flow analysis frameworks such as those proposed by Chaim et al. [29] can be used to perform this subsumption analysis. We leave the investigation of such techniques for future work.

5.7 Chapter Summary

Observing fuzzers that introduce taint tracking along with control flow, we investigate data flow as an alternate coverage metric, making *data-flow coverage* a first-class citizen. Driven by empirical results and the conventional wisdom gathered over years of software-testing research, we hypothesized data-flow-guided fuzzing to offer superior outcomes (over control-flow-guided fuzzing) in targets where control flow is decoupled from semantics.

Our results show that control-flow-guided fuzzing produces better outcomes (bug finding and coverage expansion) in most cases. The high run-time costs associated with data-flow tracking impaired the fuzzer's ability to explore a target's behavior efficiently. Despite these costs, our data-flow-guided fuzzer discovered bugs control-flow-guided fuzzers did not. These results suggest that data-flow-guided fuzzers discover *different*, not *more*, bugs. Specifically, bugs existing in program states not explicitly visible in the target's CFG.

In the next chapter we conclude the fuzzing campaign (④ in Fig. 1.1, Chapter 1). Specifically, we expand on our use of static analyses to quantify control- and data-flow coverage (Section 5.5.4) and investigate their use more broadly in measuring a fuzzer's state space search.

Chapter 6

Quantifying State Space Search

In Chapter 5 we introduced a pair of static analyses for quantifying coverage expansion using control-flow and data-flow metrics. Here, we expand on this idea and investigate the use of several state-of-the-art static analyses for quantifying the upper-bound of a fuzzer's state space search.

6.1 Introduction

Measuring how much of a target's state space has been explored *per fuzzer* is challenging. Fuzzer evaluations typically avoid measuring this by only reporting a count of covered coverage elements (e.g., the number of lines of code executed). However, a raw count is insufficient in many cases. For example, determining the *residual risk* [20] of stopping the fuzzer requires deriving an accurate upper bound of target $\mathcal{P}$'s state space, and from this determining how much of $\mathcal{P}$'s state space has been explored.

One approach for deriving an upper bound of $\mathcal{P}$'s state space is to count the number of edges in $\mathcal{P}$'s control-flow graph (CFG). However, this approach is fraught: while the number of *direct* edges (both intra- and inter-procedural) is computable during instrumentation, *indirect* edges (e.g., function calls made by dynamic dispatch, such as the calls to `bar`, `baz`, and `foo` in Fig. 1.2) and *exceptional* control flow (e.g., `try/catch` blocks) make counting edges difficult (and in some cases, impossible). Moreover, the set of *reachable* basic blocks in $\mathcal{P}$'s CFG depends on the driver (i.e., the program that interfaces with the codebase—typically a library—under test) and even the command-line arguments used to run the driver [235]! The former (basic block reachability) requires accurate control flow recovery, while the latter (command-line arguments) requires accurate data flow recovery. While perfect control- and data-flow recovery is undecidable, all is not lost: modern static analyzers [79, 80, 119, 185, 204] have demonstrated precise and scalable control- and data-flow analyses on real-world

programs [54, 119, 203]. *Can we use these techniques to make an accurate per-fuzzer evaluation?*

Despite their long history, the application of static program analyses to fuzzing—in particular, to reason about state space exploration—is under explored. This chapter aims to rectify this, and improve understanding of fuzzers' state space search through program analysis.

Chapter outline. We investigate the use of several static analyses for quantifying control-flow coverage. We first propose our analysis technique (Section 6.2), and then apply our technique to a large-scale fuzzing campaign (Section 6.3).

6.2 Program Analysis for State Space Search

Traditionally, a fuzzer's state space search is quantified using control-flow-based metrics (e.g., the number of CFG edges or lines of code executed). However, these control-flow-based metrics are often *context insensitive*, despite the fact that different invocations of a function may lead to different program behaviors (as in Figs. 1.2 and 7.1c). Moreover, accurate comparisons of fuzzers' state space search require the ability to unambiguously identify and correlate points across executions [205, 224]. Notably, context-insensitive control-flow coverage (such as the approach used by FUZZBENCH) does not meet this requirement: using {source file $\times$ line number} to uniquely identify an execution point introduces ambiguities; e.g., in the presence of loops (iteration count), function calls (context sensitivity), and data values.

We overcome this issue by introducing a technique that quantifies control-flow coverage using a *context sensitive interprocedural control-flow graph (ICFG)* analysis. This technique can be used (a) for determining how much of a target's state space has been explored by a given fuzzer, and (b) to uniquely identify and compare execution points covered by a fuzzer. The former improves the accuracy of *per-fuzzer* evaluation, while the latter improves *cross-fuzzer* evaluation. We develop (a) a static analysis to quantify an upper bound of reachable context-sensitive edges (Section 6.2.1), and (b) a dynamic analysis to trace context-sensitive edges (Section 6.2.2).

6.2.1 Static Analysis

Our static analysis first constructs an intraprocedural CFG for each function f in $\mathcal{P}$. These CFGs are then combined to create the ICFG by inserting edges from a `call` instruction to the entry block of the callee's CFG. Indirect calls are resolved via an off-the-shelf static pointer analysis (discussed further in Section 6.2.3).

We augment edges in the ICFG with calling context information. The number of calling contexts for f can be statically computed by [206, 238]:

$$\mathcal{C}(f) = \sum_{i=1...n} \mathcal{C}(g_i) \tag{6.1}$$

Where $\{g_1, \ldots, g_n\}$ are the caller functions of f. The total number of context-sensitive edges is obtained by propagating $\mathcal{C}(f)$ to each intraprocedural edge in f. Our ICFG analysis is flow-insensitive, so the number of context-sensitive edges may be infinite in the presence of recursion. In this case, we apply the approach proposed by Sumner et al. [206]: (i) a dummy entry node is inserted into $\mathcal{P}$'s call graph; (ii) edges are added from this dummy node to (a) the original entry node, and (b) any node that is a target of a backward edge (i.e., recursive call); and (iii) backward edges are removed, making the call graph acyclic. From here, Eq. (6.1) is applied. This process implicitly collapses recursive calls and thus mimics the approach taken by fuzzers using context-sensitive coverage (Section 3.2.4).

6.2.2 Dynamic Analysis

Our dynamic analysis inserts instrumentation (at compile time) at the entry of each basic block and function in $\mathcal{P}$. At run time this instrumentation logs (a) hit counts for intraprocedural branches and function calls/returns, and (b) calling contexts. Recursive function calls are collapsed into a single call so the dynamic calling context is aligned with the static calling context. An offline, *post hoc* analysis reconstructs the context-sensitive ICFG from these traces.

6.2.3 Implementation

Our static and dynamic analyses are built on LLVM v14 [120]. We use the following pointer analysis frameworks and configurations for resolving indirect calls (thus improving the accuracy of our static analysis).

Fuzz Introspector. Fuzz Introspector—part of Google's OSS-Fuzz service [189]—is a static analysis for computing the set of reachable functions in a given target [32]. However, Fuzz Introspector "*makes few efforts into resolving indirect calls*", limiting itself to C++ v-table entries and function pointers assigned to variables and passed as function arguments.[1]

[1] Per https://github.com/ossf/fuzz-introspector/blob/f01aaee5/frontends/llvm/lib/Transforms/FuzzIntrospector/FuzzIntrospector.cpp#L967-L969.

SVF. SVF [204] is a value flow and pointer analysis framework. Our SVF analysis uses a flow- and context-insensitive interprocedural pointer analysis based on the Andersen algorithm [4], similar to that used in Section 5.5.4. Also similar to Section 5.5.4, we experimented with SVF's flow-sensitive interprocedural analysis but found that the analysis failed to complete within 24 h on a majority of targets.

SEADSA. SEADSA is a context-, field-, and array-sensitive unification-based pointer analysis. We use the same configuration as Kuderski, Navas, and Gurfinkel [119]: "bottom-up + top-down context sensitive" analysis with "type awareness".

6.3 Evaluation

Our evaluation aims to answer the following research question:

RQ 1 How effective are state-of-the-art static analyses at estimating the upper bound of a target's state space? (Section 6.3.2)

6.3.1 Methodology

Target Selection

We evaluate our static analyses on the FUZZBENCH [144] benchmark. We chose FUZZBENCH over other fuzzer benchmark suites (e.g., Magma [84]) because it is widely used and "*uses code coverage as its primary evaluation metric*" [144]. We exclude the "bug-based" targets and focus on the 23 "coverage" targets. We also exclude *libjpeg-turbo*, *libpcap*, *libxml2*, *proj4*, and *systemd*, which failed to build/run. The remaining 23 targets are listed in Table 6.1.

Experimental Setup

We run a number of fuzzing campaigns using a range of fuzzers.[2] Each campaign consists of ten independent 24 h *trials* (consistent with the recommendations by Klees et al. [115]). All experiments were conducted on a Dell PowerEdge server with a 48-core Intel® Xeon® Gold 5118 2.30 GHz CPU, 512 GiB of RAM, and running Ubuntu 18.04. We use FUZZBENCH's default starting seeds to bootstrap all trials.

[2]These are the same campaigns as those in Chapter 7. Because we are only interested in the static analysis results here, we defer the details of these campaigns until Section 7.3.

TABLE 6.1: FUZZBENCH target statistics, including: the programming language the target is written in ("Lang"), the number of functions ("F"), the number of basic blocks ("BB"), and the number of static indirect call sites ("Ind.").

Target	Lang.	F	BB	Ind.
bloaty	C++	5,626	99,259	1,413
curl	C	7,040	69,772	1,677
freetype2	C	2,279	21,007	672
harfbuzz	C++	17,235	42,748	300
jsoncpp	C++	455	6,922	23
lcms	C	1,032	6,528	202
libpng	C	459	4,788	11
libxslt	C	2,227	36,379	2,163
mbedtls	C	1,851	14,311	99
openh264	C++	664	10,608	96
openssl	C	9,381	48,406	1,244
openthread	C++	7,497	25,302	158
re2	C++	4,011	10,841	49
sqlite3	C	2,265	33,105	247
stb	C	220	2,455	11
vorbis	C	339	3,802	33
woff2	C++	3,380	12,274	66
zlib	C	51	688	9

6.3.2 Static Analysis Performance (RQ 1)

As discussed in Chapter 1, static analyses have garnered a reputation for being more harmful than helpful, due to the many false positives/negatives they emit. Moreover, they often fail to scale to "real-world" programs. In Chapter 5 we developed a suite of static analyses to quantify control- and data-flow-coverage achieved by our DATAFLOW fuzzer. However, we did not investigate the accuracy of these analyses. Here, we empirically investigate the accuracy of modern static analyses in the context of fuzzing, focusing on the effectiveness of a static analysis in quantifying the upper bound of a fuzzer's state space search.

Table 6.1 summarizes the control-flow elements of the 23 FUZZBENCH coverage targets, while Table 6.2 summarizes the control-flow elements *reachable* from the target's entry point (i.e., elements that may be covered during fuzzing). We use the static analyses discussed in Section 6.2.3 to determine reachability, namely: Fuzz Introspector (commit `fe9c68b`), SVF (commit `a3fc9cc`), and SEADSA (commit `fda066e`). We also considered the `cclyzer++` pointer analysis framework [66] (commit `1a64af9`), but found that its analysis failed to complete within 24 h on most targets.

Table 6.2 shows wildly varying results across the three evaluated static analyses. In particular, SVF and SEADSA often report a greater number of reachable functions compared to Fuzz Introspector (e.g., up to $\sim$13$\times$ on the *woff2* target). We attribute

TABLE 6.2: Static analysis results, showing the number of functions ("F"), basic blocks ("BB"), edges ("E"), and context-sensitive edges ("E_{ctx}") reachable from the driver program. A ✗ indicates the analysis failed or did not complete (within 24 h).

Target	Fuzz Introspector				SVF				SEADSA			
	F	BB	E	E_{ctx}	F	BB	E	E_{ctx}	F	BB	E	E_{ctx}
bloaty	278	4,734	8,529	34,979	✗	✗	✗	✗	1,817	37,204	137,497	7,629,437
curl	1,327	14,694	33,323	133,601	✗	✗	✗	✗	✗	✗	✗	✗
freetype2	327	3,066	6,400	15,550	2,010	19,192	130,177	11,265,197	686	4,329	9,654	39,754
harfbuzz	10,521	28,851	87,695	259,811	12,132	32,833	101,924	4,186,525	10,250	27,897	82,532	353,143
jsoncpp	37	204	623	1,352	81	391	1,339	3,319	54	638	1,243	4,460
lcms	253	1,708	4,027	23,535	✗	✗	✗	✗	847	5,424	21,450	106,226
libpng	216	3,106	6,297	12,503	297	3,211	6,913	21,954	275	3,242	6,723	20,480
libxslt	477	7,453	15,095	112,247	1,669	28,732	102,445	8,690,806	1,538	28,389	71,749	4,575,223
mbedtls	816	6,790	17,783	70,045	1,135	8,462	27,626	168,129	1,094	8,338	21,937	102,192
openh264	21	64	141	191	761	10,364	24,342	109,202	611	10,366	19,971	101,145
openssl	1,535	7,494	20,635	166,769	4,698	29,835	176,491	26,363,048	✗	✗	✗	✗
openthread	5,917	20,622	67,193	261,847	3,239	8,805	26,401	186,350	5,680	20,489	63,866	981,359
re2	545	2,762	8,346	20,659	3,411	7,284	24,093	83,466	725	1,841	5,156	19,899
sqlite3	1,689	27,626	65,628	470,460	2,285	32,647	132,147	21,591,726	2,059	31,730	72,517	4,617,764
stb	158	2,186	4,891	15,636	190	2,262	5,615	24,659	161	2,255	5,115	17,992
vorbis	121	1,302	2,604	8,299	187	1,783	4,007	15,855	157	1,780	3,388	13,321
woff2	95	2,774	4,603	10,802	1,259	4,431	10,495	20,191	1,224	4,685	10,241	25,300
zlib	22	567	936	1,005	23	567	986	1,068	24	569	984	1,062

this increase to the more sophisticated pointer analysis techniques used by SVF and SEADSA, allowing these analyses to resolve more indirect calls.

Notably, there is no significant correlation between the number of indirect calls (listed in Table 6.1) and this increase in reachable functions. For example, *woff2* has only 66 indirect calls and a ∼13× increase in reachable functions. Instead, the complexity of the target's data flow determines how significantly the Fuzz Introspector and SVF/SEADSA results differ.

The number of reachable control-flow elements also varies between SVF and SEADSA. We attribute this to algorithmic and implementation differences. Notably, SVF failed to analyze more targets than SEADSA. While both failed to analyze *curl*, SEADSA failed to analyze *openssl* (while SVF's analysis completed) and SVF failed to analyze *bloaty* and *lcms* (which SEADSA's analysis completed). This—combined with `cclyzer++`'s inability to complete its analysis on *any* target within 24 h—suggests that scalability issues remain in even state-of-the-art static analysis frameworks.

Table 6.3 summarizes the number of control-flow elements covered by at least one fuzzer. Comparing these results to those in Table 6.2, we see that Fuzz Introspector often undercounts the number of reachable functions (e.g., reporting 278 reachable functions in *bloaty*, while 1,101 were covered at run time). Differences between the static and dynamic analysis results become more pronounced when comparing more sensitive coverage metrics; in particular, context-sensitive edges (E_{ctx} in Tables 6.2 and 6.3). This is unsurprising, as static analysis errors (false positives and false negatives) are compounded as the analysis becomes more sensitive. This is evident

TABLE 6.3: The number of control-flow elements covered. The control-flow elements are the same as those in Table 6.2: the number of functions ("F"), basic blocks ("BB"), edges ("E"), and context-sensitive edges ("E_{ctx}").

Target	F	BB	E	E_{ctx}
bloaty	1,101	5,571	8,762	25,944
curl	1,546	10,737	15,208	27,872
freetype2	1,016	9,199	13,588	37,820
harfbuzz	11,223	28,515	51,215	154,310
jsoncpp	113	664	903	2,173
lcms	283	1,484	2,122	5,610
libpng	162	1,789	2,441	4,686
libxslt	798	10,471	15,240	65,560
mbedtls	623	3,885	5,106	10,982
openh264	431	8,575	11,862	15,800
openssl	1,269	6,608	10,169	76,694
openthread	2,477	5,338	11,169	19,499
re2	580	3,049	4,441	7,381
sqlite3	1,762	21,755	32,931	374,597
stb	142	1,765	2,293	3,700
vorbis	131	1,280	1,728	2,753
woff2	101	1,798	2,449	2,934
zlib	20	388	547	554

in *bloaty*, where SEADSA reported 7,629,437 reachable context-sensitive edges, while only 25,944 were covered at run time. These differences (in context-sensitive edges) continue in *curl* (Fuzz Introspector), *harfbuzz* (SVF), *lcms* (SEADSA), *libxslt* (SEADSA), *mbedtls* (SVF), *openssl* (SVF), *openthread* (SEADSA), and *sqlite3* (SVF and SEADSA).

Finding

The number of reachable control-flow elements varies wildly (a) across the evaluated static analyses, and (b) compared to the number of covered control-flow elements. This suggests the static analysis results are unrealistic.

False Negatives

In Table 6.4 we summarize the accuracy of these static analysis results by examining *false negatives*: the number of control-flow elements the static analysis labeled unreachable but were reachable by a fuzzer. Quantifying false positives is undecidable; there is no way of knowing if a control-flow element (e.g., CFG edge) is unreachable or if the fuzzer failed to generate a valid input for covering that element. Because false negatives (and false positives) can only occur due to indirect function calls[3], we focus on the function call graph.

[3]Or bugs in the static analyzer, which we ignore.

TABLE 6.4: The number of static analysis *false negatives*; i.e., the number of indirect function calls that the static analysis (one of "Fuzz Introspector", "SVF", and "SEADSA") labeled unreachable but were covered by a fuzzer. The best performing static analyzer for each target is highlighted in green (lower is better). A ✗ indicates the static analysis failed.

Target	Fuzz Introspector	SVF	SEADSA
bloaty	2,155	✗	2,160
curl	451	✗	✗
freetype2	572	163	407
harfbuzz	285	62	94
jsoncpp	137	137	162
lcms	80	✗	2
libpng	14	6	6
libxslt	177	4	20
mbedtls	45	0	1
openh264	82	0	7
openssl	414	87	✗
openthread	29	20	125
re2	176	165	211
sqlite3	235	22	26
stb	8	0	0
vorbis	23	5	5
woff2	36	20	29
zlib	3	3	0

Despite failing on the most targets, SVF was the best performer with the lowest number of false negatives on the most targets. In comparison, Fuzz Introspector was the worst performer. Despite this—and its relatively straightforward approach to resolving indirect function calls (Section 6.2.3)—Fuzz Introspector outperforms SEADSA on four targets (*bloaty*, *jsoncpp*, *openthread*, and *re2*).

We analyzed these false negatives and attribute most of them to *custom memory allocators*. In Section 5.4.4 we discussed the prevalence and impact of custom memory allocators on DATAFLOW. In particular, custom memory allocators required user intervention to ensure *def-use* sites were appropriately instrumented. Here, custom memory (de)allocation routines are passed to functions inside structs. Dynamically (de)allocating memory thus requires making indirect function calls via struct offsets, which SVF and SEADSA are unable to reason about. We hypothesize that a "structure-sensitive" analysis—such as that used by `cclyzer++` [13, 66]—would perform better in these circumstances. Unfortunately, this increased accuracy incurs a significant performance penalty, often resulting in the analysis failing to complete (within 24 h).

Finding

The high variance in results across the three static analyzers, together with the number of false negatives, indicates that even state-of-the-art static analyzers are inadequate for quantifying the upper-bound of a fuzzer's state space search.

6.4 Discussion

Liyanage et al. [129] also investigated the use of static analysis to quantify *fuzzer effectiveness*; computing an upper bound on the number of reachable coverage elements.[4] Liyanage et al. [129] also found that static analysis significantly over-counted reachable coverage, and our findings reinforce their results. Consequently, we encourage static analysis developers to further improve the accuracy and scalability of their tools, particularly on "real-world" targets. In particular, this could include developing new benchmark suites that contain larger, more complex codebases more indicative of "real-world" software.

6.5 Future Work

Based on our results, we propose the following ideas for future work.

Root cause quantification for missing control-flow edges. In Section 6.3.2 we analyzed the accuracy of three static analyzers by quantifying the number of unlabeled call graph edges. While this analysis was manual, automatic techniques would help in improving the accuracy of these static analyzers. For example, Chakraborty et al. [30] develop a technique to "*automatically quantify the relative importance of different root causes of call graph unsoundness for a set of target [JavaScript] applications*". While their study focused on JavaScript call graphs, their techniques are generalizable to other languages (here, we are interested in LLVM intermediate representation (IR)), and could be used to improve the accuracy of LLVM-based static analyzers (which we found to be severely lacking).

Quantifying data flow. In Section 5.5.4 we quantified data-flow coverage by computing the set of *def-use* chains in a target. An alternative approach is to quantify data flow over *program variable values*. This approach is akin to the data abstractions used in abstract interpretation and model checking (discussed in Chapter 1). We

[4] Our work was carried out concurrently. Liyanage et al. [129] evaluated SVF on five targets from the Software-artifact Infrastructure Repository, whereas we evaluated a larger number of targets from FUZZBENCH and also evaluate Fuzz Introspector and SEADSA.

propose using a static analyzer (e.g., CLAM, a static analyzer for LLVM built on the CRAB library [79]) to compute inductive invariants—computed as basic block pre- and post-conditions—for each variable in $\mathcal{P}$. These invariants provide the range of values a variable can take. By instrumenting these variables, we can track their value at run time, providing another measure of "state space coverage". We leave the implementation and evaluation of such analyses for future work.

6.6 Chapter Summary

This chapter discussed the difficulties in quantifying a fuzzer's state space search. To address this, we proposed using context-sensitive static analyses to quantify the control-flow elements of a fuzzer's state space search. Unfortunately, our results also show that even state-of-the-art static analyses are far from accurate on typical fuzzer targets; they either failed to complete their analysis or produced unrealistic results. We hope these results and ideas for future work inspire others to improve the accuracy and scalability of static analyses for real-world codebases.

In the final chapter, we provide a comprehensive evaluation of the full range of coverage metrics discussed throughout this dissertation.

Chapter 7

Comparison of Coverage Metrics

This final chapter provides a comprehensive evaluation of the full range of coverage metrics discussed throughout this dissertation. To do this, we propose a methodology for fairly comparing coverage metrics.

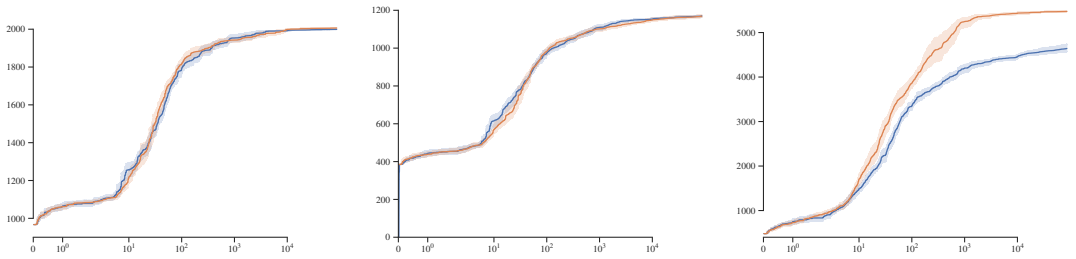
7.1 Introduction

As described in Chapter 3, fuzzers use a coverage metric to measure their progress in exploring a target's state space. This coverage metric abstracts the target's state space, leading the fuzzer to uncover new behaviors in the target (by revealing new states in the state space). Moreover, fuzzers are often evaluated based on their coverage of a target's state space: fuzzer f_1 is "better" than fuzzer f_2 if f_1 explores more of the target's state space than f_2. Popular fuzzer benchmarks (e.g., FUZZBENCH [144]) exemplify these ideas, relying solely on *aggregate control-flow coverage* accumulated over a fuzzing campaign. However, as we have repeatedly shown throughout this dissertation, this only covers a single dimension of a target's state space. How can we more-accurately characterize a fuzzer's state space search?

Chapter outline. We begin by demonstrating the weaknesses of current approaches to measuring and comparing fuzzers based on coverage profiles, motivating the need for alternative techniques (Section 7.2). We then use the dynamic analysis introduced in Chapter 6 to evaluate the performance of 15 coverage metrics—spanning the full range of control- and data-flow-based metrics discussed in Chapter 3 (Section 7.3).

7.2 Motivation

Coverage measures (e.g., number of lines of code) are commonly used to evaluate and compare a fuzzer's performance, serving as a proxy for a fuzzer's bug-finding



(A) Source. (B) Coarse-grained probabilistic edge. (C) Context-sensitive probabilistic edge.

FIGURE 7.1: Coverage exploration by AFL++ over ten repeated 24 h trials with two coverage metrics—coarse-grained probabilistic edge coverage and context-sensitive probabilistic edge coverage—on FUZZBENCH's *libpng*. The x-axis shows wall-clock time (log scale in seconds) and the y-axis shows the number of covered measurement points (source, edges, and context-sensitive edges in Figs. 7.1a to 7.1c, respectively). Each figure shows mean coverage and 95 % bootstrap confidence interval (CI).

ability (after all, you cannot find bugs in code never executed) [115]. Consequently, we can use coverage to reason about the residual risk of stopping a fuzzing campaign too early (i.e., while the fuzzer is still discovering new program behaviors) [20]. Unfortunately, reasoning about coverage as an approximation for exercised program behaviors is hard. We discuss the difficulties here.

Visualizing fuzzers' progress via plots showing coverage growth over time are a common sight in fuzzer evaluations [9, 16, 19, 21, 36, 37, 39, 40, 43, 46, 51, 62, 67, 68, 85, 90, 91, 100, 125, 133, 144, 157, 173, 174, 184, 186, 190, 198, 213, 217, 227, 229]. We continue this trend in Fig. 7.1, showing the coverage achieved by AFL++ on FUZZBENCH's *libpng* target with two coverage metrics: coarse-grained probabilistic edge coverage (AFL edge coverage, denoted $\mathcal{P}_{\text{AFL}}$) and context-sensitive probabilistic edge coverage (AFL edge coverage augmented with calling context, denoted $\mathcal{P}_{\text{ctx}}$).[1]

We generate these plots using a "record and replay" (R&R) approach. This approach allows us to view the progress made fuzzing $\mathcal{P}_{\text{AFL}}$ through the "point-of-view" (PoV) of $\mathcal{P}_{\text{ctx}}$ (and *vice versa*). R&R: (i) records $\mathcal{P}_{\text{AFL}}$'s current input at regular intervals (here, every 1,000 inputs), storing the results in $\mathcal{S}_{\text{AFL}}$; and then (ii) replays $\mathcal{S}_{\text{AFL}}$ through $\mathcal{P}_{\text{ctx}}$ (*post hoc*). This approach, proposed by Aschermann [6], ensures an unbiased comparison between coverage metrics. This is akin to the approach taken by FUZZTASTIC [127], which uses additional instrumentation to measure collision-free block coverage in parallel with the fuzzer (rather than examining the queue *post hoc*).

