THESE DE DOCTORAT DE
SORBONNE UNIVERSITE
préparée à EURECOM

École doctorale EDITE de Paris n° ED130
Spécialité: «Informatique, Télécommunications et Électronique»

Sujet de la thèse:

Fuzzing in the 2020s: Novel Approaches and Solutions

Thèse présentée et soutenue à Biot, le 8/12/2023, par

Andrea Fioraldi

devant le jury composé de:

Rapporteur & Président	Prof. Mathias Payer	EPFL
Rapporteur	Prof. Marcel Böhme	Max Planck Institute for Security and Privacy
Examinateur	Prof. Marius Muench	University of Birmingham
Examinateur	Prof. Melek Önen	EURECOM
Directeur de thèse	Prof. Davide Balzarotti	EURECOM

Preface

My PhD journey has been filled with exciting challenges and incredible support from those around me. Solving these challenges would not have been possible without the people who have been there with me.

This thesis, the cumulative results of 3 years of work, joy, delusions, and countless chess matches, is the output of a collaborative effort between all the people around me and me, not only those who put a significant amount of work in advancing the state-of-the-art in fuzz testing, but also those who supported me with friendship and encouragement.

Aristotle stated that friendship is "one soul dwelling in two bodies"; thus, my contributions to the scientific community also belong to those who are always with me, even if physically not.

Acknowledgements

I want to express my heartfelt thanks to my family, mother, father, sister, and to who is not anymore here but encouraged me for all these years to be the best version of me possible.

No words are needed for my lifelong friends, I promise this is the last degree that I get.

Thank you to the new friends who were with me along this path, even if some of you just for a few months.

To my advisor, for dealing with a rather "uncommon" PhD student, and the colleagues who exchanged precious knowledge with me, thank you.

I'm grateful also to the organization, AFLplusplus, and its members, who shared with me most of the crazy things we did in these three years.

Lastly, I want to thank my fellow hacker's friends, mHACKeroni and more, we hacked a satellite, we rock.

Abstract

Security remains at risk due to elusive software vulnerabilities, even with extensive fuzzing efforts. Coverage-guided fuzzers, focusing solely on code coverage, often fall short in discovering specific vulnerabilities.

The proliferation of diverse fuzzing tools has fragmented the field, making it challenging to combine different fuzzing techniques, assess contributions accurately, and compare tools effectively. To address this, standardized baselines are needed to ensure equitable evaluations.

AFL, due to its popularity, is often extended to implement new prototypes despite not being a naive baseline and its monolithic design. On the other hand, custom fuzzers written from scratch tend to reinvent solutions and often lack scalability on multicore systems.

This thesis addresses these challenges with several contributions:

A new feedback mechanism called InvsCov is introduced, which considers program variable relationships and code coverage. It refines program state approximation for diverse bug detection. Another additional feedback we introduce explores data dependency graphs to enhance fuzzing by rewarding new dataflow edge traversal, effectively finding vulnerabilities missed by standard coverage. We also present a thorough analysis of AFL's internal mechanisms to shed light on its design choices and their impact on fuzzing performance. Finally, to address fragmentation, LibAFL is introduced as a modular and reusable fuzzing framework. Researchers can extend the core fuzzer pipeline, evaluation of compelling techniques, and combination of orthogonal approaches. An attempt to rewrite AFL++ as a frontend to LibAFL won the SBFT'23 fuzzing competition in the bug-finding track.

These contributions advance the field of fuzz testing, addressing the challenges of sensitivity in feedback mechanisms, bug diversity, tool fragmentation, and fuzzers evaluation. They provide a foundation for improving fuzzing techniques, enabling the detection of a broader range of bugs, and fostering collaboration and standardization within the community.

Résumé

Malgré les efforts considérables mis en œuvres en matière de fuzzing, la sécurité des logiciels informatiques reste menacée par des vulnérabilités insaisissables.

Les coverage-guided fuzzers, qui se concentrent uniquement sur la couverture de code, ne parviennent généralement pas à découvrir des vulnérabilités spécifiques.

La multiplication de divers outils de fuzzing a grandement divisé la communauté et a rendu difficile la combinaison de différentes techniques de fuzzing, l'évaluation précise des contributions et la comparaison efficace des différents outils. Pour remédier à cette situation, il est nécessaire de disposer d'un socle commun afin de garantir des évaluations précises et équitables.

AFL, en raison de sa popularité, est souvent utilisé comme base lors de la création de nouveaux prototypes malgré le fait qu'il ne s'agisse pas d'une référence naïve et sa conception monolithique. D'autre part, les fuzzers personnalisés réécrits à partir de zéro ont tendance à réinventer la roue et utilisent souvent inneficacement la puissance des les systèmes multicœurs.

Cette thèse relève ces défis en apportant plusieurs contributions:

Un nouveau mécanisme de feedback appelé InvsCov est introduit, qui prend en compte les relations entre les variables du programme et la couverture du code. Il affine l'approximation de l'état du programme pour optimiser la détection de divers bugs. Une autre approche que nous introduisons explore les graphes de dépendance des données pour améliorer le fuzzing en récompensant la traversée de nouvelles arêtes du graphe de flux de données, ce qui a permis la découverte de nouvelles vulnérabilités manquées par la couverture standard efficacement. Nous présentons également une analyse approfondie des mécanismes internes d'AFL afin de mettre en lumière ses choix de conception et leur impact sur les performances du fuzzing. Enfin, pour remédier au problème de fragmentation cité précédemment, nous proposons un outil dédié au fuzzing modulaire et réutilisable: LibAFL. Les chercheurs peuvent étendre la pipeline de base du fuzzer, l'évaluation

de nouvelles techniques techniques et la combinaison d'approches orthogonales. Une tentative de réécriture de AFL++ comme frontend de LibAFL a remporté le concours de fuzzing SBFT'23 dans le domaine de la recherche de bugs.

Ces contributions font progresser le domaine du fuzzing, en abordant les défis de la sensibilité des mécanismes de feedback, de la diversité des bugs découverts, de la fragmentation des outils et de l'évaluation des fuzzers. Elles fournissent une base solide pour l'amélioration des techniques de fuzzing, permettant la détection d'une plus large gamme de bugs, et favorisant la collaboration et la standardisation au sein de la communauté.

Contents

List of Figures

Chapter 1

Introduction

In the landscape of software development and security, the pursuit of robust and secure software has become more critical than ever before. Software vulnerabilities can lead to catastrophic consequences, including data breaches, system crashes, and compromised user privacy. In this context, the field of software testing has witnessed remarkable advancements to identify and mitigate vulnerabilities efficiently. Among these advancements, *Fuzzing* has emerged as a prominent dynamic testing technique. By generating unexpected inputs and injecting them into software, fuzzing aims to uncover latent vulnerabilities that traditional testing methods might miss.

The importance of fuzzing constantly increased in the last decade since the advent of American Fuzzy Lop (AFL) [175] that consecrated coverage-guided fuzzing, a twist of traditional fuzzing that takes into account the uncovered code regions as feedback, as preferred technique able to uncover bugs almost automatically, even without any knowledge of the input format [178].

The constant innovation contributed to its success, generating great attention in academia [111], with thousands of papers published in the last years, and in industry thanks to its great bug-finding performance. During the DARPA Cyber Grand Challenge [45], a competition to advance the development of systems for "automated, scalable, machine-speed vulnerability detection and patching" played in 2016, the most successful deployed solutions [156, 27, 32] were using a variant of coverage-guided fuzzing to implement the component responsible for automated vulnerability discovery. Google, with its OSS-Fuzz [3, 150] program, everyday runs fuzzers such as AFL++ [65] and LibFuzzer [105] to test more than one thousand open source projects at scale.