[1] The highlight color reflects the plot colors.

Figure 7.1a echoes the approach used by FUZZBENCH [144], where we replayed $\mathcal{S}_{\text{AFL}}$ and $\mathcal{S}_{\text{ctx}}$ through $\mathcal{P}_{\text{src}}$ instrumented with Clang's source-based coverage (operating on the Clang AST and preprocessor) [209]. FUZZBENCH uses Clang's source-based coverage as an independent code coverage metric to enable comparisons *across* fuzzers. While this approach removes bias toward a particular coverage metric, it also means differences between metrics—and their implementations—are lost (e.g., when comparing coverage achieved by fuzzers using more fine-grained metrics).

This "information loss" is evident in Fig. 7.1: viewed through the $\mathcal{P}_{\text{AFL}}$ and $\mathcal{P}_{\text{src}}$ PoV (Figs. 7.1a and 7.1b , respectively), the coverage achieved across the two campaigns appears identical. However, the $\mathcal{P}_{\text{ctx}}$ PoV (Fig. 7.1c) clearly shows that fuzzing $\mathcal{P}_{\text{ctx}}$ explores more of $\mathcal{P}$'s state space (due to context sensitivity). *How, then, should we make cross-fuzzer coverage comparisons?* The remainder of this chapter answers this question.

7.3 Evaluation

Our evaluation aims to answer the following research questions:

RQ 1 How do the coverage expansion capabilities of different control- and data-flow-based metrics compare? (Section 7.3.2)

RQ 2 Can coverage metrics be combined to complement each other, to uncover new states in the target's state space (and thus share this information with other fuzzers—in an ensemble configuration—to improve their state space search)? (Section 7.3.3)

7.3.1 Methodology

Fuzzer Selection

We evaluate the 15 coverage metrics listed in Table 7.1. These coverage metrics span the full range of coverage metrics discussed throughout this dissertation. All 15 are (re)implemented as LLVM (v14) passes in AFL++ (commit `7101192`). The (context-sensitive) edge metrics (E_{coarse}, E_{LTO}, E_{SANCOV}, $\text{E}_{\text{1-ctx}}$, $\text{E}_{\text{2-ctx}}$, and $\text{E}_{\text{3-ctx}}$) are described in Sections 3.2.2 and 3.2.4. The data dependency graph (DDG) (E_{DDG}), memory accesses (E_{MA}), and n-gram paths metrics ($\text{P}_{\text{2-gram}}$, $\text{P}_{\text{4-gram}}$, and $\text{P}_{\text{8-gram}}$) are described in Sections 3.2.3, 3.2.5 and 3.2.7, respectively. Finally, *def-use* chain coverage ($\text{DF}_{\text{A+S/A}}$, $\text{DF}_{\text{A+S/O}}$, and $\text{DF}_{\text{A+S/V}}$) is described in Chapter 5. We ported E_{MA} from its original QEMU-based implementation [214] to operate on the LLVM intermediate representation (IR). E_{DDG} uses `DDFuzz`'s implementation of the DDG [141], while $\text{DF}_{\text{A+S/A}}$,

TABLE 7.1: Evaluated fuzzers and their coverage metrics. B has no $\mathbb{C}$, while E_{LTO}, and E_{SANCOV} uses a variable-sized $\mathbb{C}$.

Fuzzer	Description	Map size (KiB)
B	Blackbox	–
E_{coarse}	Coarse-grain probabilistic edge	64
E_{LTO}	Link-time optimization (LTO) edge	–
E_{SANCOV}	SANCOV LTO fine-grain edge	–
$E_{1\text{-ctx}}$	Edge $\oplus$ calling context ($k = 1$)	64
$E_{2\text{-ctx}}$	Edge $\oplus$ calling context ($k = 2$)	64
$E_{3\text{-ctx}}$	Edge $\oplus$ calling context ($k = 3$)	64
E_{DDG}	Edge $\oplus$ DDG	64
E_{MA}	Edge $\oplus$ memory accesses (reads + writes)	128
$P_{2\text{-gram}}$	Path. 2-gram edge	64
$P_{4\text{-gram}}$	Path. 4-gram edge	64
$P_{8\text{-gram}}$	Path. 8-gram edge	64
$DF_{A+S/A}$	*Def-use* chain. Array and struct *def* $\oplus$ access *use* sites	1,024
$DF_{A+S/O}$	*Def-use* chain. Array and struct *def* $\oplus$ access *use* offsets	1,024
$DF_{A+S/V}$	*Def-use* chain. Array and struct *def* $\oplus$ accessed value *use* sites	1,024

$DF_{A+S/O}$, and $DF_{A+S/V}$ use DATAFLOW's implementation of *def-use* chain coverage [90]. The other fuzzers listed in Table 7.1 were already implemented in AFL++. Using the same fuzzer (AFL++) and instrumentation framework (LLVM) ensures a fair evaluation [118].

Target Selection

We ues the same configuration of FUZZBENCH [144] described in Section 6.3.1.

Experimental Setup

We run one fuzzing *campaign* per {target $\times$ coverage metric} combination. Each campaign consists of ten independent 24 h *trials* (consistent with the recommendations by Klees et al. [115]). This amounts to over 8 CPU-yr of fuzzing. All experiments were carried out using the configuration described in Section 6.3.1. Additionally, we modify FUZZBENCH to support our R&R approach (Section 7.2). R&R ensures: (i) an unbiased comparison of coverage metrics, because inputs have not been retained based on a specific coverage metric; (ii) differences between metrics (and their implementations) are retained; and (iii) the sensitivity of metrics can be compared.

7.3.2 Comparison of Coverage Metrics (RQ 1)

We use our dynamic analysis (described in Section 6.2.2) to compare the coverage expansion (and thus state space search) performance of different control- and data-flow-based coverage metrics. Given the unreliability of our static analysis results

(Section 6.3), we do not quantify coverage with respect to a static interprocedural control-flow graph (ICFG) (as we did in Section 5.5.4). Instead, Table 7.2 summarizes the number of context (in)sensitive edges covered by AFL++ using the 15 coverage metrics listed in Table 7.1. The mean coverage over ten repeated 24 h trials with 95 % CI is shown. We use the Mann-Whitney U-test [139] to statistically compare coverage across fuzzers: two fuzzers cover the same number of context-(in)sensitive edges if the Mann-Whitney U-test's p-value > 0.05. We first compare performance using context-insensitive edge coverage, before turning our attention to context-sensitive edge coverage.

Context-Insensitive Edge Coverage

Context-insensitive edge coverage is the approach traditionally used to compare fuzzer performance. We use this approach in Table 7.2a to summarize and compare the aggregate coverage achieved by each coverage metric. Based on these results, E_{LTO} and E_{SanCov} were the best-performing metrics, achieving the most coverage on the most targets. This echoes our result in Chapter 5. Here, both E_{LTO} and E_{SanCov} consistently outperformed the other edge metric (E_{coarse}), reinforcing the importance of eliminating hash collisions in the coverage map. Notably, E_{SanCov}'s higher runtime cost (compared to E_{LTO}) did not translate to better performance.

Similar to our results in Section 7.2, adding context sensitivity to coarse-grained probabilistic edge coverage ($E_{\text{1-ctx}}$, $E_{\text{2-ctx}}$, and $E_{\text{3-ctx}}$) failed to meaningfully improve overall coverage. Notably, the performance of the context-sensitive edge coverage metrics improved slightly (mean 4 % increase in edge coverage across all targets) as the number of calling contexts increased (from $k = 1$ to $k = 3$). However, even when $k = 3$, the two LTO edge metrics generally outperformed the context-sensitive edge metrics. We attribute this to hash collisions in the coverage map, due to the large number of context-sensitive edges in the target program (per the results in Table 6.4).

Approximating path coverage with n-gram coverage ($P_{\text{2-gram}}$, $P_{\text{4-gram}}$, and $P_{\text{8-gram}}$) produced similar results to context-sensitive edge coverage, with little statistically-significant improvement over the two LTO edge metrics. Unlike the context-sensitive edge metrics, average performance worsened as n increased (mean 10 % decrease in edge coverage across all targets). Intuitively, we expected targets with fewer basic blocks to perform better, because they would be less prone to hash collisions in the coverage map. However, this intuition was not consistently borne out in our results. While it may explain the coverage increase—as n increased—in *zlib* (the target with the fewest number of basic blocks, per Table 6.1), it does not explain the results for

TABLE 7.2: Context-(in)sensitive edge coverage achieved by each fuzzer listed in Table 7.1 on FUZZBENCH. The mean over ten repeated 24 h trials with 95 % bootstrap CI is reported. The best performing fuzzer for each target is highlighted in green (larger is better).

(A) Context-*insensitive* edge coverage.

Target	B	E_{coarse}	$E_{1\text{-ctx}}$	$E_{2\text{-ctx}}$	$E_{3\text{-ctx}}$	E_{LTO}	E_{SANCov}	E_{MA}	E_{DDG}	$P_{2\text{-gram}}$	$P_{4\text{-gram}}$	$P_{8\text{-gram}}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
bloaty	5,271.00 ± 253.35	7,221.60 ± 82.61	7,317.67 ± 81.11	7,580.30 ± 37.10	7,532.80 ± 63.10	7,650.40 ± 59.75	7,733.30 ± 90.25	5,370.70 ± 155.80	7,853.50 ± 57.15	7,102.10 ± 70.05	7,313.20 ± 53.55	5,617.80 ± 235.00	5,351.10 ± 244.71	4,680.70 ± 296.96	5,787.40 ± 212.70
curl	9,294.90 ± 578.66	14,050.20 ± 52.05	14,217.80 ± 77.50	14,253.20 ± 60.75	14,328.70 ± 50.75	14,195.80 ± 53.90	14,186.80 ± 48.70	11,762.10 ± 49.05	14,275.80 ± 64.40	14,069.40 ± 99.55	14,359.20 ± 40.55	13,706.60 ± 36.55	9,641.00 ± 649.60	7,095.20 ± 794.52	9,929.60 ± 685.03
freetype2	6,212.50 ± 53.35	9,825.90 ± 55.30	9,789.20 ± 67.45	9,725.70 ± 66.15	9,811.90 ± 51.55	11,674.30 ± 177.10	11,721.60 ± 80.00	6,463.50 ± 87.85	10,343.00 ± 36.15	9,760.30 ± 41.45	9,926.20 ± 36.95	9,253.00 ± 82.85	6,375.50 ± 40.70	6,023.50 ± 74.30	6,606.80 ± 39.90
harfbuzz	22,877.00 ± 2,546.79	47,750.60 ± 306.01	43,029.70 ± 434.36	44,572.40 ± 3,609.96	45,808.80 ± 253.80	49,153.50 ± 154.15	49,252.00 ± 165.20	32,619.10 ± 915.91	48,644.00 ± 195.66	47,850.70 ± 202.85	47,031.90 ± 297.41	40,874.60 ± 776.97	25,176.30 ± 2,297.66	21,330.80 ± 2,721.77	21,114.80 ± 1,891.92
jsoncpp	589.10 ± 10.50	899.40 ± 0.70	900.40 ± 0.55	899.80 ± 0.60	900.60 ± 0.50	902.80 ± 0.25	901.70 ± 0.85	677.60 ± 8.80	899.67 ± 0.89	900.00 ± 1.20	901.00 ± 0.00	877.11 ± 3.56	604.30 ± 7.20	566.20 ± 23.85	635.60 ± 6.75
lcms	474.50 ± 177.45	1,282.50 ± 193.66	1,299.70 ± 187.40	1,318.90 ± 211.15	1,577.60 ± 260.00	1,403.90 ± 226.05	1,262.60 ± 189.75	812.40 ± 38.00	1,421.30 ± 201.86	1,369.50 ± 260.00	1,119.10 ± 25.05	1,159.30 ± 187.05	463.20 ± 162.60	232.70 ± 135.35	644.60 ± 102.25
libpng	2,024.20 ± 7.10	2,434.20 ± 1.50	2,437.70 ± 0.95	2,436.60 ± 1.20	2,437.70 ± 0.50	2,436.20 ± 1.80	2,435.40 ± 1.60	2,170.20 ± 18.40	2,437.50 ± 1.50	2,434.80 ± 1.95	2,439.50 ± 0.60	2,432.50 ± 2.50	2,064.70 ± 9.80	1,975.80 ± 36.70	2,098.80 ± 14.70
libxslt	6,225.10 ± 85.10	12,470.90 ± 128.70	8,749.80 ± 139.51	12,588.90 ± 366.46	12,588.90 ± 366.46	12,789.20 ± 171.96	12,724.50 ± 112.35	9,106.10 ± 105.55	12,985.00 ± 100.00	12,596.30 ± 98.40	12,823.60 ± 104.45	11,599.30 ± 312.81	6,915.50 ± 191.95	5,970.60 ± 273.76	7,608.40 ± 113.80
mbedtls	3,146.50 ± 10.75	3,622.30 ± 16.50	3,613.80 ± 25.20	3,609.90 ± 24.85	3,648.10 ± 19.55	3,680.00 ± 28.30	3,664.30 ± 31.45	3,183.80 ± 12.50	3,773.80 ± 195.20	3,604.70 ± 20.10	3,646.70 ± 22.20	3,473.10 ± 30.55	3,208.40 ± 20.90	3,112.00 ± 27.05	3,250.30 ± 16.60
openh264	11,368.67 ± 33.44	11,501.22 ± 50.50	11,408.30 ± 121.05	11,525.30 ± 62.15	11,493.60 ± 33.30	11,730.60 ± 31.85	11,674.10 ± 51.60	10,817.00 ± 72.30	11,663.20 ± 33.20	11,588.30 ± 31.90	11,408.30 ± 67.50	10,086.70 ± 304.06	10,848.20 ± 45.00	10,606.60 ± 100.00	10,962.80 ± 38.35
openssl	8,393.90 ± 101.80	9,832.10 ± 43.15	9,900.80 ± 45.40	9,917.60 ± 31.85	9,984.40 ± 30.90	9,939.60 ± 23.35	9,944.00 ± 23.95	8,570.30 ± 123.65	9,974.10 ± 42.30	9,874.20 ± 19.70	9,993.90 ± 32.95	9,905.20 ± 41.90	8,776.22 ± 100.50	8,276.60 ± 241.50	9,282.70 ± 83.90
openthread	9,578.00 ± 0.00	10,589.50 ± 5.35	10,620.50 ± 46.25	10,595.30 ± 5.35	10,599.30 ± 5.10	10,605.50 ± 4.65	10,599.00 ± 4.85	9,587.20 ± 13.80	10,625.70 ± 46.60	10,586.80 ± 5.10	10,684.00 ± 86.20	10,900.80 ± 84.30	9,578.00 ± 0.00	9,578.00 ± 0.00	9,578.00 ± 0.00
re2	2,928.70 ± 53.65	4,400.20 ± 14.85	4,402.30 ± 16.20	4,402.40 ± 13.95	4,408.70 ± 16.10	4,417.50 ± 13.25	4,416.50 ± 14.00	3,500.80 ± 30.15	4,421.56 ± 15.06	4,401.80 ± 17.05	4,417.30 ± 15.55	4,392.00 ± 17.65	3,156.50 ± 36.75	2,795.80 ± 104.15	3,235.90 ± 43.40
sqlite3	13,258.80 ± 70.45	22,948.70 ± 503.68	19,798.20 ± 276.95	22,503.70 ± 429.10	22,129.10 ± 305.15	24,582.20 ± 489.52	24,049.20 ± 640.90	11,456.00 ± 621.80	24,868.30 ± 610.33	22,988.20 ± 411.30	22,767.10 ± 590.81	18,184.40 ± 125.85	13,006.80 ± 667.16	12,378.80 ± 712.71	14,094.60 ± 261.20
stb	1,915.10 ± 17.60	2,240.70 ± 7.35	2,238.60 ± 8.30	2,242.00 ± 6.70	2,252.10 ± 9.25	2,263.90 ± 7.65	2,264.30 ± 8.15	2,016.20 ± 15.45	2,269.89 ± 7.06	2,247.20 ± 8.10	2,234.70 ± 7.45	2,182.10 ± 9.75	1,954.30 ± 20.45	1,819.30 ± 36.85	2,016.80 ± 8.75
vorbis	1,379.00 ± 35.35	1,711.60 ± 1.80	1,707.50 ± 6.30	1,711.70 ± 0.65	1,714.20 ± 1.20	1,715.50 ± 0.80	1,716.20 ± 1.05	1,384.60 ± 26.10	1,712.50 ± 0.70	1,711.00 ± 1.25	1,713.70 ± 2.20	1,714.50 ± 0.95	1,424.20 ± 39.60	1,364.20 ± 40.80	1,392.70 ± 20.65
woff2	1,648.10 ± 13.30	2,254.10 ± 17.65	2,280.80 ± 13.10	2,241.50 ± 33.80	2,288.20 ± 11.40	2,286.60 ± 20.30	2,275.70 ± 26.75	1,921.70 ± 6.40	2,274.90 ± 32.70	2,229.10 ± 26.80	2,303.50 ± 15.05	2,246.70 ± 17.05	1,688.90 ± 19.15	1,614.70 ± 36.45	1,748.20 ± 16.80
zlib	143.00 ± 10.05	530.10 ± 3.55	528.30 ± 2.60	531.00 ± 3.85	532.60 ± 3.10	528.80 ± 3.60	531.50 ± 2.85	483.90 ± 3.55	528.20 ± 3.70	530.10 ± 3.65	532.30 ± 5.40	537.80 ± 4.35	165.70 ± 13.85	157.70 ± 18.70	191.70 ± 20.90

(B) Context-*sensitive* edge coverage.

Target	B	E_{coarse}	$E_{1\text{-ctx}}$	$E_{2\text{-ctx}}$	$E_{3\text{-ctx}}$	E_{LTO}	E_{SANCov}	E_{MA}	E_{DDG}	$P_{2\text{-gram}}$	$P_{4\text{-gram}}$	$P_{8\text{-gram}}$	$DF_{A+S/A}$	$DF_{A+S/O}$	$DF_{A+S/V}$
bloaty	12,515.40 ± 430.80	16,826.30 ± 816.81	19,283.67 ± 1,048.35	19,326.00 ± 1,023.07	19,954.60 ± 852.96	18,271.70 ± 554.20	19,187.10 ± 991.70	12,306.70 ± 518.40	20,725.30 ± 979.16	16,034.30 ± 308.70	17,637.80 ± 715.11	12,768.60 ± 514.56	12,395.00 ± 403.40	11,231.70 ± 541.91	13,425.70 ± 516.45
curl	14,768.50 ± 1,044.56	22,481.20 ± 241.30	22,793.90 ± 329.01	22,817.50 ± 202.65	22,870.80 ± 263.95	22,456.50 ± 229.00	22,858.70 ± 130.46	18,645.70 ± 142.61	22,862.20 ± 174.55	22,448.40 ± 267.70	22,652.10 ± 288.81	21,913.20 ± 193.91	15,308.00 ± 1,077.06	11,107.00 ± 1,282.82	15,789.00 ± 1,162.02
freetype2	14,495.60 ± 421.11	41,204.50 ± 3,652.56	42,689.00 ± 3,584.67	40,659.20 ± 2,144.93	43,840.20 ± 3,863.51	52,281.90 ± 8,342.98	58,090.30 ± 11,348.50	15,226.00 ± 1,585.02	43,522.30 ± 3,653.18	37,455.30 ± 2,213.51	40,851.20 ± 3,270.42	31,830.10 ± 3,511.56	15,889.00 ± 1,239.77	14,466.50 ± 631.51	15,915.00 ± 1,238.70
harfbuzz	54,220.00 ± 7,832.88	144,539.10 ± 2,610.92	131,348.70 ± 2,512.26	130,894.20 ± 14,959.24	140,415.30 ± 1,820.59	149,048.00 ± 2,103.51	149,926.20 ± 1,794.35	80,868.90 ± 2,446.32	147,619.70 ± 1,213.20	140,706.50 ± 3,516.80	140,851.40 ± 1,828.26	109,356.50 ± 3,980.79	59,640.30 ± 6,966.79	48,588.10 ± 8,351.96	47,840.60 ± 5,316.62
jsoncpp	1,037.50 ± 42.05	1,778.40 ± 54.15	1,758.90 ± 38.00	1,789.00 ± 35.60	1,805.90 ± 71.45	1,769.90 ± 52.00	1,752.80 ± 36.30	1,239.80 ± 34.45	1,798.33 ± 84.33	1,791.80 ± 63.60	1,816.10 ± 60.02	1,725.78 ± 46.33	1,081.00 ± 24.40	979.80 ± 41.20	1,160.10 ± 45.45
lcms	744.90 ± 310.55	2,481.30 ± 599.76	2,457.60 ± 552.60	2,464.30 ± 593.40	3,272.70 ± 764.95	2,725.90 ± 637.51	2,402.00 ± 569.12	1,385.80 ± 118.80	2,751.10 ± 630.36	2,704.40 ± 736.51	1,965.20 ± 45.75	2,148.20 ± 459.15	724.90 ± 274.25	339.70 ± 227.00	1,040.80 ± 175.95
libpng	3,158.30 ± 70.20	3,649.50 ± 57.40	3,691.20 ± 47.20	3,671.80 ± 65.80	3,661.40 ± 62.60	3,619.90 ± 68.15	3,675.10 ± 50.65	3,238.50 ± 46.00	3,659.30 ± 43.85	3,649.10 ± 52.70	3,688.00 ± 63.10	3,662.40 ± 59.60	3,135.30 ± 74.01	3,076.20 ± 55.40	3,233.20 ± 67.55
libxslt	13,688.40 ± 346.90	37,747.60 ± 533.61	23,815.90 ± 484.61	37,741.00 ± 1,781.63	37,741.00 ± 1,781.63	39,278.30 ± 647.87	40,586.20 ± 2,045.31	23,767.40 ± 258.51	39,581.00 ± 250.00	38,313.00 ± 381.20	39,288.00 ± 295.00	34,649.50 ± 1,612.58	16,228.10 ± 744.81	13,030.30 ± 976.98	18,570.30 ± 499.31
mbedtls	5,093.00 ± 48.65	5,775.30 ± 46.90	5,735.10 ± 58.40	5,685.60 ± 77.85	5,758.80 ± 69.05	5,865.60 ± 114.10	5,866.00 ± 80.80	5,128.30 ± 40.30	6,225.90 ± 670.75	5,654.40 ± 64.30	5,909.40 ± 214.50	5,552.10 ± 80.70	5,145.40 ± 50.20	5,005.20 ± 61.05	5,236.80 ± 58.25
openh264	15,181.00 ± 71.61	15,381.00 ± 65.95	15,334.10 ± 139.65	15,433.10 ± 121.80	15,305.70 ± 31.85	15,704.30 ± 74.40	15,610.40 ± 94.40	14,658.90 ± 115.60	15,581.40 ± 63.75	15,493.10 ± 76.50	15,297.20 ± 117.05	13,757.20 ± 389.90	14,641.60 ± 101.90	14,367.90 ± 143.80	14,804.30 ± 79.70
openssl	57,944.40 ± 4,550.31	73,358.00 ± 1,103.77	75,393.10 ± 2,290.17	77,816.90 ± 2,843.81	78,926.90 ± 2,855.50	74,206.70 ± 1,746.47	75,264.00 ± 2,014.01	63,186.90 ± 4,820.97	76,192.40 ± 2,435.92	75,260.90 ± 1,886.64	75,926.40 ± 1,945.57	77,574.90 ± 3,170.08	62,945.11 ± 5,844.85	60,613.60 ± 6,141.26	70,405.20 ± 3,161.29
openthread	15,734.80 ± 52.05	18,984.50 ± 48.95	19,079.20 ± 86.25	18,973.50 ± 75.10	19,000.60 ± 74.00	19,076.30 ± 60.15	19,009.90 ± 80.10	15,888.70 ± 217.11	19,018.80 ± 88.35	19,136.50 ± 95.30	19,200.90 ± 125.50	19,417.10 ± 141.56	15,747.80 ± 94.20	15,769.80 ± 72.80	15,696.10 ± 84.65
re2	3,878.40 ± 129.40	6,749.60 ± 34.70	6,751.20 ± 49.90	6,751.20 ± 47.50	6,778.20 ± 30.05	6,773.50 ± 20.65	6,806.80 ± 29.85	5,388.10 ± 50.60	6,789.56 ± 31.50	6,744.30 ± 43.75	6,802.80 ± 28.75	6,735.40 ± 43.90	4,435.70 ± 140.30	3,614.90 ± 229.90	4,587.40 ± 138.30
sqlite3	85,343.10 ± 14,102.80	304,982.50 ± 22,719.52	258,641.40 ± 9,111.55	315,294.70 ± 24,699.64	293,307.30 ± 14,325.18	333,631.30 ± 22,847.24	349,276.10 ± 17,181.11	97,799.40 ± 11,073.08	323,894.70 ± 23,675.76	299,908.10 ± 24,424.82	319,245.90 ± 22,450.66	229,712.20 ± 11,425.58	93,745.20 ± 15,078.69	76,701.40 ± 12,163.11	110,089.80 ± 15,008.93
stb	2,906.30 ± 22.15	3,343.60 ± 12.25	3,347.40 ± 6.70	3,349.60 ± 7.35	3,361.80 ± 11.70	3,372.80 ± 10.55	3,372.40 ± 12.80	3,049.80 ± 20.50	3,380.44 ± 10.39	3,357.50 ± 7.70	3,341.70 ± 8.10	3,253.20 ± 16.20	2,956.70 ± 23.65	2,776.10 ± 45.85	3,042.40 ± 8.35
vorbis	1,831.20 ± 63.40	2,408.90 ± 45.90	2,429.90 ± 42.10	2,478.60 ± 16.70	2,468.50 ± 27.80	2,457.40 ± 27.70	2,430.90 ± 40.90	1,835.40 ± 52.25	2,440.30 ± 44.05	2,426.40 ± 46.95	2,455.20 ± 31.90	2,423.00 ± 45.05	1,914.70 ± 68.15	1,803.80 ± 69.65	1,851.60 ± 42.15
woff2	1,735.00 ± 15.10	2,363.60 ± 17.65	2,391.00 ± 13.20	2,351.90 ± 33.35	2,398.60 ± 11.45	2,400.90 ± 22.50	2,397.40 ± 30.95	2,026.10 ± 6.65	2,385.50 ± 33.00	2,338.80 ± 26.50	2,413.70 ± 14.80	2,356.90 ± 17.15	1,783.60 ± 20.10	1,701.00 ± 41.55	1,849.50 ± 16.50
zlib	150.00 ± 9.85	537.10 ± 3.65	535.30 ± 2.65	538.00 ± 3.75	539.60 ± 3.15	535.80 ± 3.60	538.50 ± 2.90	490.90 ± 3.50	535.20 ± 3.70	537.10 ± 3.50	539.30 ± 5.40	544.80 ± 4.40	172.70 ± 13.65	164.70 ± 18.60	198.70 ± 21.05

other targets (e.g., $P_{8\text{-gram}}$ was the best performer on *openthread*, despite it having a higher-than-average basic block count).