The popularity of fuzzing [128], compared to other techniques such as

static code analysis, can be also explained by the ability of fuzzers to produce proofs of vulnerabilities, crashing testcases that can be used by security researchers and developers to exploit or fix a vulnerability. The virtual absence of false positives and the ease of use of the tools made fuzzing a first-class citizen in security development cycle pipelines and not just a popular tool in the restricted community of security researchers.

The novelties in the field in the past years were constant, from the introduction of the coverage feedback to various techniques to increase the ability to uncover new program points [16, 138, 174], a necessary – but not sufficient – condition to find bugs, the introduction of models of the input space combined with the feedback to fuzz deeper parts of code [13, 72] and the engineering of solutions to automatically test multi-interaction programs such as network servers [131, 147].

Investigating the field of fuzz testing in the 2020s, after years of attention and constant improvements in the performance of fuzzers and with overall better security of the tested software, needs a shift in the angle from which we observe the field to discover that there are still important open challenges to address.

1.1 Challenges in Modern Fuzzing

Observing software vulnerabilities found by security researchers over the years, there are still some of them that affect well-fuzzed software and that fuzzers were unable to find [122].

One of the reasons why a coverage-guided fuzzer may struggle to detect bugs is that relying solely on code coverage is necessary but not sufficient for bug discovery. To cope with this challenge, more sensitive [162] feedbacks were proposed in literature, such as calling context-aware edge coverage, but while they mitigate the problem they don't fully solve them. The condition that triggers the fault can be implicit in coverage and so fuzzers need a specific feedback that reason on the whole program state.

Another kind of hard bugs for fuzzers are the ones going beyond memory corruption. Most of the fuzzing nowadays focuses on finding crashes or assertion errors, often with the aid of memory sanitizer [149, 64]. Logic bugs in memory unsafe code [99], but also other kinds of bugs in memory safe languages [40], are a work-in-progress challenge in fuzz testing that is gaining attention due to the recent expansion of this technique to the developers' world.

The proliferation of fuzzing tools, each with its own set of features and variations, has led to a fragmented landscape [111], an often underestimated

challenge in academia. This fragmentation hampers the effective combination of orthogonal fuzzing techniques, makes it challenging to assess individual contributions accurately, and complicates comparisons between different tools. To overcome these challenges there's a growing need to establish standardized baselines. These baselines serve as a common ground for evaluating and comparing fuzzing solutions, ensuring that assessments are conducted on a level playing field.

In the past, both industry and academia have gravitated towards AFL, a fuzzing tool renowned for its groundbreaking heuristics, and many enhancements have been built upon the AFL baseline to boost code coverage and bug discovery capabilities. This has led to a situation where AFL, although highly effective, is being used as a baseline for research when it is, in fact, a complex and well-engineered tool.

The problem of the current fuzzers regarding this problem, including AFL++, the most active and popular fork of AFL that we developed, is that they are designed to be tools and so without code reuse in mind. To build another fuzzer, the way to go is to spend a lot of time adapting different techniques from different fuzzers and so reinventing the wheel. Custom fuzzers written from scratch are usually naive, single core loops with a mutator, while creating the n-th fork of AFL increases the fragmentation. Moreover, all these fuzzers have scalability problems.

1.2 Contributions

In this thesis, we present several contributions to address the previously discussed challenges. The structure of the thesis starts with an introductive chapter on fuzzing background, Chapter 2, then five chapters follow, one for each contribution presented in the subsequent text, and in the end we discuss the conclusion and the future work of the thesis.

Chapter 3

While fuzz testing proved to be a very effective technique to find software bugs, we stated that one of its main limitations is the fact that popular coverage-guided designs are optimized to reach different parts of the program under test, but struggle when reachability alone is insufficient to trigger a vulnerability. In reality, many bugs require a specific program state that involves not only the control flow but also the values of some of the program variables. Unfortunately, alternative exploration strategies that have been proposed in the past to capture the program state are of little

help in practice, as they immediately result in a state explosion.