The E_{MA} and E_{DDG} metrics both attempt to incorporate data flow information with traditional edge coverage techniques. The approach used by E_{MA}—to xor memory accesses with coarse-grained probabilistic edge coverage—is the less effective of the two metrics. The large number of instrumented memory accesses significantly (a) reduces AFL++'s iteration rate, and (b) increases the size of the fuzzer's queue, resulting in *queue explosion* and making it harder for the fuzzer's scheduler to select a suitable seed to fuzz. Ultimately, this makes E_{MA} one of the worst-performing metrics. In contrast, E_{DDG}—which combines a DDG with coarse-grained probabilistic edge coverage—was the (equal) second-best-performing metric, achieving the (equal) highest coverage on ten targets. Unlike E_{MA}, E_{DDG} optimizes the inclusion of data flow information, minimizing run-time overheads and increasing coverage expansion. The E_{DDG} results reinforce one of our key findings from Section 5.5.3: fuzzers guided by control flow and data flow should be combined to maximize fuzzing outcomes.

Unsurprisingly, the worst results occurred when no coverage feedback was used (B). In extreme cases (e.g., *harfbuzz*, *lcms*), this resulted in half the coverage achieved by the control-flow-based metrics. However, to our dismay, *def-use* chain coverage ($DF_{A+S/A}$, $DF_{A+S/O}$, and $DF_{A+S/V}$) performed only marginally better than no coverage at all. Similar to our results in Section 5.5, $DF_{A+S/O}$ was the worst-performing of the three *def-use* chain metrics. Despite $DF_{A+S/V}$'s increased sensitivity (per Section 5.3.1), there was no statistically significant improvement over $DF_{A+S/A}$. Again, the run-time overheads associated with tracking *def-use* chains had a significant impact on performance.

Finding

Context-insensitive edge coverage is the traditional approach for measuring and comparing fuzzers' state space search. Based on this approach, E_{LTO} and E_{SanCov} were the best-performing coverage metrics. We attribute this result to their (a) low run-time overhead (leading to faster iteration rates), and (b) lack of hash collisions in the coverage map.

Context-Sensitive Edge Coverage

Section 6.2.1 introduced our approach for measuring *context-sensitive* control-flow coverage. Such an approach is important for uniquely identifying execution points. Similar to Table 7.2a, Table 7.2b summarizes aggregate coverage. Importantly, the

previous context-insensitive results are reflected here, with E_{LTO}, E_{SANCOV}, and E_{DDG} once again the best performers, achieving the highest coverage on the most targets. However, many of the other metrics—in particular, $E_{1\text{-ctx}}$, $E_{2\text{-ctx}}$, $E_{3\text{-ctx}}$, and $P_{2\text{-gram}}$—appeared to perform better when accounting for context sensitivity.

Following the results in Section 7.2 (particularly in Fig. 7.1), the improvements made by the context-sensitive edge and n-gram coverage metrics are not surprising. While n-gram coverage does not explicitly incorporate calling context, recording the previous n basic blocks implictly captures calling context at function call (and return) sites. Once again, B, E_{MA}, $DF_{A+S/A}$, $DF_{A+S/O}$, and $DF_{A+S/V}$ were the worst performers.

Given our results, it is intuitive to assume that "the best performing metric is the one used to compare metrics". For example, when comparing fuzzers' ability to expand edge coverage, the best performers used traditional edge coverage as their coverage metric (Section 7.3.2). Similarly, when comparing fuzzers' ability to expand context-sensitive edge coverage, the best performers incorporated context sensitivity in their coverage metric. However, our previous results in Chapter 5 caution against such thinking. In Section 5.5.4, edge coverage outperformed data-flow coverage when comparing performance using a data-flow-based metric (in this case, *def-use* chains). If anything, our results here and in Chapter 5 lead us to conclude that "the best performing metric is one based on control-flow".

Finding

Our context-sensitive results largely mirror our context-*insensitive* results: traditional edge coverage (as implemented in E_{LTO} and E_{SANCOV}) is the best-performing coverage metric. While adding context sensitivity is important for uniquely identifying execution points in a target, this ability is not strictly required when comparing fuzzer performance (based on coverage expansion). However, it may still be useful in other circumstances (e.g., to understand fuzz blockers).

7.3.3 Complementary Metrics (RQ 2)

Güler et al. [78] introduced CUPID, a system for automatically identifying fuzzers that complement each other when executed collaboratively in an ensemble configuration. CUPID is based on the intuition that fuzzers covering different parts of a target's state space should benefit from sharing their progress with other fuzzers. Importantly, CUPID is not simply based on which individual fuzzers achieve the

"best" coverage. Instead, CUPID "*measures the degree in which multiple fuzzers are complementary...[computing] the union of the expected mean code coverage*".

We adopt CUPID's approach to understand how well the 15 coverage metrics in Table 7.1 complement each other, and how much (or little) their state space search overlaps. To do this, we compute a *probability map* of how often a fuzzer was able to cover each control-flow graph (CFG) edge in a 24 h campaign (across ten independent trials). Using this probability map, we can understand (a) which coverage metrics uncover program states that other metrics do not, and (b) which coverage metrics may benefit from an ensemble fuzzing configuration.

Figures 7.2 and 7.3 illustrate the probability maps for context-insensitive and context-sensitive CFG edges (respectively) in *lcms*, while Fig. 7.4 illustrates the probability maps for context-insensitive CFG edges in *sqlite3* (the context-sensitive version contains too many edges to visualize).[2] Each cell represents a single edge in the context-insensitive CFG, while the color of a cell represents the probability of covering that edge: darker cells correspond to high probabilities. While it is difficult to discern the differences in individual edges (especially as the number of edges increases, becoming even more pronounced when context sensitivity is taken into account), these illustrations help visualize high-level differences across the 15 coverage metrics. In particular, the number of red cells in Figs. 7.2 and 7.3 reinforces the (relative) poor performance of B, E_{MA}, $DF_{A+S/A}$, $DF_{A+S/O}$, and $DF_{A+S/V}$.

We use these probability maps to determine how much a given coverage metric complements another (when combined in an ensemble fuzzing configuration). As described by Güler et al. [78], the combined probability of a set of fuzzers $\mathcal{F}$ covering a given CFG edge is calculated as

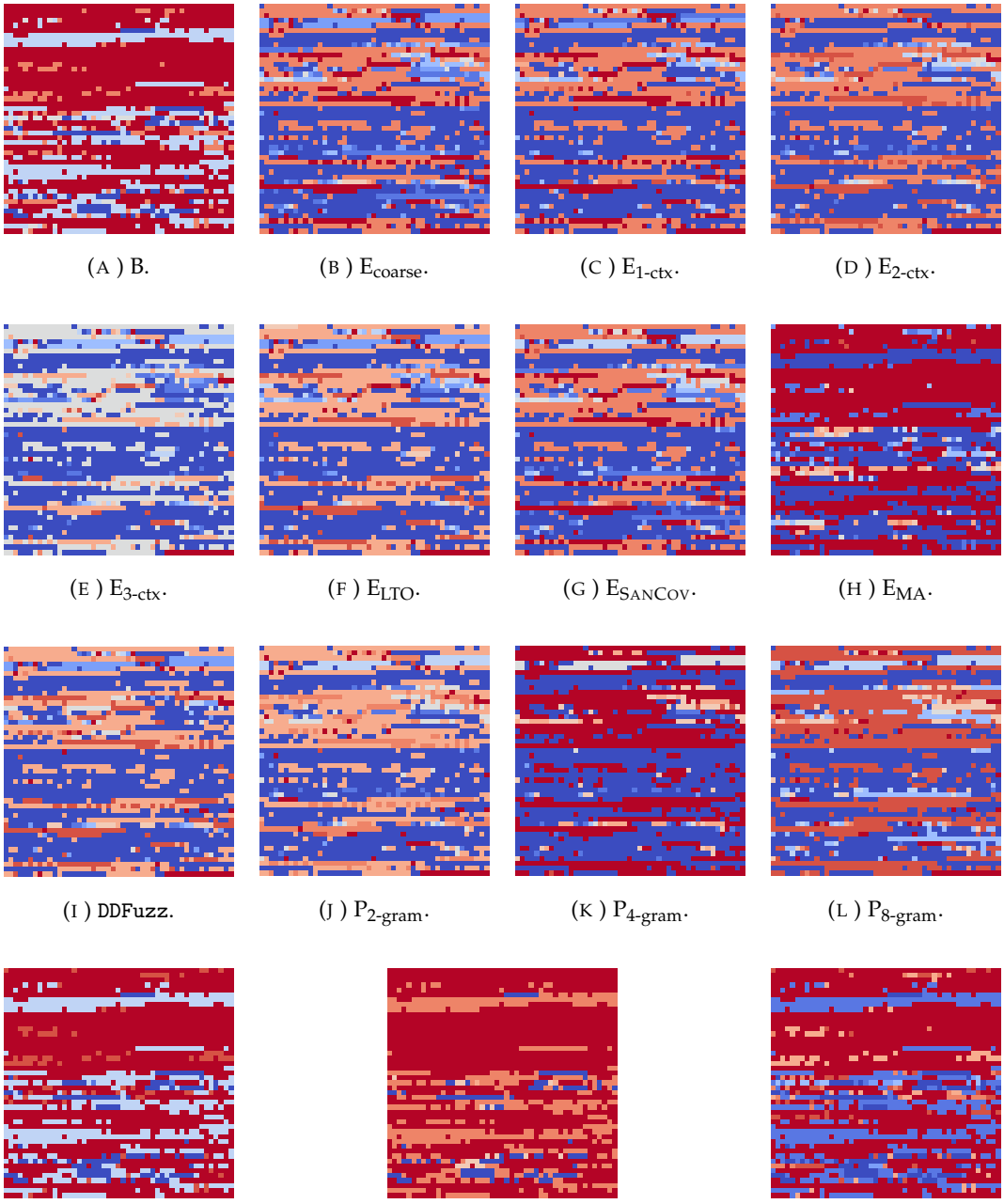
$$P_{\mathcal{F}}(e) = 1 - \prod_{f \in \mathcal{F}} \left(1 - P_f(e)\right), \tag{7.1}$$

where $\mathcal{F}$ is the set of fuzzers, e is the CFG edge the combined probability is being calculated for, and P_f is the probability of fuzzer f covering edge e.

Using Eq. (7.1), the sum of all edge probabilities can be calculated to determine how well the fuzzers in $\mathcal{F}$ complement each other. This is calculated as

$$\mathbb{E}_{\mathcal{F}} = \sum_{e \in E} P_{\mathcal{F}}(e), \tag{7.2}$$

[2] We select these targets for illustration because they displayed the highest variance across coverage metrics in Tables 7.2a and 7.2b.



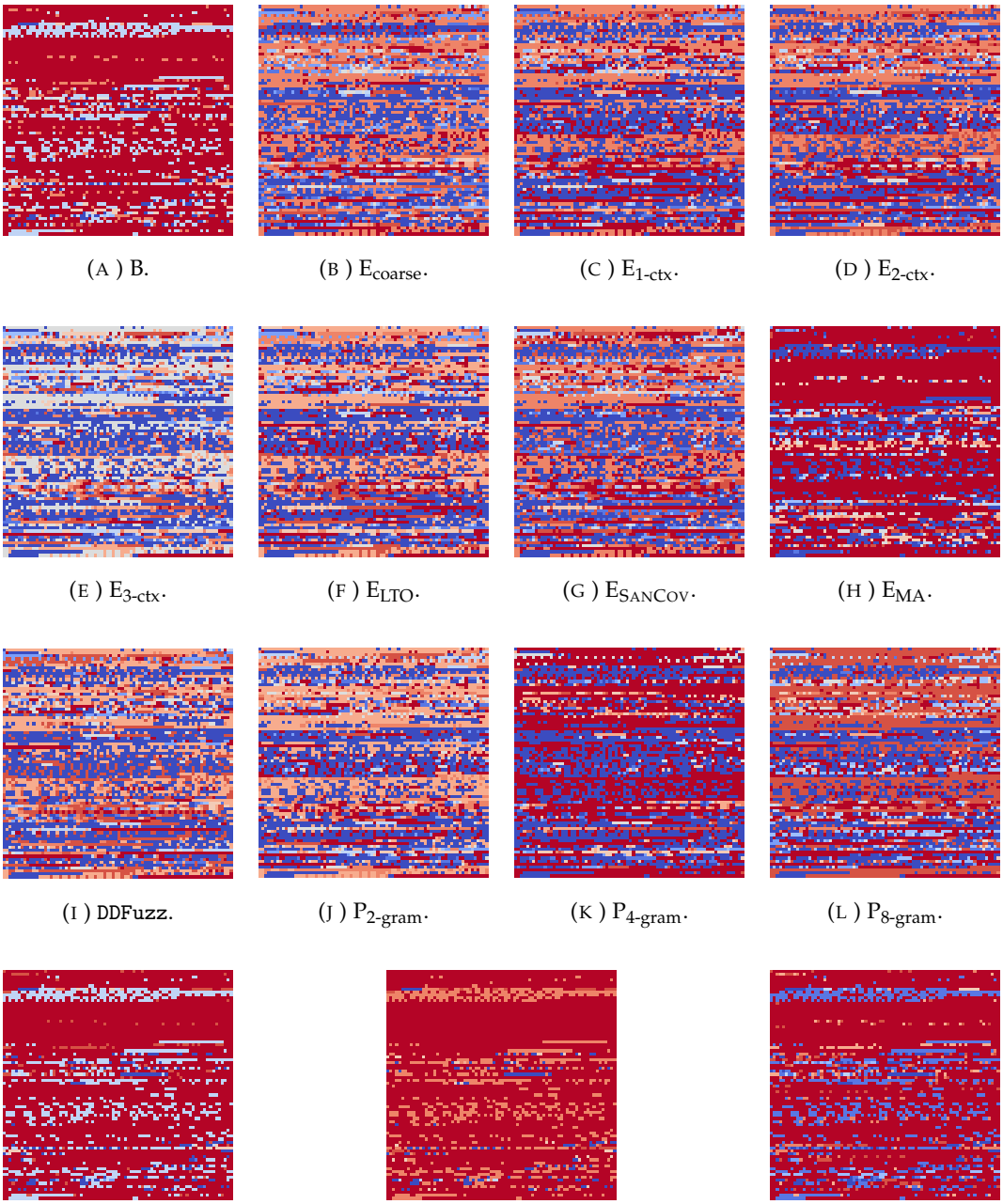
(A) B. (B) E_{coarse}. (C) $E_{1\text{-ctx}}$. (D) $E_{2\text{-ctx}}$.

(E) $E_{3\text{-ctx}}$. (F) E_{LTO}. (G) E_{SANCov}. (H) E_{MA}.

(I) `DDFuzz`. (J) $P_{2\text{-gram}}$. (K) $P_{4\text{-gram}}$. (L) $P_{8\text{-gram}}$.

(M) $DF_{A+S/A}$. (N) $DF_{A+S/O}$. (O) $DF_{A+S/V}$.

FIGURE 7.2: Probability maps for *lcms*. Each cell represents a single edge in the context-*insensitive* CFG. The color of a cell represents the probability that this edge will be covered by the given coverage metric (red is probability = 0, blue is probability = 1).



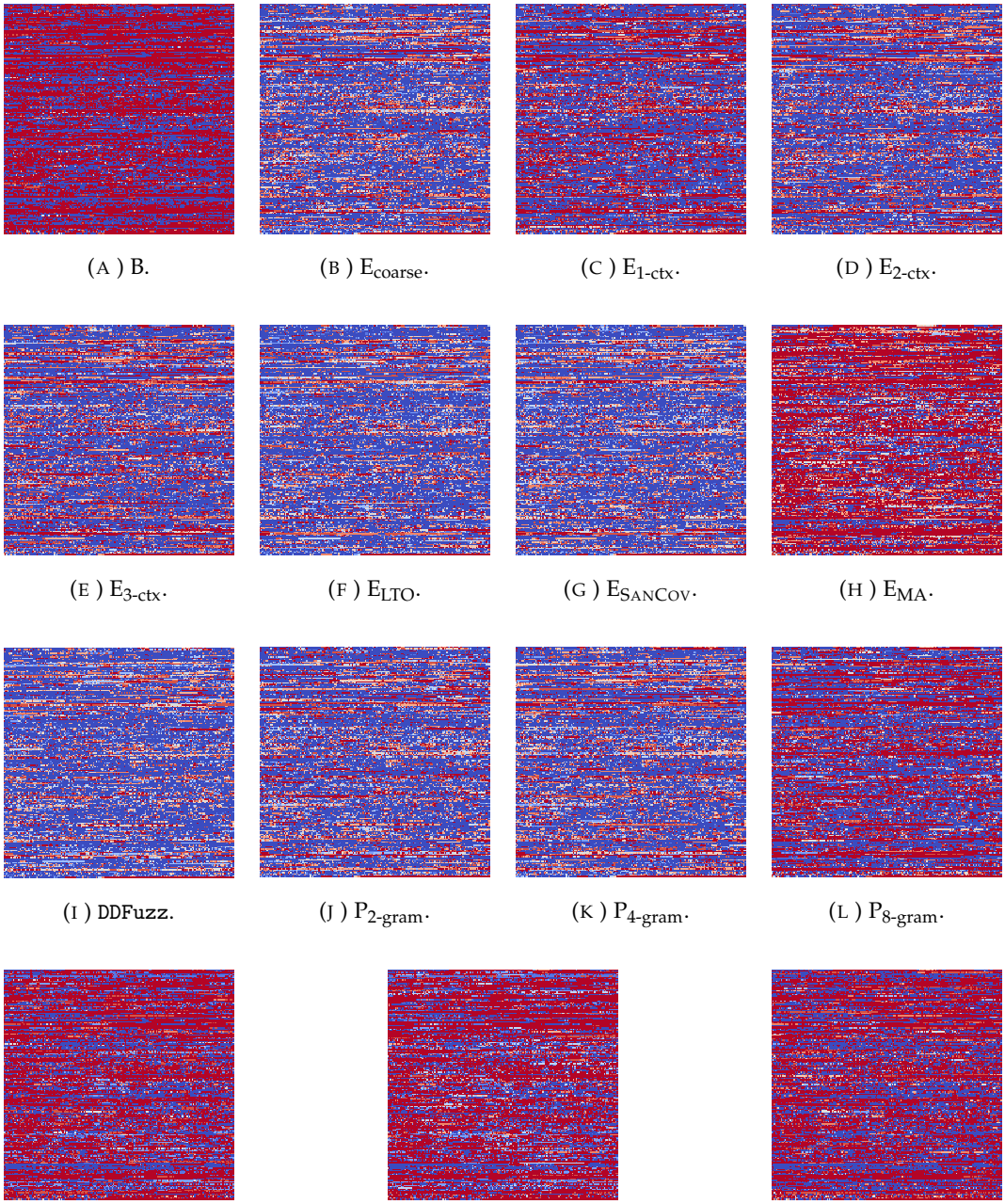
(A) B. (B) E_{coarse}. (C) $E_{1\text{-ctx}}$. (D) $E_{2\text{-ctx}}$.

(E) $E_{3\text{-ctx}}$. (F) E_{LTO}. (G) E_{SANCOV}. (H) E_{MA}.

(I) `DDFuzz`. (J) $P_{2\text{-gram}}$. (K) $P_{4\text{-gram}}$. (L) $P_{8\text{-gram}}$.

(M) $DF_{A+S/A}$. (N) $DF_{A+S/O}$. (O) $DF_{A+S/V}$.

FIGURE 7.3: Probability maps for *lcms*. Each cell represents a single edge in the context-*sensitive* CFG. The color of a cell represents the probability that this edge will be covered by the given coverage metric (red is probability = 0, blue is probability = 1).



(A) B. (B) E_{coarse}. (C) $E_{1\text{-ctx}}$. (D) $E_{2\text{-ctx}}$.

(E) $E_{3\text{-ctx}}$. (F) E_{LTO}. (G) E_{SANCov}. (H) E_{MA}.

(I) `DDFuzz`. (J) $P_{2\text{-gram}}$. (K) $P_{4\text{-gram}}$. (L) $P_{8\text{-gram}}$.

(M) $DF_{A+S/A}$. (N) $DF_{A+S/O}$. (O) $DF_{A+S/V}$.

FIGURE 7.4: Probability maps for *sqlite3*. Each cell represents a single edge in the context-*insensitive* CFG. The color of a cell represents the probability that this edge will be covered by the given coverage metric (red is probability = 0, red is probability = 1).

where E is the set of all CFG edges in a target. Equation (7.2) represents the expected number of edges that would be covered by the set of fuzzers $\mathcal{F}$.

Figure 7.5 shows the predicted improvements in context-(in)sensitive CFG edge coverage when pairing coverage metrics. These improvements are based on the coverage results in Tables 7.2a and 7.2b. Each heatmap shows the predicted improvements for a single target, where each cell contains (a) the expected number of covered edges (per Eq. (7.2)), and (b) the percentage increase over only using a single coverage metric. The colors reflect the relative performance of the coverage metric: red is worst performance, blue is best performance.

These heatmaps make it clear—across a majority of targets—that accurate edge coverage metrics (E_{LTO} and E_{SANCOV}) are superior to all other coverage metrics when maximizing context-*insensitive* coverage. In particular, these metrics benefitted little when combined with any of the other 13 metrics (only up to ∼3.5 %). The exceptions to this result were *mbedtls*, *sqlite3*, and *stb*, which were predicted to improve their coverage by 4 %, 10 %, and 11 %, respectively. Interestingly, these improvements occurred when E_{DDG} was incorporated, suggesting that incorporating some form of data-flow-based coverage may yield small improvements to overall coverage.

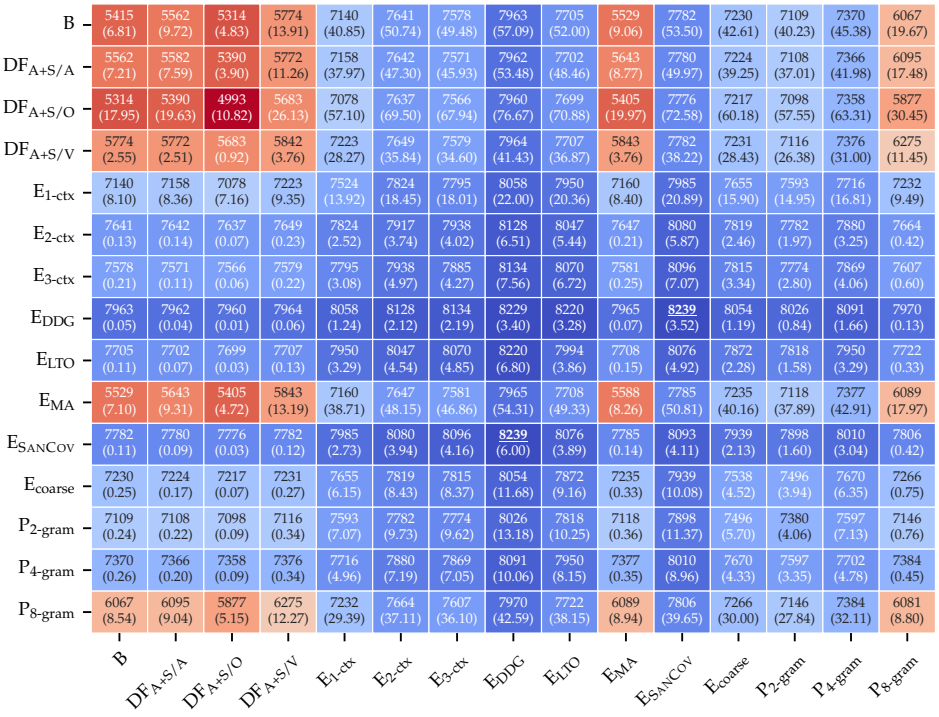
These results generally hold when considering context-*sensitive* coverage. While E_{LTO} and E_{SANCOV} continue to dominate, the performance gains are much larger (e.g., up to 61 % on *libpng*) and occur when pairing with a wider range of coverage metrics (e.g., context-sensitive edge coverage and n-gram coverage).

Finding

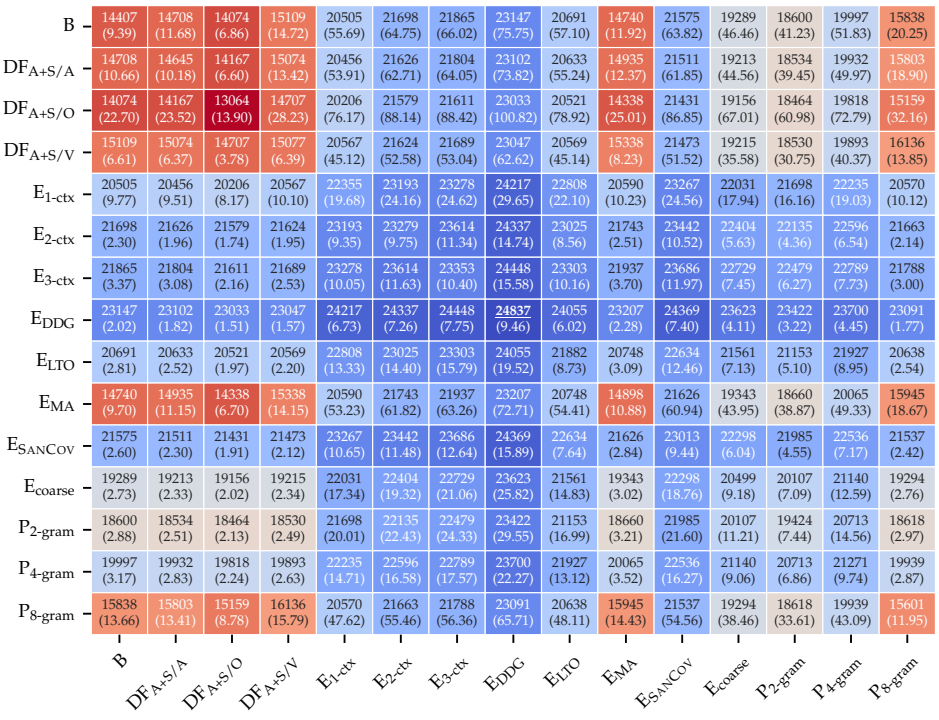
The best-performing coverage metrics—E_{LTO} and E_{SANCOV}—were predicted to benefit little when combined with other coverage metrics. When performance was predicted to improve, it was generally when E_{DDG} was incorporated, suggesting that some form of data-flow-based coverage may yield small improvements to overall coverage.

7.4 Comparison to Previous Studies

Salls et al. [183] and Wang et al. [214] have also conducted studies on the performance of fuzzer coverage metrics. However, these studies: (i) did not take into account data-flow-based metrics; (ii) used a smaller set of "real-world" targets; and (iii) did not compare fuzzers based on coverage (instead focusing on counting crashes in synthetic benchmarks such as the Cyber Grand Challenge (CGC) and LAVA-M).

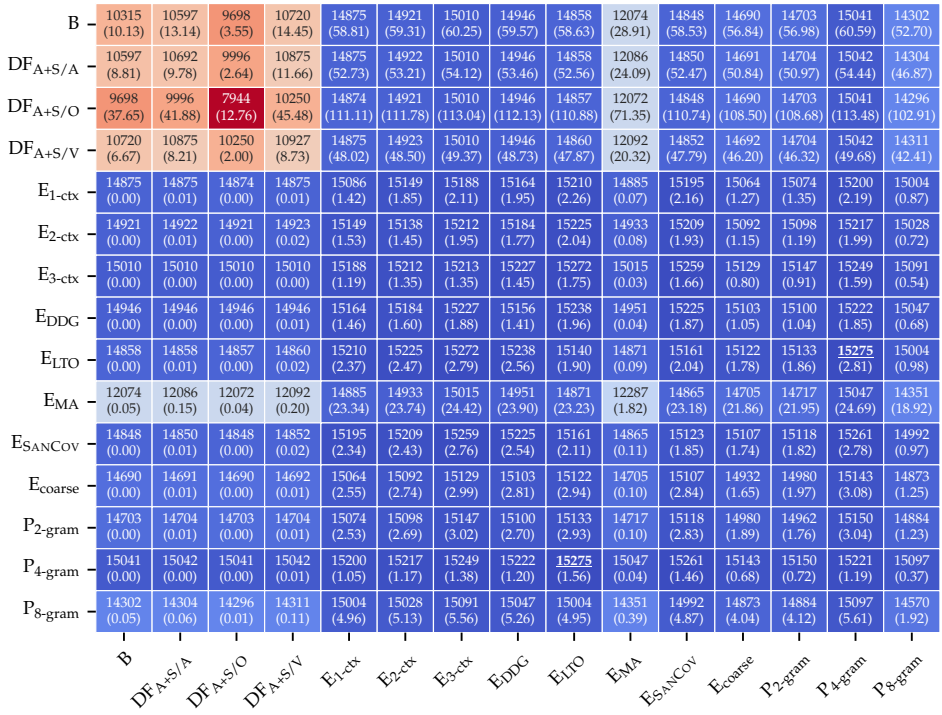


(A) *bloaty* (context-insensitive).

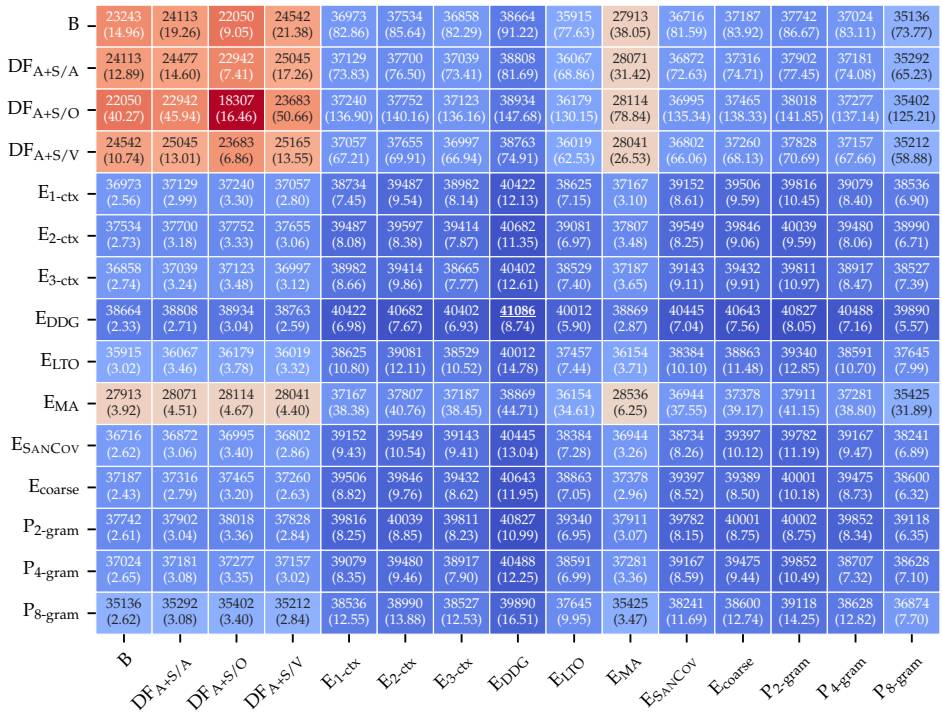


(B) *bloaty* (context-sensitive).

FIGURE 7.5: The predicted improvements to ICFG control-flow coverage. These improvements are made by pairing the fuzzer on the y-axis with the fuzzer on the x-axis. Each cell contains the expected number of covered edges and the percentage increase over only using the fuzzer on the y-axis (in brackets). The highest expected coverage is **highlighted**, while the cell color reflects the relative performance of the fuzzer: red is worst performance, blue is best performance.

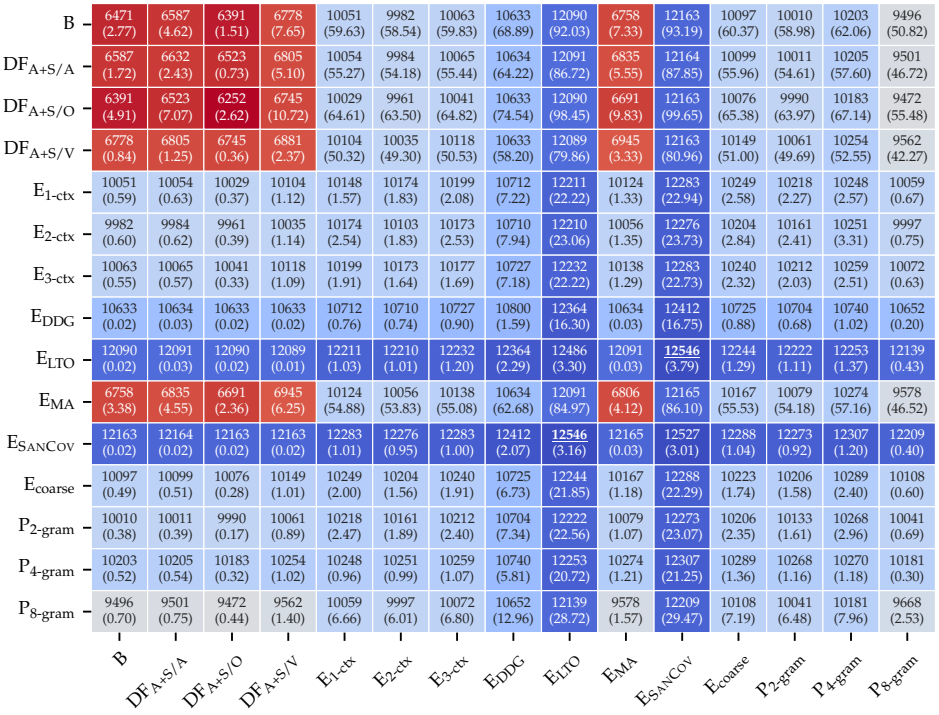


(C) *curl* (context-insensitive).

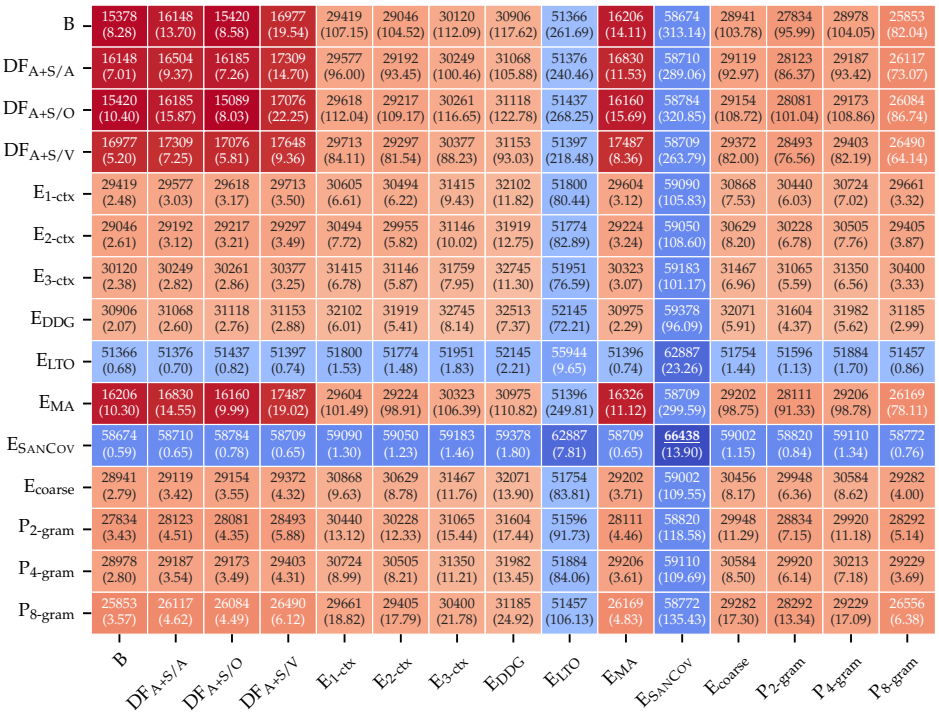


(D) *curl* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

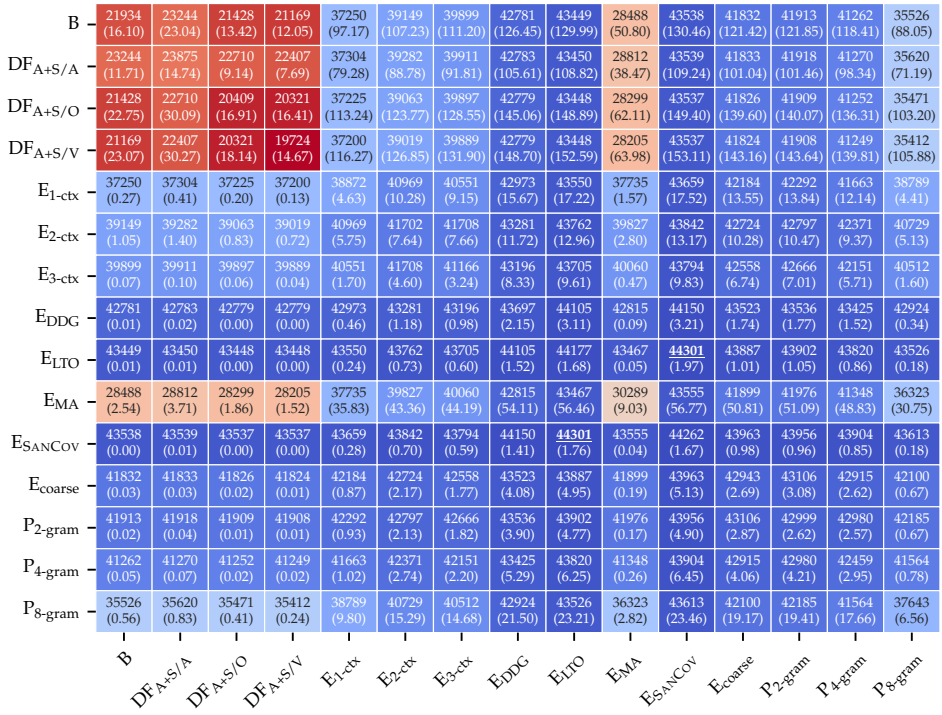


(E) *freetype2* (context-insensitive).

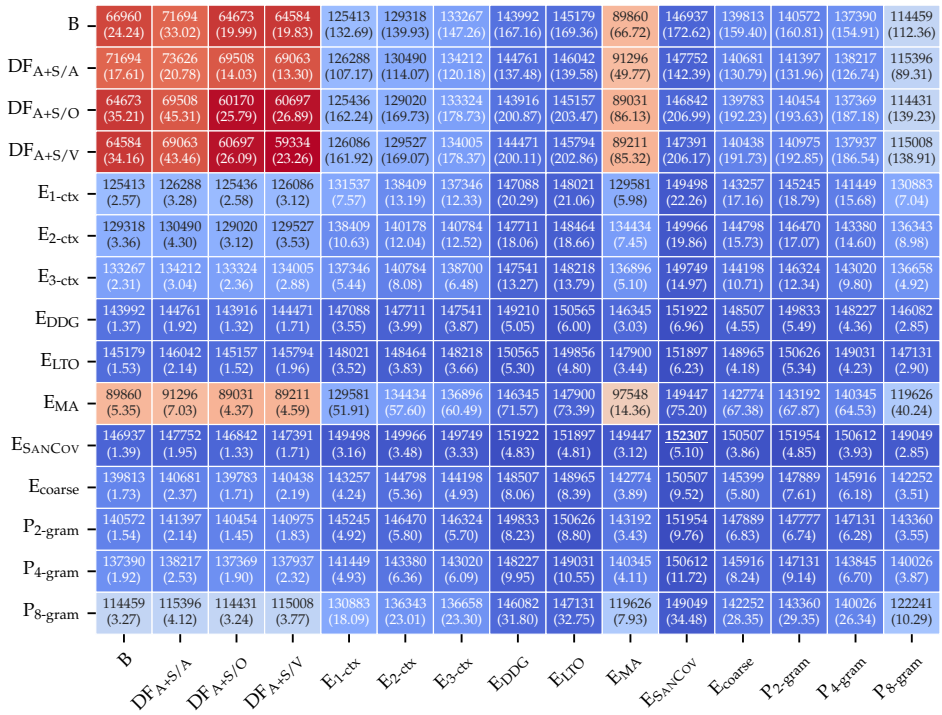


(F) *freetype2* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

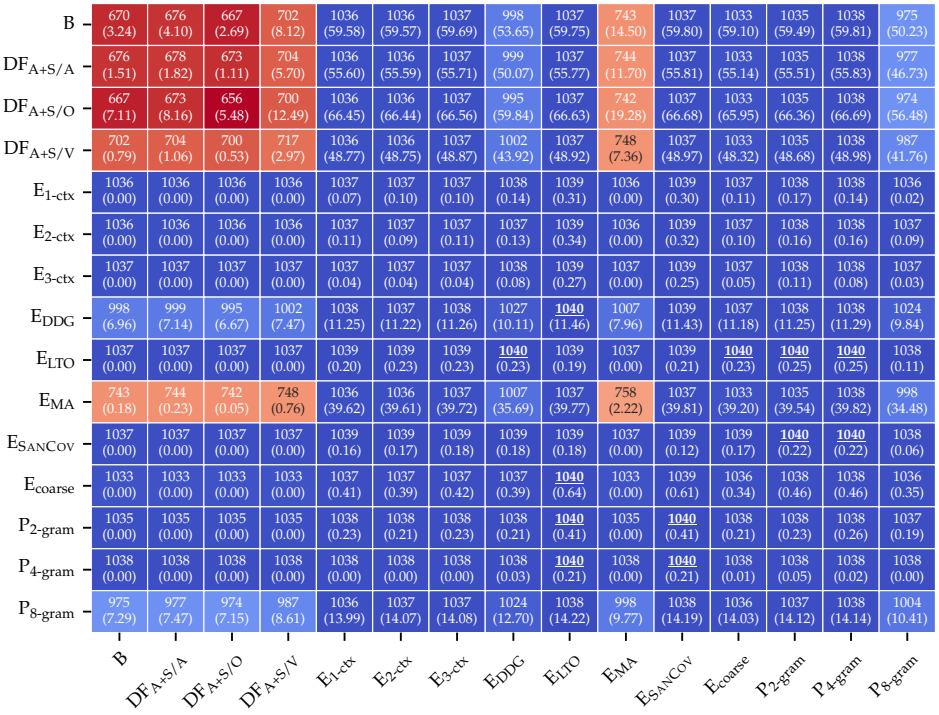


(G) *harfbuzz* (context-insensitive).

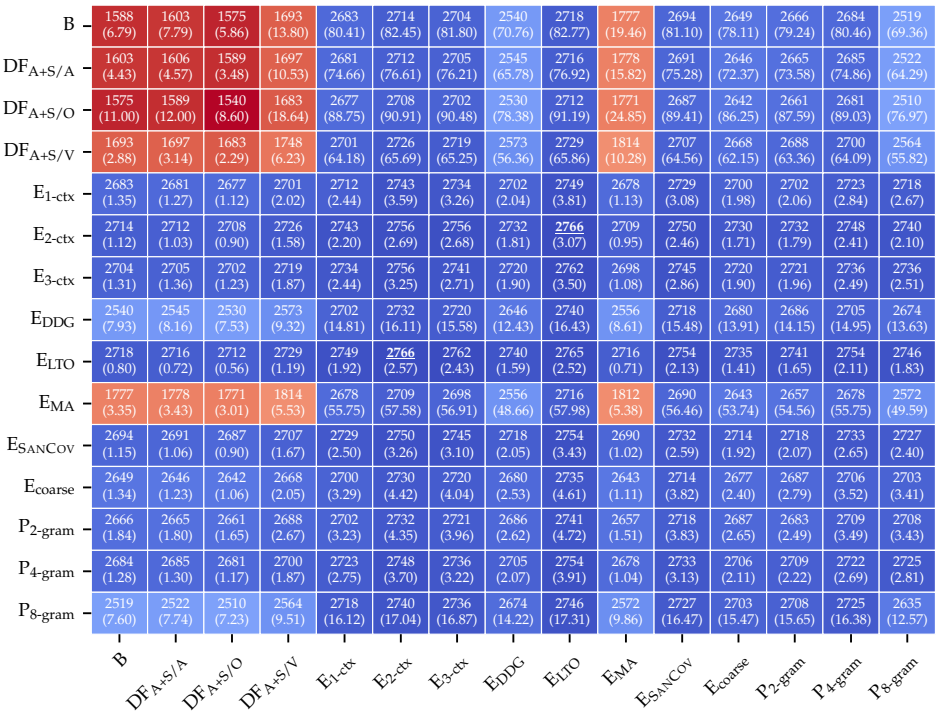


(H) *harfbuzz* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

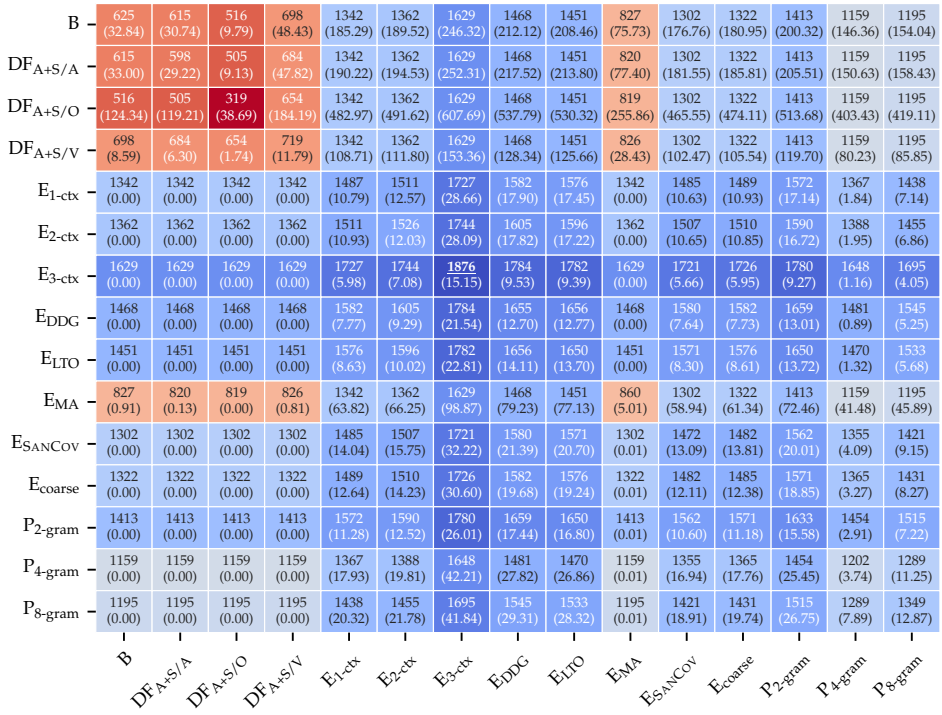


(I) *jsoncpp* (context-insensitive).

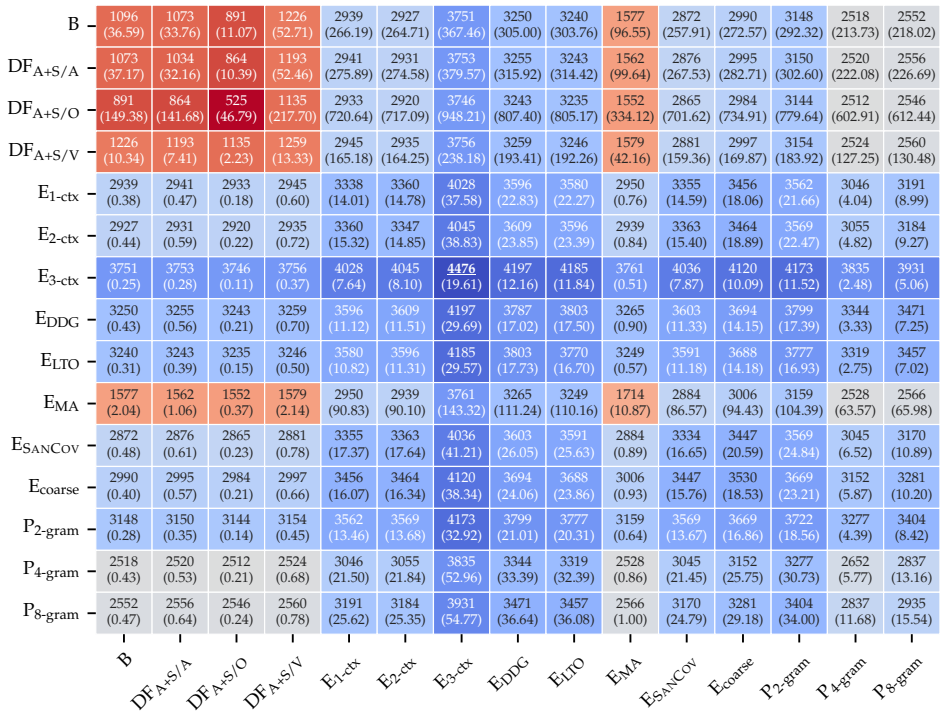


(J) *jsoncpp* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

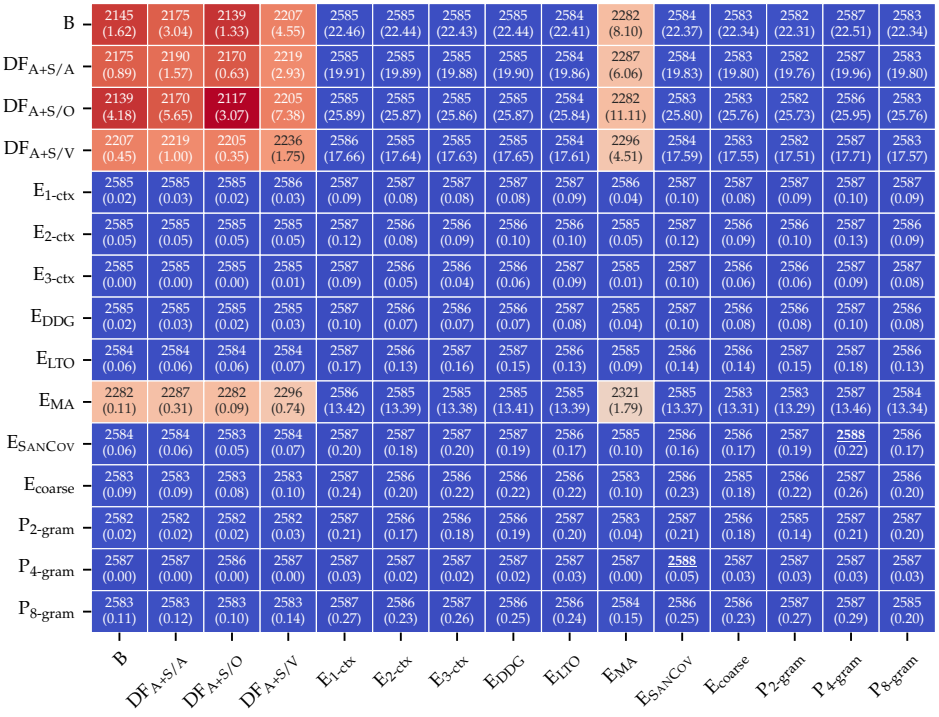


(K) *lcms* (context-insensitive).