In this contribution, we propose a new feedback mechanism that augments code coverage by taking into account the usual values and relationships among program variables. For this purpose, we learn *likely invariants* over variables at the basic-block level and partition the program state space accordingly. Our feedback can distinguish when an input violates one or more invariants and reward it, thus refining the program state approximation that code coverage normally offers.

We implemented our technique in a prototype called InvsCov, developed on top of LLVM and AFL++. Our experiments show that our approach can find more, and different, bugs with respect to fuzzers that use a pure code-coverage feedback. Furthermore, they led to the discovery of two vulnerabilities in a library tested daily on OSS-Fuzz, and still present at the time in its latest version.

Chapter 4

To further investigate the field of alternative feedbacks to uncover more bugs, we examined existing program representations looking for a match between expressiveness of the structure and adaptability to the context of fuzz testing. In particular, we believe that data dependency graphs (DDGs) represent a good candidate for this task, as the set of information embedded in this data structure is potentially useful for finding vulnerable constructs by stressing combinations of def-use pairs that would be difficult for a traditional fuzzer to trigger. Since some portions of the dependency graph overlap with the control flow of the program, it is possible to reduce the additional instrumentation to cover only "interesting" data-flow dependencies, those that help the fuzzer to visit the code in a distinct way compared to standard methodologies.

To test these observations, we propose a new approach that rewards the fuzzer not only with code coverage information but also when new edges in the data dependency graph are hit. Our results show that the adoption of data dependency instrumentation in coverage-guided fuzzing is a promising solution that can help to discover bugs that would otherwise remain unexplored by standard coverage approaches. This is demonstrated by the 72 different vulnerabilities that our data-dependency driven approach can identify when executed on 38 target programs from three different datasets.

Chapter 5

AFL is one of the most used and extended fuzzer, adopted by industry and academic researchers alike. While the community agrees on AFL's effectiveness at discovering new vulnerabilities and its outstanding usability, many of its internal design choices remain untested to date. Security practitioners often clone the project "as-is" and use it as a starting point to develop new techniques, usually taking everything under the hood for granted. Instead, we believe that a careful analysis of the different parameters could help modern fuzzers improve their performance and explain how each choice can affect the outcome of security testing, either negatively or positively.

The goal of this paper is to provide a comprehensive understanding of the internal mechanisms of AFL by performing experiments and comparing different metrics used to evaluate fuzzers. This can help to show the effectiveness of some techniques and to clarify which aspects are outdated. To perform our study we performed nine unique experiments that we carried out on the popular Fuzzbench platform. Each test focuses on a different aspect of AFL, ranging from its mutation approach to the feedback encoding scheme and its scheduling methodologies.

Our findings show that each design choice affects different factors of AFL. While some of these are positively correlated with the number of detected bugs or the coverage of the target application, other features are instead related to usability and reliability. Most importantly, we believe that the outcome of our experiments indicates which parts of AFL we should preserve in the design of modern fuzzers.

Chapter 6

The release of AFL marked an important milestone in the area of software security testing, revitalizing fuzzing as a major research topic and spurring a large number of research studies that attempted to improve and evaluate the different aspects of the fuzzing pipeline.

Many of these studies implemented their techniques by forking the AFL codebase. While this choice might seem appropriate at first, combining multiple forks into a single fuzzer requires a high engineering overhead, which hinders progress in the area and prevents fair and objective evaluations of different techniques. The highly fragmented landscape of the fuzzing ecosystem also prevents researchers from combining orthogonal techniques and makes it difficult for end users to adopt new prototype solutions.

To tackle this problem, in this chapter, we propose LibAFL, a framework to build modular and reusable fuzzers. We discuss the different com-

ponents generally used in fuzzing and map them to an extensible framework. LibAFL allows researchers and engineers to extend the core fuzzer pipeline and share their new components for further evaluation. As part of LibAFL, we integrated techniques from more than 20 previous works and conduct extensive experiments to show the benefit of our framework to combine and evaluate different approaches. We hope this can help to shed light on current advancements in fuzzing and provide a solid base for comparative and extensible research in the future.