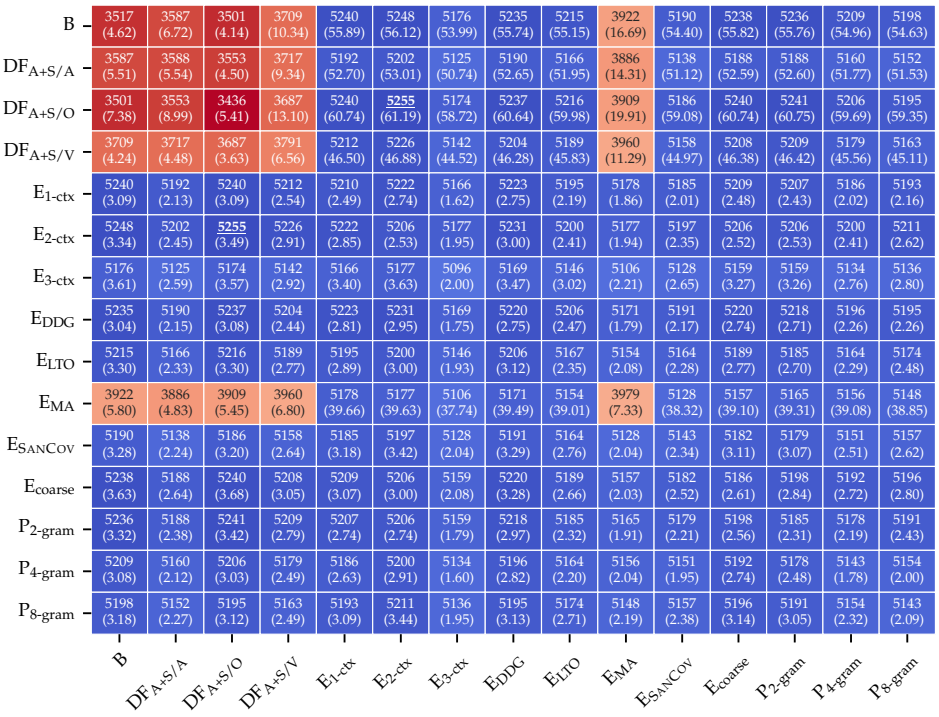


(L) *lcms* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

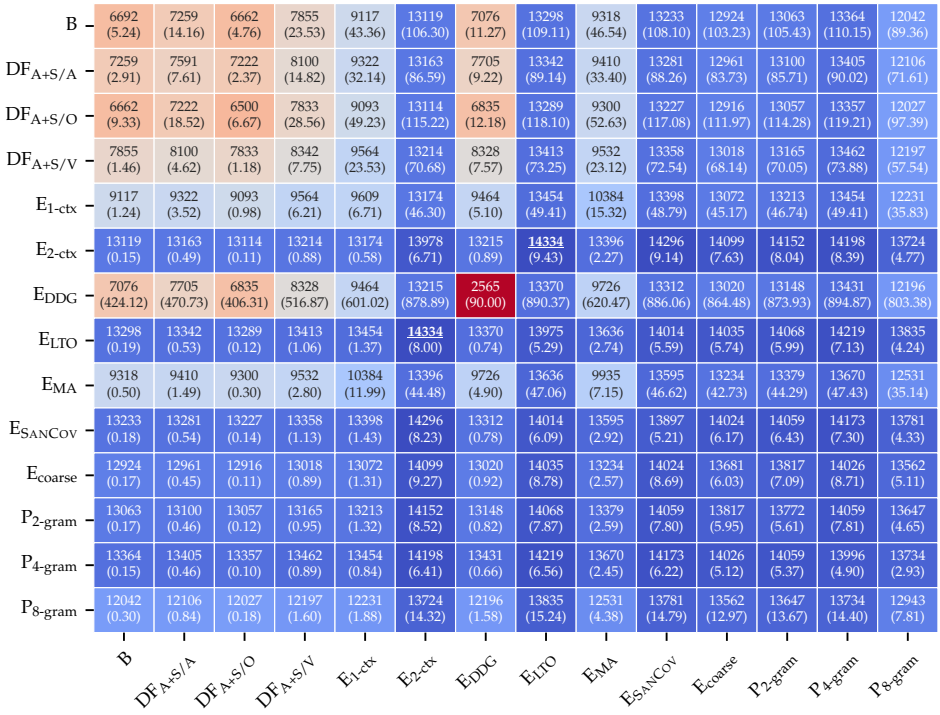


(M) *libpng* (context-insensitive).

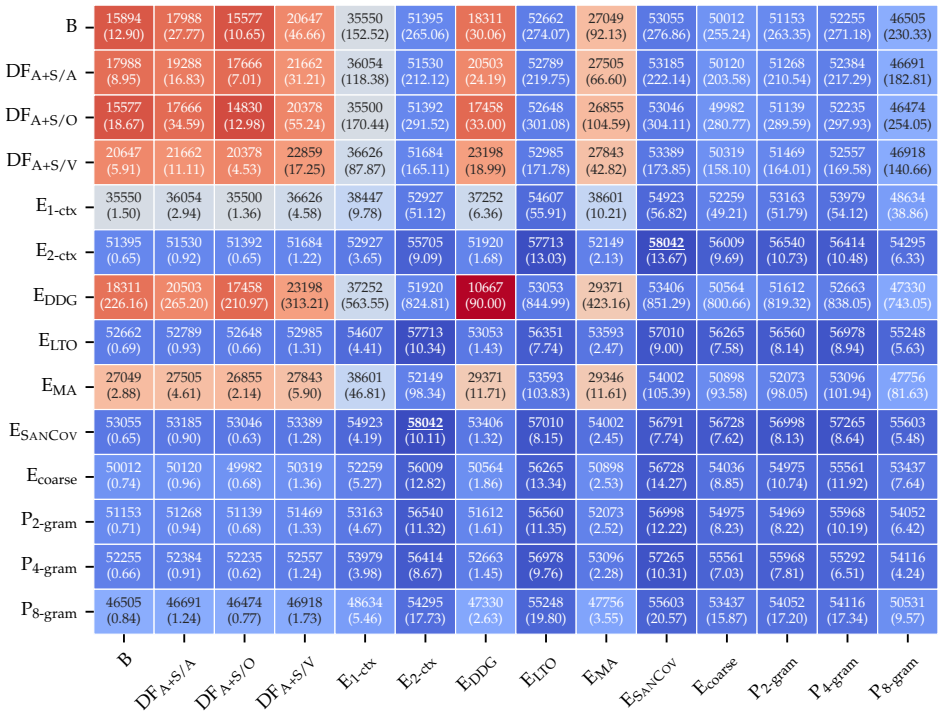


(N) *libpng* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

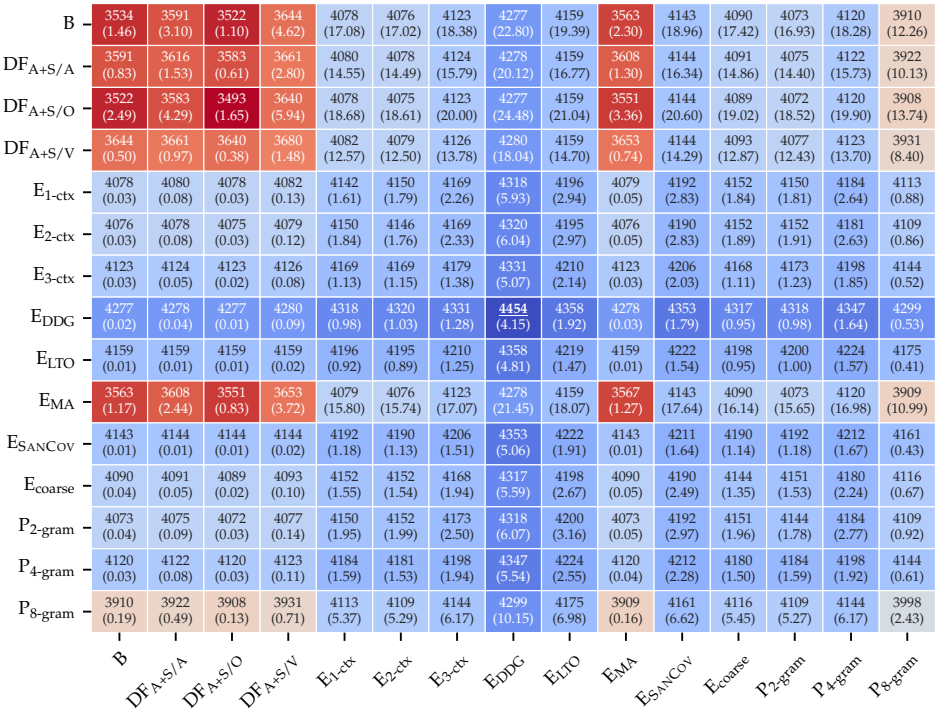


(O) *libxslt* (context-insensitive).

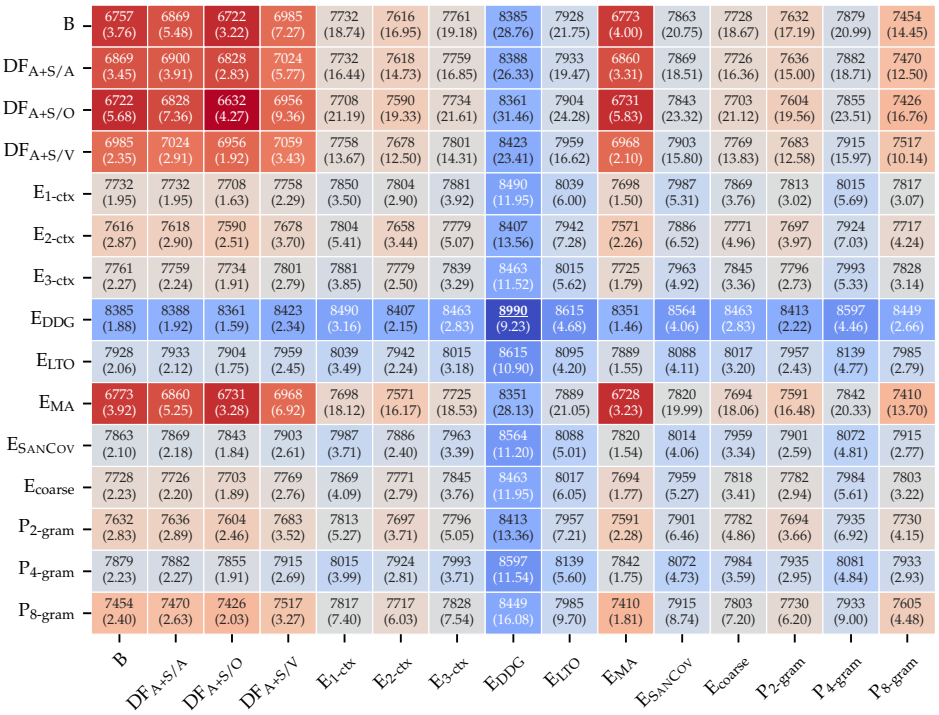


(P) *libxslt* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

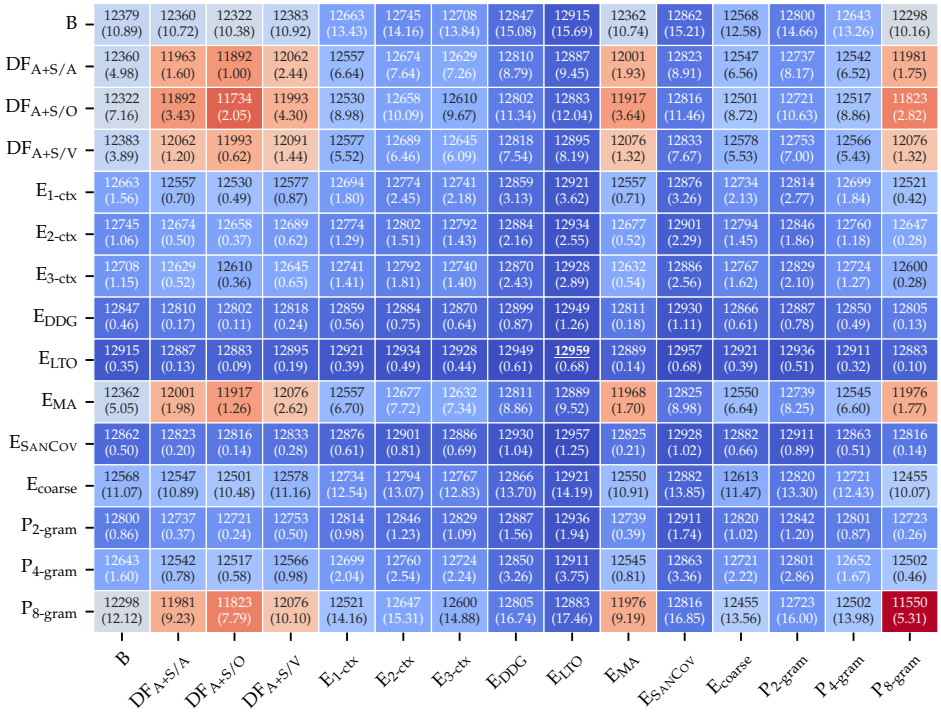


(Q) *mbedtls* (context-insensitive).

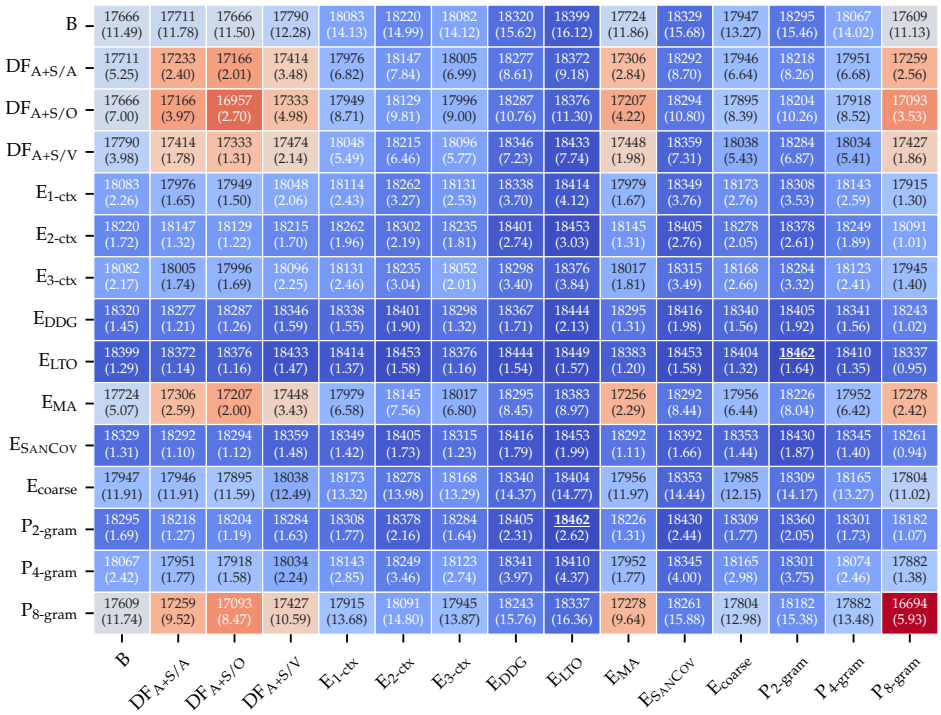


(R) *mbedtls* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

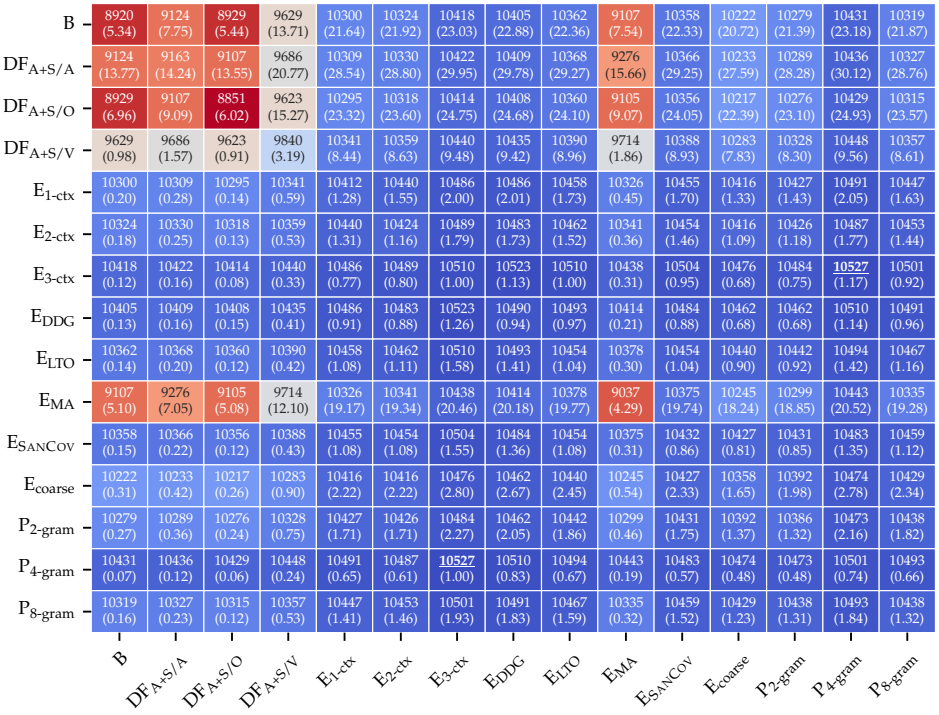


(S) *openh264* (context-insensitive).

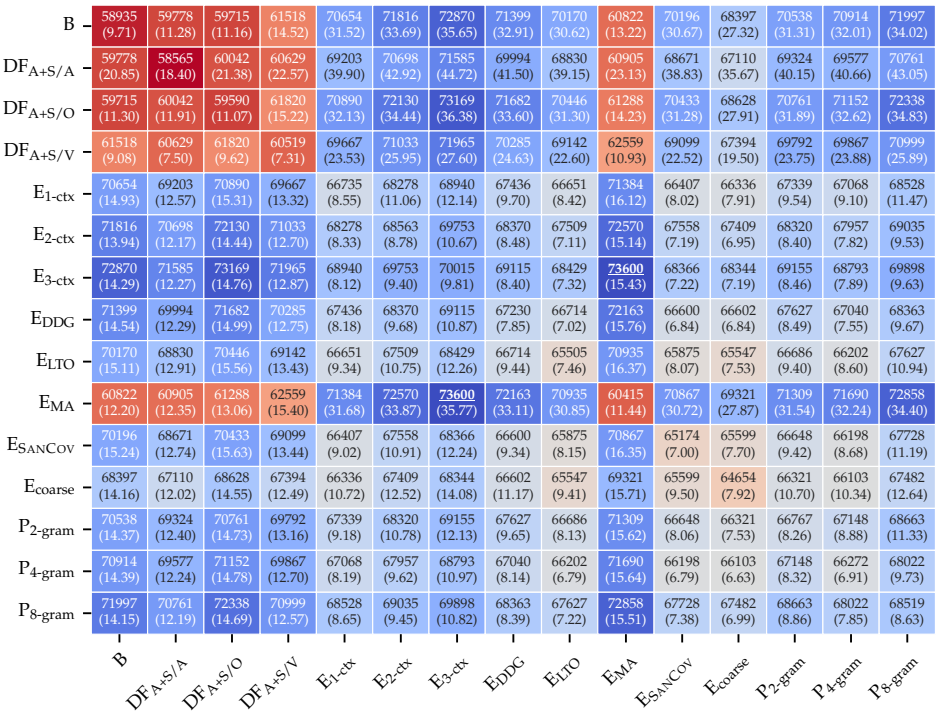


(T) *openh264* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

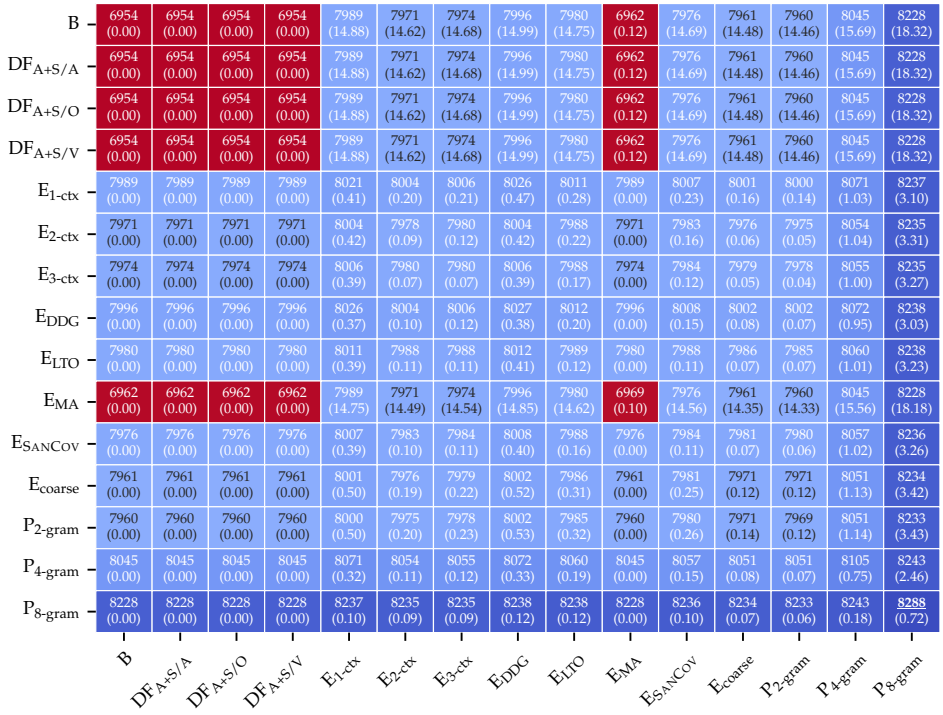


(U) *openssl* (context-insensitive).

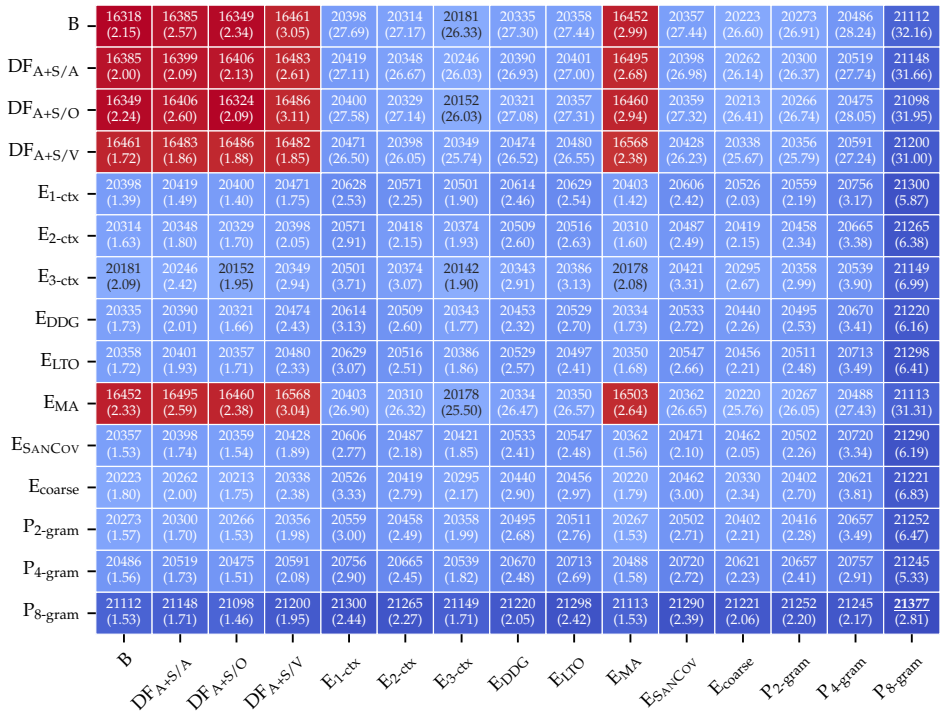


(V) *openssl* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

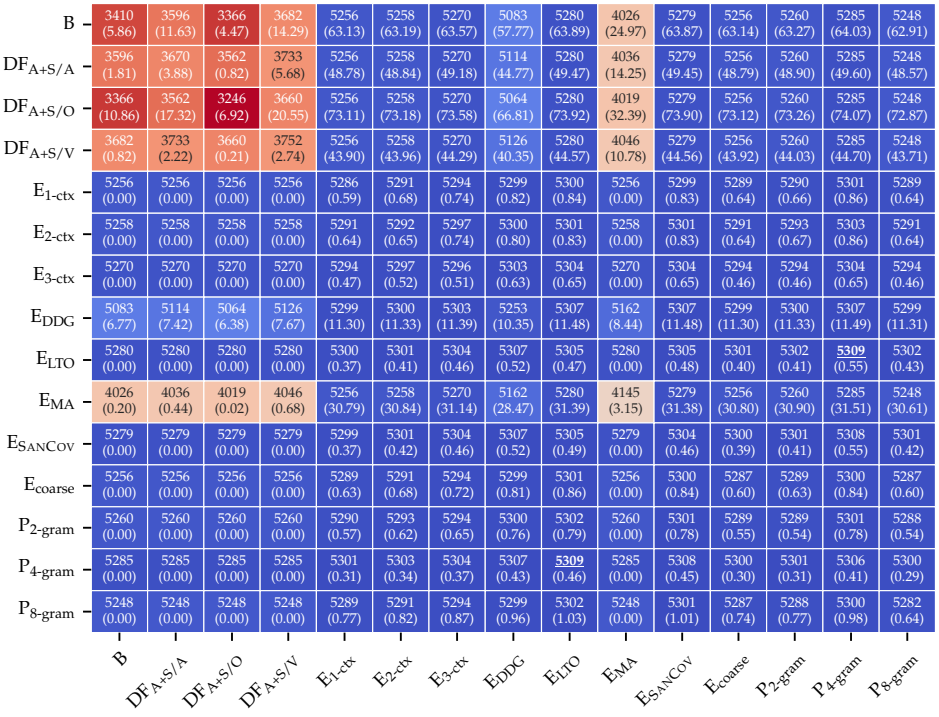


(W) *openthread* (context-insensitive).

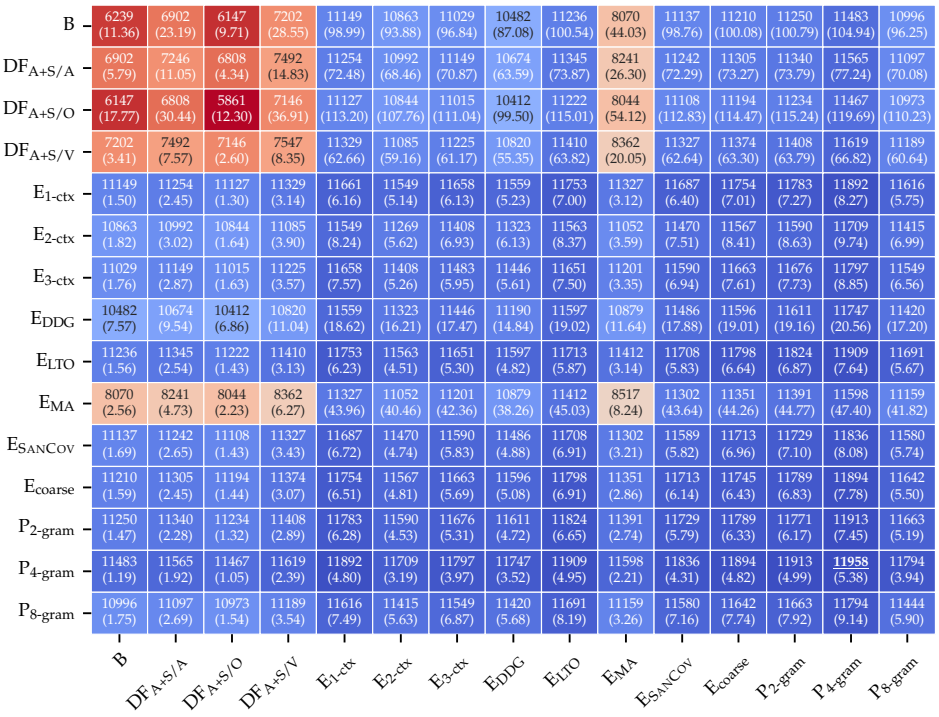


(X) *openthread* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

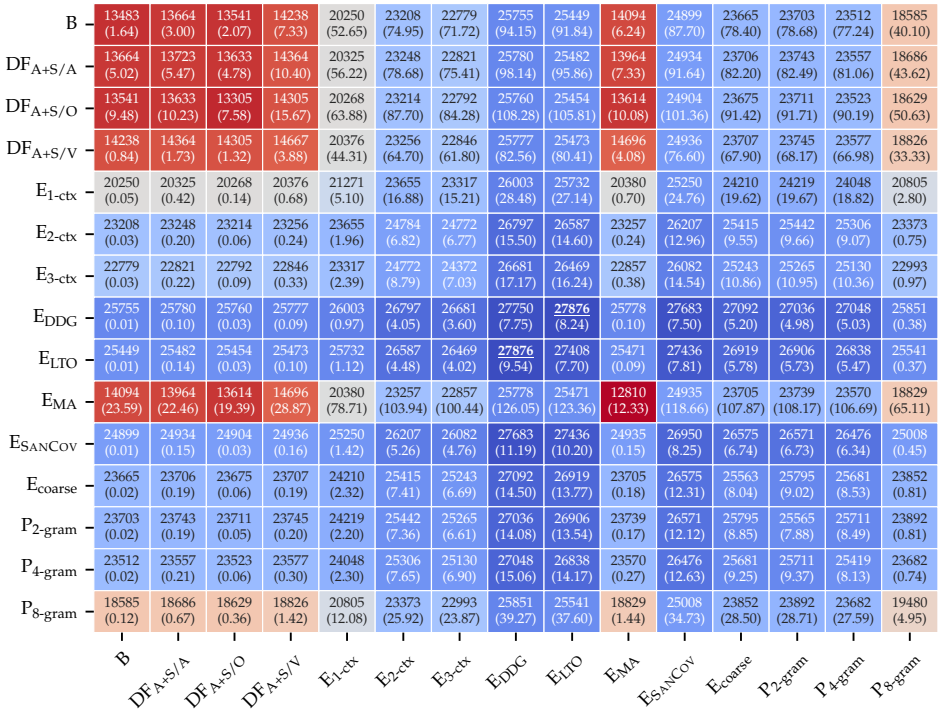


(Y) *re2* (context-insensitive).

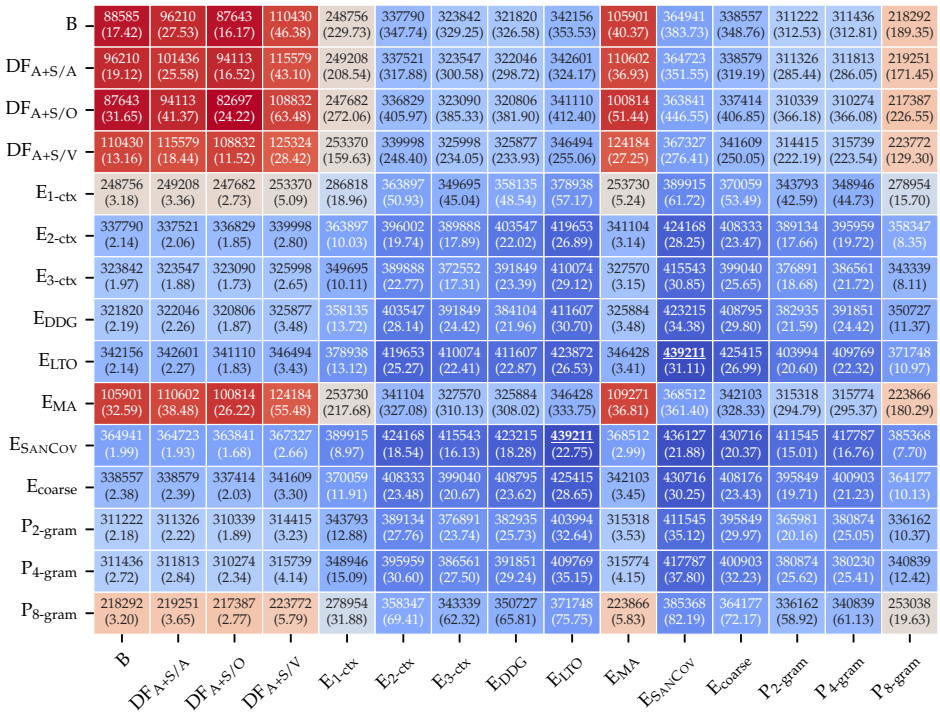


(Z) *re2* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

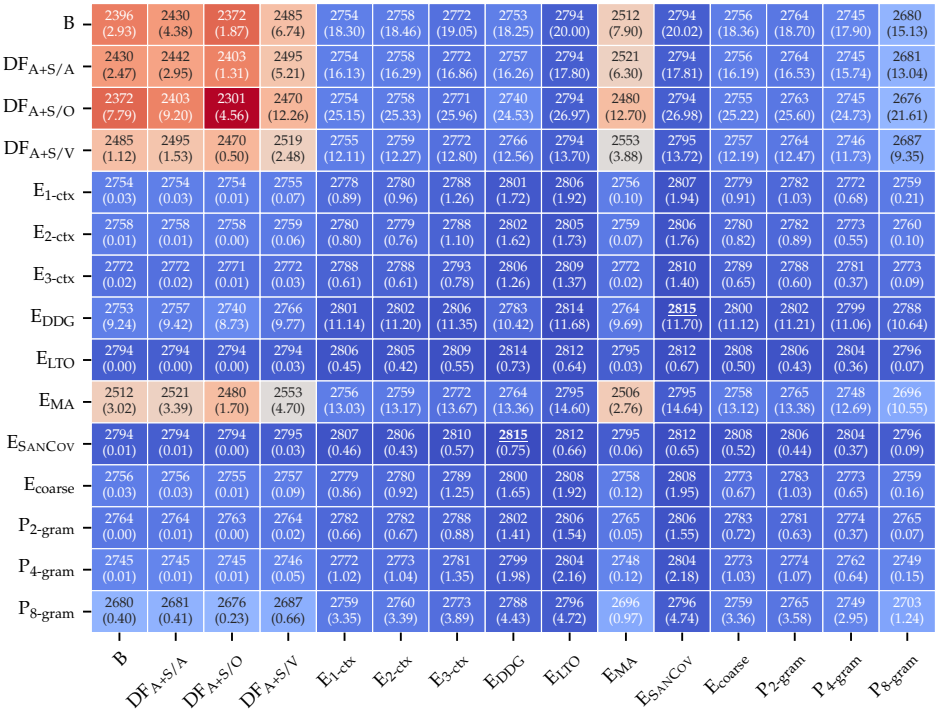


(AA) *sqlite3* (context-insensitive).

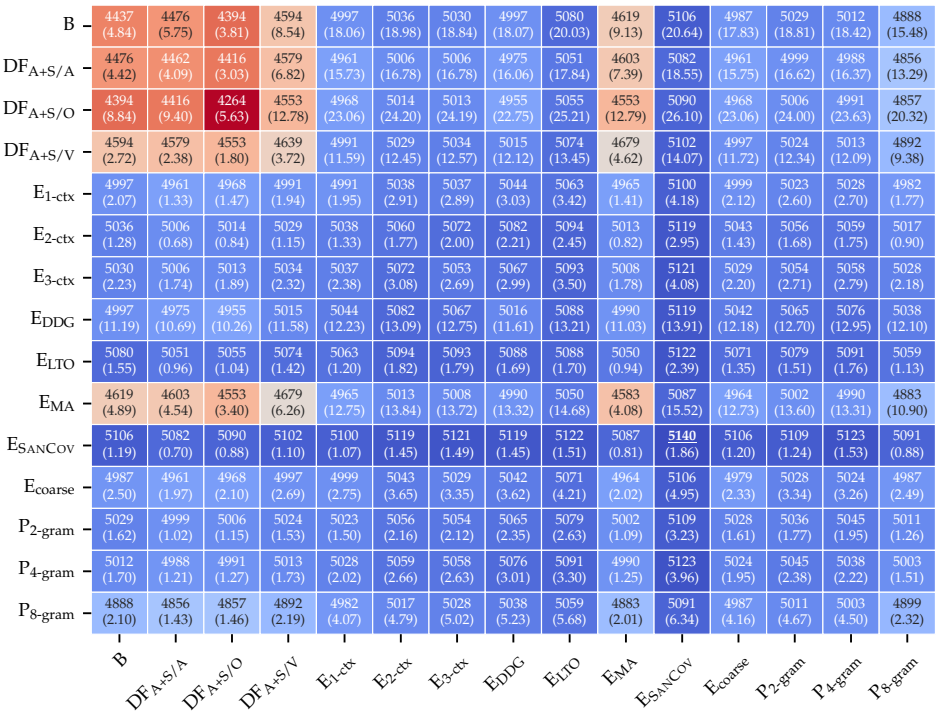


(AB) *sqlite3* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

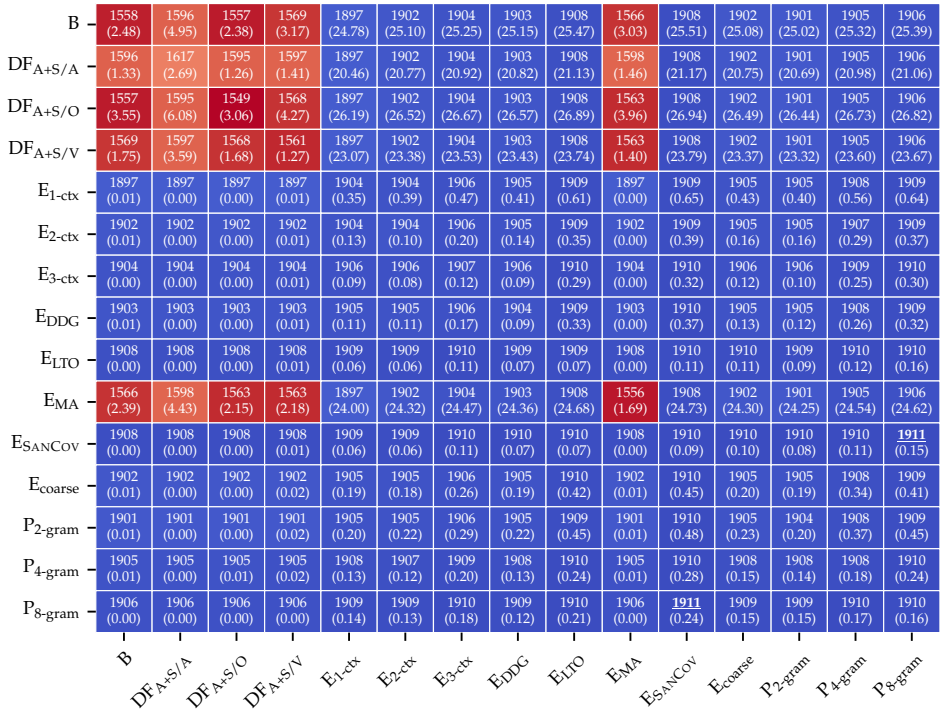


(AC) *stb* (context-insensitive).

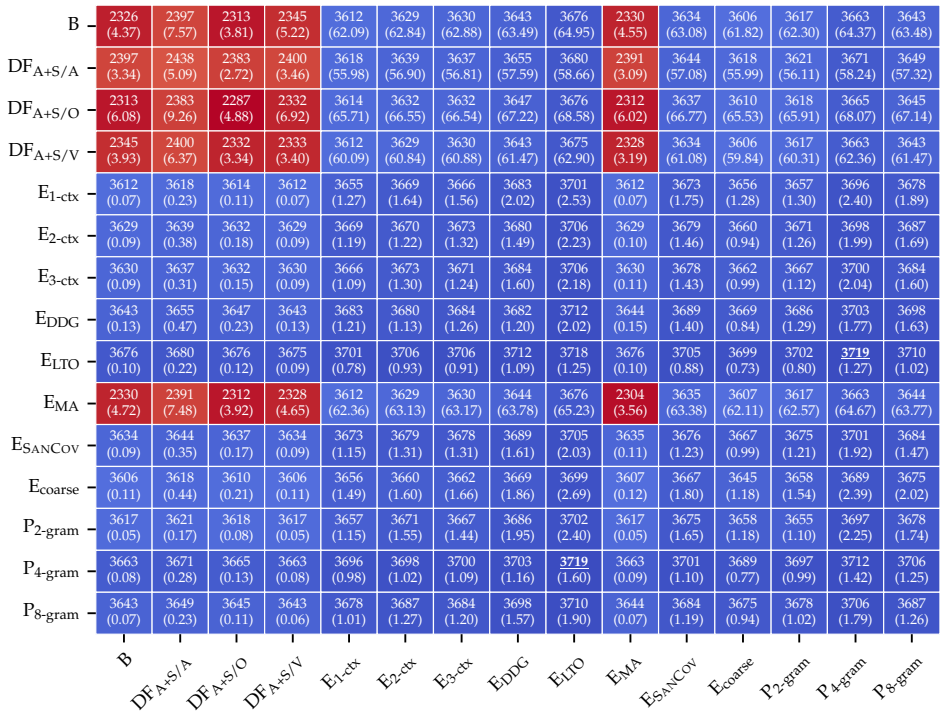


(AD) *stb* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

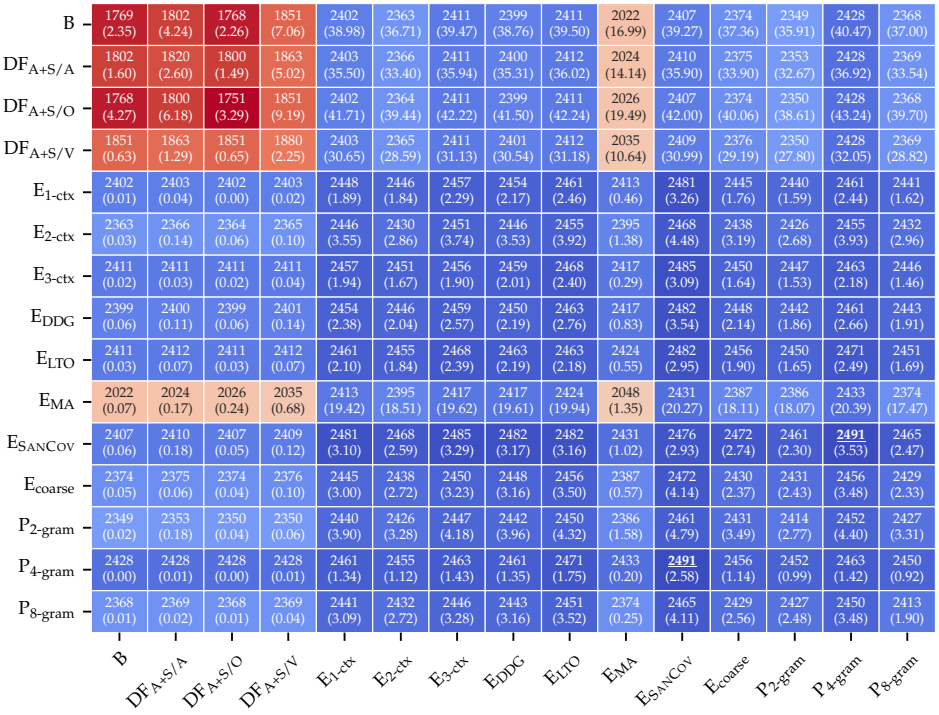


(AE) *vorbis* (context-insensitive).

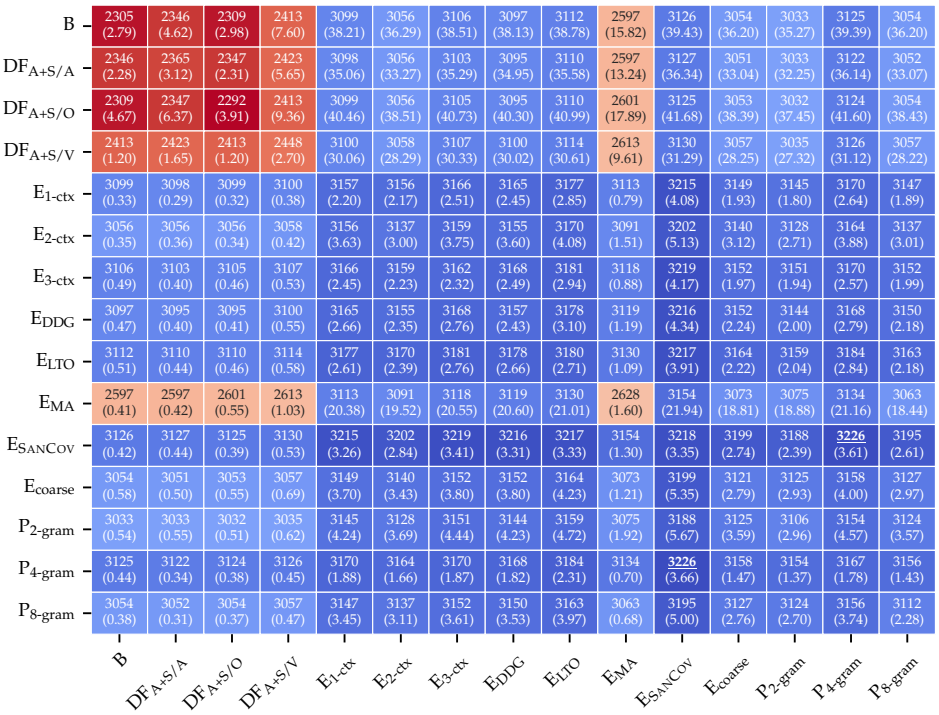


(AF) *vorbis* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

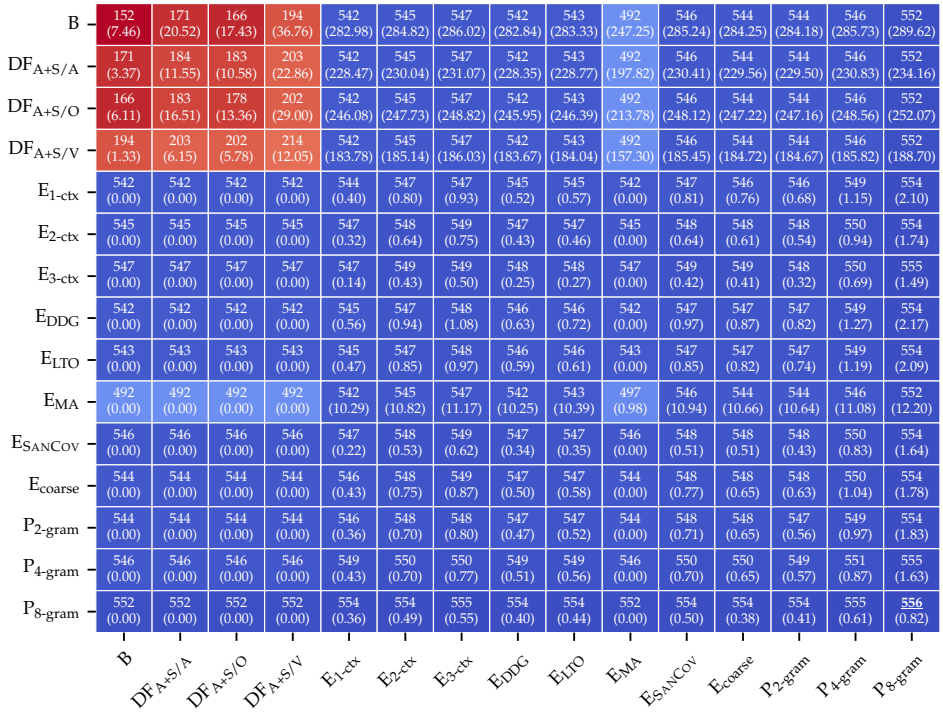


(AG) *woff2* (context-insensitive).

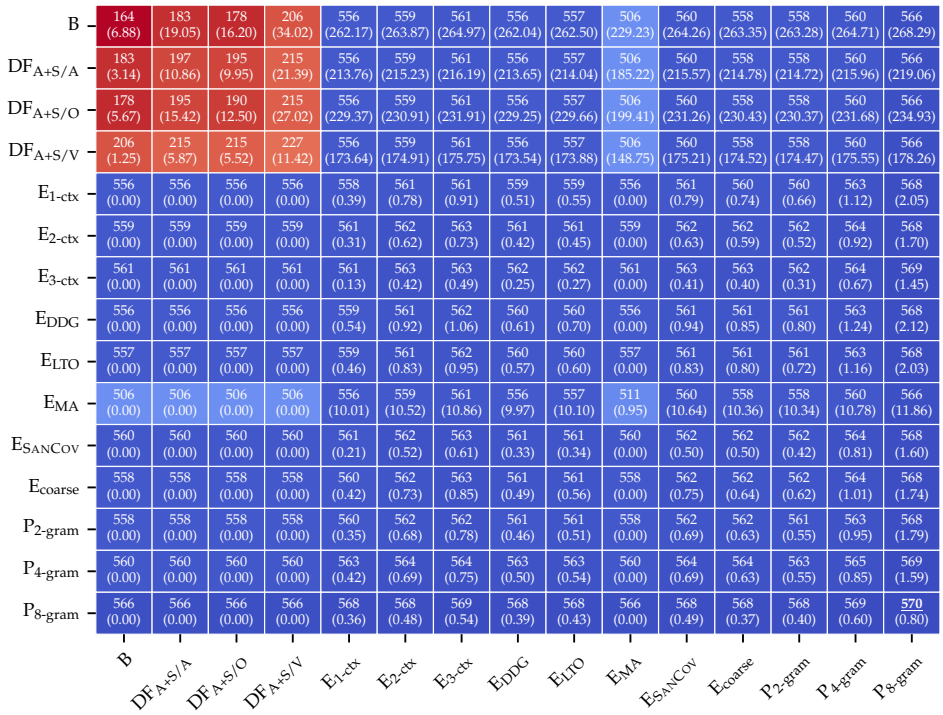


(AH) *woff2* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).



(AI) *zlib* (context-insensitive).



(AJ) *zlib* (context-sensitive).

FIGURE 7.5: Predicted coverage improvements (continued).

The evaluation performed by Wang et al. [214] was inconclusive when comparing the performance of individual coverage metrics: for CGC "*there is no single [coverage metric] winner that beats everyone else*", while the use of LAVA-M was "*not very suitable for our goal [of comparing the performance of coverage metrics]*". Finally, the real-world targets report crashes rather than bugs, and these bugs were not thoroughly deduplicated (e.g., it is unrealistic to assume up to 1,400 unique bugs were found in the *info2cap* target). Despite this, Wang et al. [214] found that combining multiple coverage metrics in an ensemble configuration (where each fuzzer ran in parallel and shared seeds) "*outperformed the baseline [i.e., fuzzing with a single coverage metric] by wide margins*".

Salls et al. [183] similarly studied the performance of different coverage metrics, however, their study focused on formalizing coverage metrics by borrowing concepts from static analysis. Again, their study involved a relatively small evaluation of targets sourced from the CGC benchmark and two real-world programs. Salls et al. [183] showed that fuzzing outcomes were maximized when multiple coverage metrics were used in an ensemble configuration (which we did not investigate). When restricted to only a single metric, the "*edge + return loc*"—or context-sensitive edge coverage here—metric achieved the highest code coverage.

In contrast to these two studies, ours is the first large-scale comparison of coverage metrics that takes into account data-flow-based metrics. Moreover, we focus on the coverage achieved by different coverage metrics (as opposed to crash and bug counts). Once again, we found that more-sensitive coverage metrics did not necessarily lead to improvements in fuzzer performance, due to the "queue explosion" problem [141]. While Salls et al. [183] recommend combining multiple coverage metrics to maximize fuzzer performance, we do not recommend combining highly sensitive metrics (e.g., memory accesses, more sensitive forms of *def-use* chains) unless aggresive filtering occurs when sharing seeds between fuzzing nodes (assuming an ensemble fuzzing strategy is used). Moreover, our results in Section 7.3.3 show that there is little reason to look past traditional edge coverage metrics: either the E_{LTO} or E_{SANCOV} varieties will suffice for maximizing a fuzzer's state space search. If an ensemble configuration with multiple metrics is used, a combination of lossless edge coverage, context-sensitive edge coverage, edge coverage augmented with a DDG, and n-gram coverage should be preferred.

7.5 Future Work

We see a number of promising directions for future work, which we discuss here.

Visualizing coverage. As discussed in Section 7.2, a fuzzer's state-space search is typically visualized with a "(control-flow) coverage over time" plot. However, *is this the best approach for visualizing coverage metrics across multiple dimensions*? In particular, how can we incorporate the data-flow coverage approaches discussed throughout this dissertation (e.g., here and in Section 5.5.4) with traditional control-flow coverage? If we think of control- and data-flow as different axes of a target's state space (as illustrated in Fig. 3.2), then a method for visualizing high-dimensional data—e.g., Principal Component Analysis (PCA) or t-distributed stochastic neighbor embedding (t-SNE) [136]—may be appropriate. We leave improvements to the visualization of fuzzers' state-space search as future work.

Querying state space search. While effective, fuzzers often benefit from human-in-the-loop interaction. For example, IJON [7] uses human-provided annotations to guide the fuzzer. Similarly, Yan et al. [226] observed that many security professionals optimize fuzzing by assisting and guiding the testing process. Unsurprisingly, improving fuzzing via human interaction requires a thorough understanding of state space coverage (e.g., to understand fuzz blockers [70]). While tools such as Fuzz Introspector [32] and InFuzz [226] improve human understanding of a fuzzer's state space search (e.g., via a web-based user interface), they do not provide the user with a *convenient querying interface*. This lack of a convenient querying interface hinders security professionals—who often have relevant domain expertise—from better understanding fuzzer performance (e.g., beyond aggregate coverage statistics that are typically available). We propose using a logic-based query language—e.g., Datalog—for analyzing and querying the coverage achieved during a fuzzing campaign. Datalog has a long history of being used to build static program analyses [15, 26, 64, 66, 81, 179]. Developing a query tool in Datalog—rather than a traditional graph database (e.g., Neo4j [218])—would allow one to combine static and dynamic analysis data in a single query tool. This combination would enable a better understanding of fuzzer roadblocks and a mechanism for quantifying data flow coverage. We leave building such a query tool as future work.

Ensemble fuzzing. Our evaluation focused on understanding the performance of fuzzer coverage metrics individually. However, as discussed in Sections 7.3.3 and 7.4, different coverage metrics can be combined using an ensemble fuzzing configuration [44, 78, 171, 183, 214]. In this configuration, multiple coverage metrics are run in parallel and seeds are shared across fuzzing instances. However, care must be taken to prevent queue explosion, which can hinder (a) an individual fuzzer's seed scheduler, and (b) the effective sharing of seeds across fuzzers. Moreover, a diversity of fuzzers may also be beneficial (e.g., non-AFL-based fuzzers). While we

used the CUPID [78] approach for predicting the performance of combining multiple coverage metrics, we leave the experimentation, optimization, and validation of these predictions as future work.

7.6 Chapter Summary

This final chapter discussed the difficulties in measuring and quantifying a fuzzer's state space search. To address this, we proposed using context-sensitive static and dynamic analyses to quantify the control-flow elements of a fuzzer's state space search. We use these analyses to compare the full range of control- and data-flow-based coverage metrics discussed throughout this dissertation. Our results once again demonstrated the superiority of traditional edge coverage metrics. Unfortunately, our results also show that even state-of-the-art static analyses are far from accurate on typical fuzzer targets; they either failed to complete their analysis or produced unrealistic results. We hope these results and ideas for future work inspire others to improve the accuracy and scalability of static analyses for real-world code bases.