Chapter 7

In this last contribution, we present a first attempt at rewriting the widely used fuzzer AFL++ as a frontend of LibAFL, our new framework for fuzzers development. This prototype, AFLrustrust as it is written in the Rust programming language, was evaluated in the SBST'23 Fuzzing Competition with great results even though it is just a first attempt with missing components that are still under development.

1.3 List of Publications

During the development of this thesis, several papers were published, but only some of them are contributions to this dissertation. We report all of them, while we underline the ones that are included in this thesis (in chronological order):

- Andrea Fioraldi, Daniele Cono D'Elia, Davide Balzarotti
  "The Use of Likely Invariants as Feedback for Fuzzers"
  in 30th USENIX Security Symposium (USENIX Security 21)

- Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, Davide Balzarotti
  "Registered Report: Dissecting American Fuzzy Lop - A FuzzBench Evaluation"
  in the 1st International Fuzzing Workshop (FUZZING 2022)

- Alessandro Mantovani, Andrea Fioraldi and Davide Balzarotti
  "Fuzzing with Data Dependency Information"
  EuroSP 2022, Genoa, Italy

- Andrea Fioraldi and Dominik Maier and Dongjia Zhang and Davide Balzarotti
  "LibAFL: A Framework to Build Modular and Reusable Fuzzers"

Proceedings of the 29th ACM conference on Computer and communications security (CCS) , Los Angeles, U.S.A.

- Andrea Fioraldi and Alessandro Mantovani and Dominik Maier and Davide Balzarotti
  "Dissecting American Fuzzy Lop – A FuzzBench Evaluation"
  ACM Trans. Softw. Eng. Methodol.

- Andrea Fioraldi and Dominik Maier and Dongjia Zhang and Addison Crump
  "AFLrustrust: A LibAFL-based AFL++ prototype"
  The 16th Intl. Workshop on Search-Based and Fuzz Testing, Fuzzing Competition

- Addison Crump and Andrea Fioraldi and Dominik Maier and Dongjia Zhang
  "LibAFL_libfuzzer: Libfuzzer on Top of LibAFL"
  The 16th Intl. Workshop on Search-Based and Fuzz Testing, Fuzzing Competition

- Addison Crump and Dongjia Zhang and Syeda Mahnur Asif and Dominik Maier and Andrea Fioraldi and Thorsten Holz and Davide Balzarotti
  "CrabSandwich: Fuzzing Rust with Rust (Registered Report)"
  Proceedings of the 2nd International Fuzzing Workshop (FUZZING) 2023

- Pietro Borrello, Andrea Fioraldi, Daniele Cono D'Elia, Davide Balzarotti, Leonardo Querzoni, Cristiano Giuffrida
  "Predictive Context-sensitive Fuzzing"
  In Network and Distributed System Security Symposium, NDSS 24

Chapter 2

Background

In this chapter we introduce several background notions required to understand the contributions of this thesis. We start discussing fuzz testing and its variants, then we present the concept of likely invarints and data dependency graphs.

2.1 Fuzz Testing

Fuzz Testing, or *fuzzing*, is a family of software testing techniques first proposed in the '90s [118]. Recently, fuzzing techniques saw significant improvements in their effectiveness, and contributed to the discovery of many security vulnerabilities [128, 111]. Nonetheless, the key idea behind Fuzz Testing research remained simple: repeatedly execute the program under test by using randomly generated inputs, usually chosen to be either unexpected or invalid. Fuzzing tools monitor a program for failures, such as invalid memory accesses or out-of-memory crashes, and report to the user the inputs that triggered such behaviors.

The most naive embodiment of fuzzing just provides random inputs to the program under test without any knowledge about its characteristics (e.g., input format) or the program execution. This approach, albeit still effective in testing legacy code [117], has obvious limitations. Therefore, many different solutions have been proposed over the past decades to increase the effectiveness in bug finding far beyond naive fuzzing. We can group these techniques according to the following three criteria: 1) the amount of information they require to know from the program, 2) the technique they use to generate new testcases, and 3) the feedback they use to guide the exploration.