Next, we summarize this dissertation and provide some final reflections.

Chapter 8

Conclusions and Reflections

While fuzzing has repeatedly demonstrated its effectiveness at finding bugs in real-world software, it is not a panacea for society's software security woes. For example, most fuzzing applications remain focused on exposing memory safety violations (despite other bug classes—e.g., privilege escalation, type confusion—enabling exploitation), and "deep" bugs continue to evade long-running fuzzing campaigns (e.g., with the fuzzer unable to satisfy complex control- and data-flow dependencies). It is this last point that we focus on in this dissertation. In particular, those deep bugs lurking in the corners of a program's state space.

This dissertation attempts to understand better and improve fuzzers' state space search. To this end, we present several techniques, frameworks, and methodologies for measuring and enhancing a fuzzer's ability to explore a target program's state space.

Chapters 2 and 3 start with an overview of coverage-guided greybox fuzzing, exploring and systematizing the different techniques fuzzers use to measure a target's state space. We find that most prior work relies on control-flow-based coverage metrics. While performant, these metrics only consider one dimension of a program's state space (i.e., features based on control flow), disregarding data-flow features.

Chapter 4 empirically demonstrates the importance of carefully curating a fuzzer's initial seed set, bootstrapping the fuzzer's state space search. Our results show that fuzzing outcomes (i.e., bug discovery and state space exploration) vary significantly depending on this initial seed set. This led to the design of a new fuzzing corpus minimizer, OPTIMIN. Unlike prior work, OPTIMIN can derive an optimal corpus by encoding the corpus minimization problem as a boolean satisfiability (SAT) problem. OPTIMIN has since been adopted by the broader fuzzing community and incorporated into AFL++[1] and LIBAFL[2], two state-of-the-art coverage-guided greybox

[1] https://github.com/AFLplusplus/AFLplusplus/commit/62f1bfed99b82bc073c138a00ff9a30bb596d09d
[2] https://github.com/AFLplusplus/LibAFL/pull/739

fuzzers.

Chapter 5 promotes data-flow coverage to a first-class citizen (rather than using it to complement control-flow coverage, or ignoring it altogether). We design and implement a new fuzzer, DATAFLOW, that abstracts a target's state space based on data-flow features (rather than traditional control-flow features); specifically, *def-use* chains. We show that, while not as performant as control-flow-driven fuzzers, DATAFLOW can discover bugs traditional fuzzers miss.

Chapter 6 investigates the difficulties of measuring fuzzers' state space search. We evaluate the use of several static analysis frameworks for quantifying a fuzzer's state space search, with our results revealing the limitations of these frameworks.

Finally, in Chapter 7 we implement the full range of coverage metrics discussed throughout this dissertation, comparing their ability to explore a target's state space. Our implementation of n-gram coverage has since been adopted by the broader fuzzing community and incorporated into AFL++.[3]

Each of the chapters that make up this dissertation supports our thesis: enhancing state space search leads to improved fuzzing outcomes. Without a strong foundation, a fuzzer's state space exploration is limited. Some bugs only manifest in program states not explicitly visible in control-flow-based abstractions. Security professionals deploying fuzzers require rigorous techniques to reason about their fuzzer's state space search. Ultimately, these insights lead to better fuzzing outcomes and, ideally, more secure software.

8.1 Reflections

This dissertation opens the door to new approaches for improving fuzzers' state space exploration. We present a summary of reflections here; detailed future work ideas are presented at the end of each chapter.

Chapter 4 focused on characterizing initial seed sets using traditional control-flow features (e.g., edge coverage). However, as this dissertation shows, control-flow features cover only one dimension of a target's state space. Do fuzzers perform better when bootstrapped with seeds that also maximize coverage across data-flow features?

In Chapter 5, we concluded that one of the main impediments to DATAFLOW's success was its high run-time cost (reducing fuzzer iteration rates). Does improving

[3] https://github.com/AFLplusplus/AFLplusplus/commit/5a74cffa0f22b4e3b3dbc829dfb1c8f7c7a6fb76

and optimizing the performance of our *def-use* chain coverage implementation lead to better fuzzing outcomes?

The static analyses presented in Chapter 6 rely on access to source code. This simplifies analysis, particularly compared to binary code (where control flow is harder to recover, and types are nonexistent). Can similar analyses be used by those measuring and comparing fuzzer performance on targets where source code is not available (e.g., proprietary software, embedded firmware)?

Bibliography

[1] Humberto Abdelnur et al. *Spectral Fuzzing: Evaluation & Feedback*. Research Report RR-7193. Inria, 2010. URL: https://hal.inria.fr/inria-00452015.

[2] Alif Ahmed et al. "BigMap: Future-proofing Fuzzers with Efficient Large Maps". In: *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN. IEEE, 2021, pp. 531–542. DOI: 10.1109/DSN48987.2021.00062.

[3] Mike Aizatsky et al. *Announcing OSS-Fuzz: Continuous fuzzing for open source software*. 2016. URL: https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html (visited on 05/11/2022).

[4] Lars Ole Andersen. "Program analysis and specialization for the C programming language". PhD thesis. University of Copenhagen, 1994.

[5] Andrea Arcuri and Lionel Briand. "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering". In: *International Conference on Software Engineering*. ICSE. 2011, pp. 1–10. DOI: 10.1145/1985793.1985795.

[6] Cornelius Aschermann. *On Measuring and Visualizing Fuzzer Performance*. 2020. URL: https://hexgolems.com/2020/08/on-measuring-and-visualizing-fuzzer-performance/ (visited on 12/03/2022).

[7] Cornelius Aschermann et al. "Ijon: Exploring Deep State Spaces via Fuzzing". In: *Security and Privacy*. S&P. IEEE, 2020, pp. 1597–1612. DOI: 10.1109/SP40000.2020.00117.

[8] Cornelius Aschermann et al. "NAUTILUS: Fishing for Deep Bugs with Grammars". In: *Network and Distributed System Security*. NDSS. The Internet Society, 2019. DOI: 10.14722/ndss.2019.23412.

[9] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence". In: *Network and Distributed System Security*. NDSS. The Internet Society, 2019. DOI: 10.14722/ndss.2019.23371.

[10] Florent Avellaneda. "A short description of the solver EvalMaxSAT". In: *MaxSAT Evaluations*. MSE. 2020, pp. 8–9.

[11] Nathaniel Ayewah et al. "Using Static Analysis to Find Bugs". In: *IEEE Software* 25.5 (2008), pp. 22–29. DOI: 10.1109/MS.2008.130.

[12] Fahiem Bacchus et al. *MaxSAT Evaluation 2021 - Rules*. 2021. URL: https://maxsat-evaluations.github.io/2021/rules.html#input (visited on 06/06/2022).

[13] George Balatsouras and Yannis Smaragdakis. "Structure-Sensitive Points-To Analysis for C and C++". In: *Static Analysis Symposium*. SAS. Springer, 2016, pp. 84–104. DOI: 10.1007/978-3-662-53413-7_5.

[14] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *Computing Surveys*. CSUR 51.3 (2018), pp. 1–39. DOI: 10.1145/3182657.

[15] Langston Barrett. *treeedb*. 2023. URL: https://github.com/langston-barrett/treeedb (visited on 10/01/2023).

[16] Tim Blazytko et al. "GRIMOIRE: Synthesizing Structure While Fuzzing". In: *USENIX Security*. SEC. USENIX, 2019, pp. 1985–2002.

[17] Marcel Böhme. "STADS: Software Testing as Species Discovery". In: *Transactions on Software Engineering Methodology*. TOSEM 27.2 (2018). DOI: 10.1145/3210309.

[18] Marcel Böhme, Cristian Cadar, and Abhik Roychoudhury. "Fuzzing: Challenges and Reflections". In: *IEEE Software* 38.3 (2021), pp. 79–86. DOI: 10.1109/MS.2020.3016773.

[19] Marcel Böhme and Brandon Falk. "Fuzzing: On the Exponential Cost of Vulnerability Discovery". In: *European Software Engineering Conference/Foundations of Software Engineering*. ESEC/FSE. 2020, pp. 713–724. DOI: 10.1145/3368089.3409729.

[20] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. "Estimating Residual Risk in Greybox Fuzzing". In: *European Software Engineering Conference and Foundations of Software Engineering*. ESEC/FSE. 2021, pp. 230–241. DOI: 10.1145/3468264.3468570.

[21] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. "Boosting Fuzzer Efficiency: An Information Theoretic Perspective". In: *European Software Engineering Conference/Foundations of Software Engineering*. ESEC/FSE. 2020, pp. 678–689. DOI: 10.1145/3368089.3409748.

[22] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. "Coverage-Based Greybox Fuzzing as Markov Chain". In: *Computer and Communications Security*. CCS. ACM, 2016, pp. 1032–1043. DOI: 10.1145/2976749.2978428.

[23] Marcel Böhme, László Szekeres, and Jonathan Metzman. "On the Reliability of Coverage-Based Fuzzer Benchmarking". In: *International Conference on Software Engineering*. ICSE. 2022. DOI: 10.1145/3510003.3510230.

[24] Marcel Böhme et al. "Directed Greybox Fuzzing". In: *Computer and Communications Security*. CCS. ACM, 2017, pp. 2329–2344. DOI: 10.1145/3133956.3134020.

[25] Michael D. Bond and Kathryn S. McKinley. "Probabilistic Calling Context". In: *Object-Oriented Programming Systems, Languages and Applications*. OOPSLA. 2007, pp. 97–112. DOI: 10.1145/1297027.1297035.

[26] Martin Bravenboer and Yannis Smaragdakis. "Strictly Declarative Specification of Sophisticated Points-to Analyses". In: *Object Oriented Programming Systems Languages and Applications*. OOPSLA. ACM, 2009, pp. 243–262. DOI: 10.1145/1640089.1640108.

[27] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Operating Systems Design and Implementation*. OSDI. 2008, pp. 209–224.

[28] Miguel Castro, Manuel Costa, and Tim Harris. "Securing Software by Enforcing Data-Flow Integrity". In: *Operating Systems Design and Implementation*. OSDI. 2006, pp. 147–160.

[29] Marcos Lordello Chaim et al. "Efficiently Finding Data Flow Subsumptions". In: *Software Testing, Verification and Validation*. ICST. 2021, pp. 94–104. DOI: 10.1109/ICST49551.2021.00021.

[30] Madhurima Chakraborty et al. "Automatic Root Cause Quantification for Missing Edges in JavaScript Call Graphs". In: *European Conference on Object-Oriented Programming*. Vol. 222. ECOOP. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 3:1–3:28. DOI: 10.4230/LIPICS.ECOOP.2022.3.

[31] Oliver Chang. *Taking the next step: OSS-Fuzz in 2023*. 2023. URL: https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html (visited on 02/03/2023).

[32] Oliver Chang et al. *Introducing Fuzz Introspector, an OpenSSF Tool to Improve Fuzzing Coverage*. 2022. URL: https://openssf.org/blog/2022/06/09/introducing-fuzz-introspector-an-openssf-tool-to-improve-fuzzing-coverage/ (visited on 03/10/2023).

[33] Oliver Chang et al. *OSS-Fuzz: Five months later, and rewarding projects*. 2017. URL: https://opensource.googleblog.com/2017/05/oss-fuzz-five-months-later-and.html (visited on 05/11/2022).

[34] Hongxu Chen et al. "Hawkeye: Towards a Desired Directed Grey-Box Fuzzer". In: *Computer and Communications Security*. CCS. ACM, 2018, pp. 2095–2108. DOI: 10.1145/3243734.3243849.

[35] Hongxu Chen et al. "MUZZ: Thread-Aware Grey-Box Fuzzing for Effective Bug Hunting in Multithreaded Programs". In: *USENIX Security*. SEC. USENIX, 2020, pp. 2325–2342.

[36] Ju Chen et al. “JIGSAW: Efficient and Scalable Path Constraints Fuzzing”. In: *Security and Privacy*. S&P. IEEE, 2022, pp. 1531–1531. DOI: 10.1109/SP46214.2022.00102.

[37] Peng Chen and Hao Chen. “Angora: Efficient Fuzzing by Principled Search”. In: *Security and Privacy*. S&P. IEEE, 2018, pp. 711–725. DOI: 10.1109/SP.2018.00046.

[38] Xi Chen, Asia Slowinska, and Herbert Bos. “On the Detection of Custom Memory Allocators in C Binaries”. In: *Empirical Softw. Engg.* 21.3 (2016), pp. 753–777. DOI: 10.1007/s10664-015-9362-z.

[39] Yaohui Chen et al. “MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing”. In: *Research in Attacks, Intrusions and Defenses*. RAID. 2020, pp. 77–92.

[40] Yaohui Chen et al. “PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary”. In: *Asia Computer and Communications Security*. AsiaCCS. ACM, 2019, pp. 633–645. DOI: 10.1145/3321705.3329828.

[41] Yaohui Chen et al. “SAVIOR: Towards Bug-Driven Hybrid Testing”. In: *Security and Privacy*. S&P. IEEE, 2020, pp. 1580–1596. DOI: 10.1109/SP40000.2020.00002.

[42] Yiqun T. Chen et al. “Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size”. In: *Automated Software Engineering*. ASE. ACM, 2021, pp. 237–249. DOI: 10.1145/3324884.3416667.

[43] Yongheng Chen et al. “One Engine to Fuzz ’em All: Generic Language Processor Testing with Semantic Validation”. In: *Security and Privacy*. S&P. IEEE, 2021, pp. 642–658. DOI: 10.1109/SP40001.2021.00071.

[44] Yuanliang Chen et al. “EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers”. In: *USENIX Security*. USENIX, 2019, pp. 1967–1983.

[45] B. Chess and G. McGraw. “Static analysis for security”. In: *Security and Privacy*. S&P 2.6 (2004), pp. 76–79. DOI: 10.1109/MSP.2004.111.

[46] Jaeseung Choi et al. “Grey-Box Concolic Testing on Binary Code”. In: *International Conference on Software Engineering*. ICSE. 2019, pp. 736–747. DOI: 10.1109/ICSE.2019.00082.

[47] Zheng Leong Chua et al. “One Engine To Serve ’em All: Inferring Taint Rules Without Architectural Semantics”. In: *Network and Distributed System Security*. NDSS. The Internet Society, 2019. DOI: 10.14722/ndss.2019.23339.

[48] Edmund M. Clarke, Orna Grumberg, and David E. Long. “Model Checking and Abstraction”. In: *Transactions on Programming Languages and Systems*. TOPLAS 16.5 (1994), pp. 1512–1542. DOI: 10.1145/186025.186051.

[49] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. “MemFuzz: Using Memory Accesses to Guide Fuzzing”. In: *Software Testing, Validation and Verification*. ICST. 2019, pp. 48–58. DOI: 10.1109/ICST.2019.00015.

[50] Baozeng Ding et al. "Baggy Bounds with Accurate Checking". In: *International Symposium on Software Reliability Engineering Workshops*. ISSREW. 2012, pp. 195–200. DOI: 10.1109/ISSREW.2012.24.

[51] Ren Ding et al. "Hardware Support to Improve Fuzzing Performance and Precision". In: *Computer and Communications Security*. CCS. ACM, 2021, pp. 2214–2228. DOI: 10.1145/3460120.3484573.

[52] Zhen Yu Ding and Claire Le Goues. "An Empirical Study of OSS-Fuzz Bugs". In: *Mining Software Repositories*. MSR. 2021, pp. 131–142. DOI: 10.1109/MSR52588.2021.00026.

[53] Sung Ta Dinh et al. "Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases". In: *Network and Distributed System Security*. NDSS. The Internet Society, 2021. DOI: 10.14722/ndss.2021.24224.

[54] Dino Distefano et al. "Scaling Static Analyses at Facebook". In: *Communications of the ACM*. CACM 62.8 (2019), pp. 62–70. DOI: 10.1145/3338112.

[55] Brendan Dolan-Gavitt et al. "LAVA: Large-Scale Automated Vulnerability Addition". In: *Security and Privacy*. S&P. IEEE, 2016, pp. 110–121. DOI: 10.1109/SP.2016.15.

[56] Thomas Dullien. "Weird Machines, Exploitability, and Provable Unexploitability". In: *Transactions on Emerging Topics in Computing*. TETC 8.2 (2020), pp. 391–403. DOI: 10.1109/TETC.2017.2785299.

[57] Bradley Efron. "Bootstrap Methods: Another Look at the Jackknife". In: *The Annals of Statistics* 7.1 (1979), pp. 1–26. DOI: 10.1214/aos/1176344552.

[58] Facebook. *HHVM*. 2022. URL: https://github.com/facebook/hhvm (visited on 05/20/2022).

[59] Brandon Falk. *Fuzzing: Corpus Minimization*. 2021. URL: https://youtu.be/947b0lgyvJs (visited on 05/24/2022).

[60] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *Transactions on Programming Languages and Systems*. TOPLAS 9.3 (1987), pp. 319–349. DOI: 10.1145/24039.24041.

[61] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. "The Use of Likely Invariants as Feedback for Fuzzers". In: *USENIX Security*. SEC. USENIX, 2021, pp. 2829–2846.

[62] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. "WEIZZ: Automatic Grey-Box Fuzzing for Structured Binary Formats". In: *International Symposium on Software Testing and Analysis*. ISSTA. 2020, pp. 1–13. DOI: 10.1145/3395363.3397372.

[63] Andrea Fioraldi et al. “AFL++: Combining Incremental Steps of Fuzzing Research”. In: *Workshop on Offensive Technologies*. WOOT. 2020.

[64] Antonio Flores-Montoya and Eric Schulte. “Datalog Disassembly”. In: *USENIX Security*. SEC. USENIX, 2020, pp. 1075–1092.

[65] Phyllis G. Frankl and Stewart N. Weiss. “An experimental comparison of the effectiveness of branch testing and data flow testing”. In: *Transactions on Software Engineering*. TSE 19.8 (1993), pp. 774–787. DOI: 10.1109/32.238581.

[66] Galois Inc. *cclyzer++*. 2022. URL: https://galoisinc.github.io/cclyzerpp/ (visited on 04/09/2023).

[67] Shuitao Gan et al. “CollAFL: Path Sensitive Fuzzing”. In: *Security and Privacy*. S&P. IEEE, 2018, pp. 679–696. DOI: 10.1109/SP.2018.00040.

[68] Shuitao Gan et al. “GREYONE: Data Flow Sensitive Fuzzing”. In: *USENIX Security*. SEC. USENIX, 2020, pp. 2577–2594.

[69] Vijay Ganesh, Tim Leek, and Martin Rinard. “Taint-Based Directed Whitebox Fuzzing”. In: *International Conference on Software Engineering*. ICSE. 2009, pp. 474–484. DOI: 10.1109/ICSE.2009.5070546.

[70] Wentao Gao et al. “Beyond the Coverage Plateau: A Comprehensive Study of Fuzz Blockers (Registered Report)”. In: *Fuzzing Workshop*. FUZZING. ACM, 2023, pp. 47–55. DOI: 10.1145/3605157.3605177.

[71] GitLab. *GitLab Protocol Fuzzer Community Edition*. 2022. URL: https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce (visited on 06/23/2022).

[72] *GNU Binutils*. 2023. URL: https://www.gnu.org/software/binutils/ (visited on 02/26/2023).

[73] Patrice Godefroid, Michael Y. Levin, and David Molnar. “SAGE: Whitebox Fuzzing for Security Testing”. In: *Queue* 10.1 (2012), pp. 20–27. DOI: 10.1145/2090147.2094081.

[74] Philipp Goerz et al. “Systematic Assessment of Fuzzers using Mutation Analysis”. In: *USENIX Security*. SEC. USENIX, 2023.

[75] Google. *Fuzzer Test Suite*. 2021. URL: https://github.com/google/fuzzer-test-suite (visited on 05/10/2022).

[76] Rahul Gopinath, Carlos Jensen, and Alex Groce. “Code Coverage for Suite Evaluation by Developers”. In: *International Conference on Software Engineering*. ICSE. 2014, pp. 72–82. DOI: 10.1145/2568225.2568278.

[77] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. “QuickFuzz: An Automatic Random Fuzzer for Common File Formats”. In: *Haskell*. HASKELL. 2016, pp. 13–20. DOI: 10.1145/2976002.2976017.

[78] Emre Güler et al. "Cupid: Automatic Fuzzer Selection for Collaborative Fuzzing". In: *Annual Computer Security Applications Conference*. ACSAC. ACM, 2020, pp. 360–372. DOI: 10.1145/3427228.3427266.

[79] Arie Gurfinkel and Jorge A. Navas. "Abstract Interpretation of LLVM with a Region-Based Memory Model". In: *Software Verification*. VSTTE. Springer International Publishing, 2021, pp. 122–144. DOI: 10.1007/978-3-030-95561-8_8.

[80] Arie Gurfinkel et al. "The SeaHorn Verification Framework". In: *Computer Aided Verification*. CAV. Springer International Publishing, 2015, pp. 343–361. DOI: 10.1007/978-3-319-21690-4_20.

[81] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. "CodeQuest: Scalable Source Code Queries with Datalog". In: *European Conference on Object-Oriented Programming*. ECOOP. Springer Berlin Heidelberg, 2006, pp. 2–27. DOI: 10.1007/11785477_2.

[82] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. "CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines". In: *Network and Distributed System Security*. NDSS. The Internet Society, 2019. DOI: 10.14722/ndss.2019.23263.

[83] Nikolas Havrikov and Andreas Zeller. "Systematically Covering Input Structure". In: *Automated Software Engineering*. ASE. IEEE, 2020, pp. 189–199. DOI: 10.1109/ASE.2019.00027.

[84] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. "Magma: A Ground-Truth Fuzzing Benchmark". In: *Measurement and Analysis of Computing Systems*. POMACS 4.3 (2020). DOI: 10.1145/3428334.

[85] Xiaoyu He et al. "SoFi: Reflection-Augmented Fuzzing for JavaScript Engines". In: *Computer and Communications Security*. CCS. ACM, 2021, pp. 2229–2242. DOI: 10.1145/3460120.3484823.

[86] Mats P. E. Heimdahl and Devaraj George. "Test-Suite Reduction for Model Based Tests: Effects on Test Quality and Implications for Testing". In: *Automated Software Engineering*. ASE. IEEE, 2004, pp. 176–185. DOI: 10.1109/ASE.2004.1342735.

[87] Aki Helin. *Radamsa*. 2022. URL: https://gitlab.com/akihe/radamsa (visited on 06/17/2022).

[88] Hadi Hemmati. "How Effective Are Code Coverage Criteria?" In: *Software Quality, Reliability and Security*. QRS. 2015, pp. 151–156. DOI: 10.1109/QRS.2015.30.

[89] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *Computer Architecture News* 34.4 (2006), pp. 1–17. DOI: 10.1145/1186736.1186737.

[90] Adrian Herrera, Mathias Payer, and Antony L. Hosking. "datAFLow: Toward a Data-Flow-Guided Fuzzer". In: *Transactions on Software Engineering Methodology*. TOSEM 1.1 (2023). DOI: 10.1145/3587156.

[91] Adrian Herrera, Mathias Payer, and Antony L. Hosking. "datAFLow: Towards a Data-Flow-Guided Fuzzer". In: *Fuzzing Workshop*. FUZZING. The Internet Society, 2022. DOI: 10.14722/fuzzing.2022.23001.

[92] Adrian Herrera et al. "Seed Selection for Successful Fuzzing". In: *International Symposium on Software Testing and Analysis*. ISSTA. 2021, pp. 230–243. DOI: 10.1145/3460319.3464795.

[93] Joseph R. Horgan and Saul London. "Data Flow Coverage and the C Language". In: *Testing, Analysis, and Verification*. TAV. 1991, pp. 87–97. DOI: 10.1145/120807.120815.

[94] Joseph R. Horgan, Saul London, and Michael R. Lyu. "Achieving software quality with testing coverage measures". In: *Computer* 27.9 (1994), pp. 60–69. DOI: 10.1109/2.312032.

[95] Katherine Hough and Jonathan Bell. "A Practical Approach for Dynamic Taint Tracking with Control-Flow Relationships". In: *Transactions on Software Engineering Methodology*. TOSEM 31.2 (2021). DOI: 10.1145/3485464.

[96] *How SQLite Is Tested*. 2022. URL: https://www.sqlite.org/testing.html (visited on 02/03/2023).

[97] Chin-Chia Hsu et al. "InsTrim: Lightweight Instrumentation for Coverage-guided Fuzzing". In: *Binary Analysis Research*. BAR. The Internet Society, 2018. DOI: 10.14722/bar.2018.23014.

[98] Hwa-You Hsu and Alessandro Orso. "MINTS: A General Framework and Tool for Supporting Test-Suite Minimization". In: *International Conference on Software Engineering*. ICSE. 2009, pp. 419–429. DOI: 10.1109/ICSE.2009.5070541.

[99] Heqing Huang et al. "BEACON: Directed Grey-Box Fuzzing with Provable Path Pruning". In: *Security and Privacy*. S&P. IEEE, 2022, pp. 104–118. DOI: 10.1109/SP46214.2022.00007.

[100] Heqing Huang et al. "Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction". In: *Security and Privacy*. S&P. IEEE, 2020, pp. 1613–1627. DOI: 10.1109/SP40000.2020.00063.

[101] Monica Hutchins et al. "Experiments on the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria". In: *International Conference on Software Engineering*. ICSE. 1994, pp. 191–200. DOI: 10.1109/ICSE.1994.296778.

[102] Laura Inozemtseva and Reid Holmes. "Coverage is Not Strongly Correlated with Test Suite Effectiveness". In: *International Conference on Software Engineering*. ICSE. 2014, pp. 435–445. DOI: 10.1145/2568225.2568271.

[103] Kyriakos K. Ispoglou et al. "FuzzGen: Automatic Fuzzer Generation". In: *USENIX Security*. SEC. USENIX, 2020, pp. 2271–2287.

[104] Yuseok Jeon et al. "FuZZan: Efficient Sanitizer Metadata Design for Fuzzing". In: *Annual Technical Conference*. ATC. 2020, pp. 249–263.

[105] Yuseok Jeon et al. "HexType: Efficient Detection of Type Confusion Errors for C++". In: *Computer and Communications Security*. CCS. ACM, 2017, pp. 2373–2387. DOI: 10.1145/3133956.3134062.

[106] Ranjit Jhala and Rupak Majumdar. "Software Model Checking". In: *Computing Surveys*. CSUR 41.4 (2009), pp. 1–54. DOI: 10.1145/1592434.1592438.

[107] Zhiyuan Jiang et al. "Evocatio: Conjuring Bug Capabilities from a Single PoC". In: *Computer and Communications Security*. CCS. ACM, 2022, pp. 1599–1613. DOI: 10.1145/3548606.3560575.

[108] Zhiyuan Jiang et al. "Igor: Crash Deduplication Through Root-Cause Clustering". In: *Computer and Communications Security*. CCS. ACM, 2021, pp. 3318–3336. DOI: 10.1145/3460120.3485364.

[109] Brittany Johnson et al. "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?" In: *International Conference on Software Engineering*. ICSE. IEEE, 2013, pp. 672–681. DOI: 10.5555/2486788.2486877.

[110] Edward L. Kaplan and Paul Meier. "Nonparametric Estimation from Incomplete Observations". In: *Journal of the American Statistical Association* 53.282 (1958), pp. 457–481. ISSN: 01621459. DOI: 10.2307/2281868.

[111] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations*. 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[112] Kenney. *Kenney*. 2022. URL: https://www.kenney.nl/ (visited on 05/20/2022).

[113] Yonit Kesten and Amir Pnueli. "Control and data abstraction: the cornerstones of practical formal verification". In: *Software Tools for Technology Transfer*. STTT 2.4 (2000), pp. 328–342. DOI: 10.1007/s100090050040.

[114] Se-Won Kim, Xavier Rival, and Sukyoung Ryu. "A Theoretical Foundation of Sensitivity in an Abstract Interpretation Framework". In: *Transactions on Programming Languages and Systems*. TOPLAS 40.3 (2018). DOI: 10.1145/3230624.

[115] George Klees et al. "Evaluating Fuzz Testing". In: *Computer and Communications Security*. CCS. ACM, 2018, pp. 2123–2138. DOI: 10.1145/3243734.3243804.

[116] David G. Kleinbaum and Mitchel Klein. *Survival Analysis*. 2012. DOI: 10.1007/978-1-4419-6646-9.

[117] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems". In: *Software Analysis, Evolution, and Reengineering*. SANER. 2015, pp. 560–564. DOI: 10.1109/SANER.2015.7081877.

[118] Erik van der Kouwe et al. "SoK: Benchmarking Flaws in Systems Security". In: *European Security and Privacy*. EuroS&P. IEEE, 2019, pp. 310–325. DOI: 10.1109/EuroSP.2019.00031.

[119] Jakub Kuderski, Jorge A. Navas, and Arie Gurfinkel. "Unification-based Pointer Analysis without Oversharing". In: *Formal Methods in Computer Aided Design*. FMCAD. IEEE, 2019, pp. 37–45. DOI: 10.23919/FMCAD.2019.8894275.

[120] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Code Generation and Optimization*. CGO. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[121] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. "Constraint-guided Directed Greybox Fuzzing". In: *USENIX Security*. SEC. USENIX, 2021, pp. 3559–3576.

[122] Yuekang Li et al. "Cerebro: Context-Aware Adaptive Fuzzing for Effective Vulnerability Detection". In: *European Software Engineering Conference/Foundations of Software Engineering*. ESEC/FSE. 2019, pp. 533–544. DOI: 10.1145/3338906.3338975.

[123] Yuekang Li et al. "Steelix: Program-State Based Binary Fuzzing". In: *European Software Engineering Conference/Foundations of Software Engineering*. ESEC/FSE. 2017, pp. 627–637. DOI: 10.1145/3106237.3106295.

[124] Yuwei Li et al. "UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers". In: *USENIX Security*. SEC. USENIX, 2021, pp. 2777–2794.

[125] Jie Liang et al. "PATA: Fuzzing with Path Aware Taint Analysis". In: *Security and Privacy*. S&P. IEEE, 2022, pp. 154–170. DOI: 10.1109/SP46214.2022.00010.

[126] Jun-Wei Lin et al. "Nemo: Multi-Criteria Test-Suite Minimization with Integer Nonlinear Programming". In: *International Conference on Software Engineering*. ICSE. 2018, pp. 1039–1049. DOI: 10.1145/3180155.3180174.

[127] Stephan Lipp et al. "FuzzTastic: A Fine-Grained, Fuzzer-Agnostic Coverage Analyzer". In: *International Conference on Software Engineering: Companion Proceedings*. ICSE. ACM, 2022, pp. 75–79. DOI: 10.1145/3510454.3516847.

[128] Zhengyang Liu and John Criswell. "Flexible and Efficient Memory Object Metadata". In: *International Symposium on Memory Management*. ISMM. 2017, pp. 36–46. DOI: 10.1145/3092255.3092268.

[129] Danushka Liyanage et al. "Reachable Coverage: Estimating Saturation in Fuzzing". In: *International Conference on Software Engineering*. ICSE. ACM, 2023.

[130] LLVM. *libFuzzer - a library for coverage-guided fuzz testing.* 2022. URL: https://llvm.org/docs/LibFuzzer.html (visited on 06/23/2022).

[131] Lunge Technology. *Cyber Grand Challenge Corpus*. 2017. URL: http://www.lungetech.com/cgc-corpus/ (visited on 05/22/2022).

[132] Simon Luo et al. "Make out like a (Multi-Armed) Bandit: Improving the Odds of Fuzzer Seed Scheduling with T-Scheduler". In: *Asia Computer and Communications Security*. AsiaCCS. 2024.

[133] Chenyang Lyu et al. "EMS: History-Driven Mutation for Coverage-based Fuzzing". In: *Network and Distributed System Security*. NDSS. The Internet Society, 2022. DOI: 10.14722/ndss.2022.23162.

[134] Chenyang Lyu et al. "MOPT: Optimized Mutation Scheduling for Fuzzers". In: *USENIX Security*. SEC. USENIX, 2019, pp. 1949–1966.

[135] Chenyang Lyu et al. "SmartSeed: Smart Seed Generation for Efficient Fuzzing". In: *arXiv preprint* (2019). DOI: 10.48550/arXiv.1807.02606.

[136] Laurens van der Maaten and Geoffrey Hinton. "Visualizing Data using t-SNE". In: *Journal of Machine Learning Research* 9.86 (2008), pp. 2579–2605.

[137] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. "Ankou: Guiding Grey-Box Fuzzing towards Combinatorial Difference". In: *International Conference on Software Engineering*. ICSE. 2020, pp. 1024–1036. DOI: 10.1145/3377811.3380421.

[138] Valentin J. M. Manès et al. "The Art, Science, and Engineering of Fuzzing: A Survey". In: *Transactions on Software Engineering*. TSE 47.11 (2021), pp. 2312–2331. DOI: 10.1109/TSE.2019.2946563.

[139] Henry B. Mann and Donald R. Whitney. "On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other". In: *The Annals of Mathematical Statistics* 18.1 (1947), pp. 50–60. DOI: 10.1214/aoms/1177730491.

[140] Nathan Mantel. "Evaluation of survival data and two new rank order statistics arising in its consideration". In: *Cancer Chemotherapy Reports* 50.3 (1966), pp. 163–170.

[141] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. "Fuzzing with Data Dependency Information". In: *European Security and Privacy*. EuroS&P. IEEE, 2022, pp. 286–302. DOI: 10.1109/EuroSP53844.2022.00026.

[142] Björn Mathis et al. "Parser-Directed Fuzzing". In: *Programming Language Design and Implementation*. PLDI. 2019, pp. 548–560. DOI: 10.1145/3314221.3314651.

[143] Ruijie Meng et al. "Linear-time Temporal Logic guided Greybox Fuzzing". In: *International Conference on Software Engineering*. ICSE. 2022. DOI: 10.1145/3510003.3510082.

[144] Jonathan Metzman et al. "FuzzBench: An Open Fuzzer Benchmarking Platform and Service". In: *European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE. 2021, pp. 1393–1403. DOI: 10.1145/3468264.3473932.

[145] Xianya Mi et al. "LeanSym: Efficient Hybrid Fuzzing Through Conservative Constraint Debloating". In: *Research in Attacks, Intrusions and Defenses*. RAID. 2021, pp. 62–77. DOI: 10.1145/3471621.3471852.

[146] Bart Miller. *CS736 Projects*. 1988. URL: https://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf (visited on 08/06/2022).

[147] Matt Miller. *Trends and Challenges in the Vulnerability Mitigation Landscape*. 2019.

[148] The Motion Monkey. *Free Retro Arcade Video Game Sound Effects*. 2017. URL: https://www.themotionmonkey.co.uk/free-resources/retro-arcade-sounds/ (visited on 05/20/2022).

[149] Mozilla. *Fuzzing—Test Samples*. 2020. URL: https://firefox-source-docs.mozilla.org/tools/fuzzing/index.html (visited on 05/11/2022).

[150] Mozilla. *Introducing the ASan Nightly Project*. 2018. URL: https://blog.mozilla.org/security/2018/07/19/introducing-the-asan-nightly-project/ (visited on 05/11/2022).

[151] Stefan Nagy and Matthew Hicks. "Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing". In: *Security and Privacy*. S&P. IEEE, 2019, pp. 787–802. DOI: 10.1109/SP.2019.00069.

[152] Stefan Nagy et al. "Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing". In: *USENIX Security*. SEC. USENIX, 2021, pp. 1683–1700.

[153] Stefan Nagy et al. "Same Coverage, Less Bloat: Accelerating Binary-Only Fuzzing with Coverage-Preserving Coverage-Guided Tracing". In: *Computer and Communications Security*. CCS. ACM, 2021, pp. 351–365. DOI: 10.1145/3460120.3484787.

[154] George C. Necula et al. "CCured: Type-Safe Retrofitting of Legacy Software". In: *Transactions on Programming Languages and Systems*. TOPLAS 27.3 (2005), pp. 477–526. DOI: 10.1145/1065887.1065892.

[155] Neelofar et al. "Instance Space Analysis of Search-Based Software Testing". In: *IEEE Transactions on Software Engineering*. TSE (2022), pp. 1–20. DOI: 10.1109/TSE.2022.3228334.

[156] Manh-Dung Nguyen et al. “Binary-level Directed Fuzzing for Use-After-Free Vulnerabilities”. In: *Research in Attacks, Intrusions and Defenses*. RAID. 2020, pp. 47–62.

[157] Ivica Nikolić et al. “Refined Grey-Box Fuzzing with Sivo”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. DIMVA. 2021, pp. 106–129. DOI: 10.1007/978-3-030-80825-9_6.

[158] NIST. *CVE-2021-43527*. 2021. URL: https://nvd.nist.gov/vuln/detail/CVE-2021-43527 (visited on 02/13/2023).

[159] Tim Nosco et al. “The Industrial Age of Hacking”. In: *USENIX Security*. SEC. USENIX, 2020, pp. 1129–1146.

[160] Carlos Oliveira et al. “Mapping the Effectiveness of Automated Test Suite Generation Techniques”. In: *IEEE Transactions on Reliability*. TR 67.3 (2018), pp. 771–785. DOI: 10.1109/TR.2018.2832072.

[161] *Open Game Art*. 2022. URL: https://opengameart.org/ (visited on 05/20/2022).

[162] Tavis Ormandy. *This shouldn't have happened: A vulnerability postmortem*. 2021. URL: https://googleprojectzero.blogspot.com/2021/12/this-shouldnt-have-happened.html (visited on 02/12/2023).

[163] Sebastian Österlund et al. “ParmeSan: Sanitizer-Guided Greybox Fuzzing”. In: *USENIX Security*. SEC. USENIX, 2020, pp. 2289–2306.

[164] Rohan Padhye et al. “FuzzFactory: Domain-Specific Fuzzing with Waypoints”. In: *Programming Languages*. OOPSLA 3 (2019). DOI: 10.1145/3360600.

[165] Rohan Padhye et al. “Semantic Fuzzing with Zest”. In: *International Symposium on Software Testing and Analysis*. ISSTA. 2019, pp. 329–340. DOI: 10.1145/3293882.3330576.

[166] Shankara Pailoor, Andrew Aday, and Suman Jana. “Moonshine: Optimizing OS Fuzzer Seed Selection with Trace Distillation”. In: *USENIX Security*. SEC. USENIX, 2018, pp. 729–743.

[167] Weiyu Pan et al. “Grammar-Agnostic Symbolic Execution by Token Symbolization”. In: *International Symposium on Software Testing and Analysis*. ISSTA. 2021, pp. 374–387. DOI: 10.1145/3460319.3464845.

[168] Soyeon Park et al. “Fuzzing JavaScript Engines with Aspect-preserving Mutation”. In: *Security and Privacy*. S&P. IEEE, 2020, pp. 1629–1642. DOI: 10.1109/SP40000.2020.00067.

[169] Terence Parr and Kathleen Fisher. “LL(*): The Foundation of the ANTLR Parser Generator”. In: *Programming Language Design and Implementation*. PLDI. 2011, pp. 425–436. DOI: 10.1145/1993498.1993548.

[170] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. “T-Fuzz: Fuzzing by Program Transformation”. In: *Security and Privacy*. S&P. IEEE, 2018, pp. 697–710. DOI: 10.1109/SP.2018.00056.

[171] Van-Thuan Pham et al. “Towards Systematic and Dynamic Task Allocation for Collaborative Parallel Fuzzing”. In: *Automated Software Engineering*. ASE. 2021, pp. 1337–1341. DOI: 10.1109/ASE51524.2021.9678810.

[172] Daniel Plohmann et al. “Malpedia: A Collaborative Effort to Inventorize the Malware Landscape”. In: *Journal on Cybercrime & Digital Investigations* 3.1 (2018). DOI: 10.18464/cybin.v3i1.17.

[173] Sebastian Poeplau and Aurélien Francillon. “Symbolic execution with SymCC: Don’t interpret, compile!” In: *USENIX Security*. SEC. USENIX, 2020, pp. 181–198.

[174] Sebastian Poeplau and Aurélien Francillon. “SymQEMU: Compilation-based symbolic execution for binaries”. In: *Network and Distributed System Security*. NDSS. The Internet Society, 2021. DOI: 10.14722/ndss.2021.23118.

[175] Sandra Rapps and Elaine J. Weyuker. “Selecting Software Test Data Using Data Flow Information”. In: *Transactions on Software Engineering*. TSE 11.4 (1985), pp. 367–375. DOI: 10.1109/TSE.1985.232226.

[176] Sanjay Rawat et al. “VUzzer: Application-aware Evolutionary Fuzzing”. In: *Network and Distributed System Security*. NDSS. The Internet Society, 2017. DOI: 10.14722/ndss.2017.23404.

[177] Alexandre Rebert et al. “Optimizing Seed Selection for Fuzzing”. In: *USENIX Security*. SEC. USENIX, 2014, pp. 861–875.

[178] *RegExLib: Regular Expression Library*. 2022. URL: https://regexlib.com/ (visited on 05/20/2022).

[179] Thomas W. Reps and Raghu Ramakrishnan. “Demand Interprocedural Program Analysis Using Logic Databases”. In: *Applications of Logic Databases*. Springer, 1995, pp. 163–196. DOI: 10.1007/978-1-4615-2207-2_8.

[180] H. G. Rice. “Classes of Recursively Enumerable Sets and Their Decision Problems”. In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366. DOI: 10.1090/s0002-9947-1953-0053041-6.

[181] Matt Ruhstaller, TPM, and Oliver Chang. *A New Chapter for OSS-Fuzz*. 2018. URL: https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html (visited on 05/11/2022).

[182] Caitlin Sadowski et al. “Lessons from Building Static Analysis Tools at Google”. In: *Communications of the ACM*. CACM 61.4 (2018), pp. 58–66. DOI: 10.1145/3188720.

[183] Christopher Salls et al. “Exploring Abstraction Functions in Fuzzing”. In: *Communications and Network Security*. CNS. 2020, pp. 1–9. DOI: 10.1109/CNS48642.2020.9162273.

[184] Christopher Salls et al. “Token-Level Fuzzing”. In: *USENIX Security*. SEC. USENIX, 2021, pp. 2795–2809.

[185] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. "PhASAR: An Inter-procedural Static Analysis Framework for C/C++". In: *Tools and Algorithms for the Construction and Analysis of Systems*. TACAS. Springer International Publishing, 2019, pp. 393–410.

[186] Sergej Schumilo et al. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types". In: *USENIX Security*. SEC. USENIX, 2021, pp. 2597–2614.

[187] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *Annual Technical Conference*. ATC. 2012, pp. 309–318.

[188] Kosta Serebryany. "Continuous Fuzzing with libFuzzer and AddressSanitizer". In: *Cybersecurity Development*. SecDev. 2016, pp. 157–157. DOI: 10.1109/SecDev.2016.043.

[189] Kostya Serebryany. "OSS-Fuzz - Google's continuous fuzzing service for open source software". In: *USENIX Security*. SEC. USENIX, 2017.

[190] Dongdong She, Abhishek Shah, and Suman Jana. "Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis". In: *Security and Privacy*. S&P. IEEE, 2022, pp. 1558–1558. DOI: 10.1109/SP46214.2022.00129.

[191] Dongdong She et al. "MTFuzz: Fuzzing with a Multi-Task Neural Network". In: *European Software Engineering Conference and Foundations of Software Engineering*. ESEC/FSE. 2020, pp. 737–749. DOI: 10.1145/3368089.3409723.

[192] Dongdong She et al. "Neutaint: Efficient Dynamic Taint Analysis with Neural Networks". In: *Security and Privacy*. S&P. IEEE, 2022, pp. 1527–1543. DOI: 10.1109/SP40000.2020.00022.

[193] Dongdong She et al. "NEUZZ: Efficient Fuzzing with Neural Program Smoothing". In: *Security and Privacy*. IEEE, 2019, pp. 803–817. DOI: 10.1109/SP.2019.00052.

[194] Yan Shoshitaishvili et al. "(State of) The Art of War: Offensive Techniques in Binary Analysis". In: *Security and Privacy*. S&P. IEEE, 2016, pp. 138–157. DOI: 10.1109/SP.2016.17.

[195] Laurent Simon and Akash Verma. "Improving Fuzzing through Controlled Compilation". In: *European Security and Privacy*. EuroS&P. IEEE, 2020, pp. 34–52. DOI: 10.1109/EuroSP48549.2020.00011.

[196] Software Engineering Institute. *CERT BFF*. 2016. URL: https://www.cert.org/vulnerability-analysis/tools/bff.cfm (visited on 06/17/2022).

[197] Dokyung Song et al. "SoK: Sanitizing for Security". In: *Security and Privacy*. S&P. IEEE, 2019, pp. 1275–1295. DOI: 10.1109/SP.2019.00010.

[198] Suhwan Song et al. "CrFuzz: Fuzzing Multi-Purpose Programs through Input Validation". In: *European Software Engineering Conference and Foundations of*

Software Engineering. ESEC/FSE. 2020, pp. 690–700. DOI: 10.1145/3368089.3409769.

[199] Prashast Srivastava and Mathias Payer. "Gramatron: Effective Grammar-Aware Fuzzing". In: *International Symposium on Software Testing and Analysis*. ISSTA. 2021, pp. 244–256. DOI: 10.1145/3460319.3464814.

[200] JHU/APL Staff. *Assembled Labeled Library for Static Analysis Research (ALLSTAR) Dataset*. 2019. URL: https://allstar.jhuapl.edu/ (visited on 05/19/2022).

[201] Nick Stephens et al. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution". In: *Network and Distributed System Security*. NDSS. The Internet Society, 2016. DOI: 10.14722/ndss.2016.23368.

[202] Ting Su et al. "A Survey on Data-Flow Testing". In: *Computing Surveys*. CSUR 50.1 (2017). DOI: 10.1145/3020266.

[203] Yulei Sui and Jingling Xue. "On-Demand Strong Update Analysis via Value-Flow Refinement". In: *Foundations of Software Engineering*. FSE. ACM, 2016, pp. 460–473. DOI: 10.1145/2950290.2950296.

[204] Yulei Sui and Jingling Xue. "SVF: Interprocedural Static Value-Flow Analysis in LLVM". In: *Compiler Construction*. CC. 2016, pp. 265–266. DOI: 10.1145/2892208.2892235.

[205] William N. Sumner and Xiangyu Zhang. "Identifying Execution Points for Dynamic Analyses". In: *Automated Software Engineering*. ASE. IEEE, 2013, pp. 81–91. DOI: 10.1109/ASE.2013.6693069.

[206] William N. Sumner et al. "Precise Calling Context Encoding". In: *International Conference on Software Engineering*. ICSE. 2010, pp. 525–534. DOI: 10.1145/1806799.1806875.

[207] Robert Swiecki. *honggfuzz*. 2016. URL: https://honggfuzz.dev/ (visited on 05/19/2022).

[208] The Clang Team. *SanitizerCoverage*. 2022. URL: https://clang.llvm.org/docs/SanitizerCoverage.html (visited on 06/14/2022).

[209] The Clang Team. *Source-Based Code Coverage*. 2022. URL: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html (visited on 05/22/2022).

[210] *UndefinedBehaviorSanitizer*. 2022. URL: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html (visited on 06/20/2022).

[211] Jonas Benedict Wagner. "Elastic Program Transformations Automatically Optimizing the Reliability/Performance Trade-off in Systems Software". In: (2017). DOI: 10.5075/epfl-thesis-7745.

[212] Daimeng Wang et al. "SyzVegas: Beating Kernel Fuzzing Odds with Reinforcement Learning". In: *USENIX Security*. SEC. USENIX, 2021, pp. 2741–2758.

[213] Jinghan Wang, Chengyu Song, and Heng Yin. "Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing". In: NDSS. The Internet Society, 2021. DOI: 10.14722/ndss.2021.24486.

[214] Jinghan Wang et al. "Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing". In: *Research in Attacks, Intrusions and Defenses*. RAID. 2019, pp. 1–15.

[215] Junjie Wang et al. "Skyfire: Data-Driven Seed Generation for Fuzzing". In: *Security and Privacy*. S&P. IEEE, 2017, pp. 579–594. DOI: 10.1109/SP.2017.23.

[216] Junjie Wang et al. "Superion: Grammar-Aware Greybox Fuzzing". In: *International Conference on Software Engineering*. ICSE. 2019, pp. 724–735. DOI: 10.1109/ICSE.2019.00081.

[217] Yanhao Wang et al. "Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization". In: *Network and Distributed System Security*. NDSS. The Internet Society, 2020. DOI: 10.14722/ndss.2020.24422.

[218] Jim Webber. "A Programmatic Introduction to Neo4j". In: *Systems, Programming, and Applications: Software for Humanity*. SPLASH. ACM, 2012, pp. 217–218. DOI: 10.1145/2384716.2384777.

[219] Cheng Wen et al. "MemLock: Memory Usage Guided Fuzzing". In: *International Conference on Software Engineering*. ICSE. 2020, pp. 765–777. DOI: 10.1145/3377811.3380396.

[220] W. Eric Wong et al. "Effect of test set minimization on fault detection effectiveness". In: *International Conference on Software Engineering*. ICSE. 1995, pp. 41–50. DOI: 10.1145/225014.225018.

[221] WordPress. *WordPress*. 2022. URL: https://github.com/WordPress/WordPress (visited on 05/20/2022).

[222] Mingyuan Wu et al. "Evaluating and Improving Neural Program-Smoothing-based Fuzzing". In: *International Conference on Software Engineering*. ICSE. 2022. DOI: 10.1145/3510003.3510089.

[223] Mingyuan Wu et al. "One Fuzzing Strategy to Rule Them All". In: *International Conference on Software Engineering*. ICSE. DOI: 10.1145/3510003.3510174.

[224] Bin Xin, William N. Sumner, and Xiangyu Zhang. "Efficient Program Execution Indexing". In: *Programming Language Design and Implementation*. PLDI. 2008, pp. 238–248. DOI: 10.1145/1375581.1375611.

[225] Wen Xu et al. "Designing New Operating Primitives to Improve Fuzzing Performance". In: *Computer and Communications Security*. CCS. ACM, 2017, pp. 2313–2328. DOI: 10.1145/3133956.3134046.

[226] Qian Yan et al. “InFuzz: An Interactive Tool for Enhancing Efficiency in Fuzzing through Visual Bottleneck Analysis (Registered Report)”. In: *Fuzzing Workshop*. FUZZING. ACM, 2023, pp. 56–61. DOI: 10.1145/3605157.3605847.

[227] Shengbo Yan et al. “PathAFL: Path-Coverage Assisted Fuzzing”. In: *Asia Computer and Communications Security*. AsiaCCS. ACM, 2020, pp. 598–609. DOI: 10.1145/3320269.3384736.

[228] Wei You et al. “SLF: Fuzzing without Valid Seed Inputs”. In: *International Conference on Software Engineering*. ICSE. 2019, pp. 712–723. DOI: 10.1109/ICSE.2019.00080.

[229] Tai Yue et al. “EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit”. In: *USENIX Security*. SEC. USENIX, 2020, pp. 2307–2324.

[230] Insu Yun et al. “QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing”. In: *USENIX Security*. SEC. USENIX, 2018, pp. 745–761.

[231] Michał Zalewski. *American Fuzzy Lop (AFL)*. 2015. URL: http://lcamtuf.coredump.cx/afl/ (visited on 05/19/2022).

[232] Gen Zhang et al. “MobFuz: Adaptive Multi-objective Optimization in Gray-box Fuzzing”. In: *Network and Distributed System Security*. NDSS. The Internet Society, 2022. DOI: 10.14722/ndss.2022.24314.

[233] Kunpeng Zhang et al. “Path Transitions Tell More: Optimizing Fuzzing Schedules via Runtime Program States”. In: *International Conference on Software Engineering*. ICSE. 2022. DOI: 10.1145/3510003.3510063.

[234] Zenong Zhang et al. “FIXREVERTER: A Realistic Bug Injection Methodology for Benchmarking Fuzz Testing”. In: *USENIX Security*. SEC. USENIX, 2022, pp. 3699–3715.

[235] Zenong Zhang et al. “Registered Report: Fuzzing Configurations of Program Options”. In: *Fuzzing Workshop*. FUZZING. The Internet Society, 2022. DOI: 10.14722/fuzzing.2022.23008.

[236] Zhuo Zhang et al. “StochFuzz: Sound and Cost-effective Fuzzing of Stripped Binaries by Incremental and Stochastic Rewriting”. In: *Security and Privacy*. S&P. IEEE, 2021, pp. 659–676. DOI: 10.1109/SP40001.2021.00109.

[237] Chijin Zhou et al. “Zeror: Speed up Fuzzing with Coverage-Sensitive Tracing and Scheduling”. In: *Automated Software Engineering*. ASE. 2020, pp. 858–870. DOI: 10.1145/3324884.3416572.

[238] Tong Zhou et al. “Valence: Variable Length Calling Context Encoding”. In: *Compiler Construction*. CC. ACM, 2019, pp. 147–158. DOI: 10.1145/3302516.3307351.

[239] Xiaogang Zhu and Marcel Böhme. "Regression Greybox Fuzzing". In: *Computer and Communications Security*. CCS. ACM, 2021, pp. 2169–2182. DOI: 10.1145/3460120.3484596.

[240] Peiyuan Zong et al. "FuzzGuard: Filtering out Unreachable Inputs in Directed Grey-Box Fuzzing through Deep Learning". In: *USENIX Security*. SEC. USENIX, 2020, pp. 2255–2269.

[241] Zyte. *Scrapy*. 2022. URL: https://scrapy.org/ (visited on 05/20/2022).