According to the first criterion, we can distinguish three main categories of fuzzers:

- *White-box* fuzzers, which build a full picture of the program using program analyses. Concolic executors like SAGE [71] and SYMCC [132] belong to this category, as they collect a model of the program in terms of logic constraints during the execution. The cost of such white-box analyses, however, may often be untenable [128];
- *Black-box* fuzzers, which blindly generate random inputs for testing. They can access knowledge about the input format, but generate inputs regardless of how the program implementation looks like [171] [113];
- *Grey-box* fuzzers, which fall halfway between the two previous categories. They access limited information provided by a lightweight instrumentation applied to the program under test, blending the program analysis and testing stages [128]. An example of such information is the *code coverage* extracted from a testcase by systems like AFL [180] and LIBFUZZER [105].

According to our second criterion, we can distinguish instead fuzzers based on their input generation methodology. The two most commonly used approaches in this respect are *generational* and *mutational* fuzzers. A generational fuzzer creates new testcases from scratch, either randomly or by relying on some form of format specification—like a grammar [88] or a domain specific language [55]. Mutational fuzzers instead derive new testcases from a set of prior testcases by mutation; the mutations can be generic [180], target-specific [160], or driven by a user-supplied [13] [130] or inferred [22] [63] format specification.

Finally, by using our third and last criterion, fuzzers can be divided according to the information they use to drive their exploration, which we call *Feedback*. A popular and very effective technique is coverage-guided fuzzing, which uses code coverage as feedback to drive the testcase generation. Previous studies have shown that coverage-based fuzzers are often one order of magnitude more effective at discovering bugs [49]. As also other forms of feedback are possible, we will refer more in general to this fuzzing design as *Feedback-Driven Fuzz Testing*.

2.2 Feedback-Driven Fuzz Testing

In short, when a CGF solution generates a testcase that triggers a previously unexplored portion of the program, it deems the testcase as *interesting*

and adds it to a *queue* of inputs (dubbed *seeds*) maintained for further processing. By combining this technique with a mutational approach, we obtain an evolutionary algorithm driven by code exploration.

Code coverage can be measured in different ways, for instance by considering basic blocks alone or by including entire calling contexts [162]. By far, the most popular criterion used for coverage-guided fuzzers is *edge coverage*, which maximizes the number of edges visited in the control flow graph (CFG) of program functions. Fuzzers like AFL [180] extend pure edge coverage by also including a hit count for edges (i.e., how many times a testcase exercises them) to better approximate the program state. Recently, Ankou developed this idea further by adding coverage-equivalent testcases to the queue depending on the results of an online principal component analysis for hit count differences between executions.

As we anticipated before, other metrics are possible for driving fuzzer evolution. FuzzFactory [126] recently studied several alternatives, such as the fact that the size of memory allocations can be a useful feedback to expose out-of-memory bugs, while the number of identical bits in the operands of a comparison instruction [103] can help in circumventing fuzzing roadblocks. In short, all these feedback techniques act as shortcuts to domain-specific testing goals for which code coverage is not an adequate description.

A more general approach would be to consider, alongside control flow decisions, also data flow information regarding the program state. The most naive embodiment of this feedback—and to the best of our knowledge also the sole to date—is the 'memory' feedback, where every newly observed data values from memory load and store operations are considered as novelty factor for the fuzzer. Unfortunately, this solution easily leads to state explosion [162].

2.2.1 Increasing code coverage

One of the main objectives for many proposed optimizations to coverage-guided fuzzing is to reach a higher amount of covered program points in the same time window compared to previous solutions. This metric comes from the observation that a fuzzer cannot uncover a fault in an unxeplored portion of the code.

During the last years, the community identified some artifacts in the code that prevent a fuzzer to explore the code behind these so-called "roadblocks". The main types of roadblocks are:

Multi-byte comparisons. The probability that a generic byte-level mutator guess from scratch the value needed to bypass a certain multi-byte comparison is near to 0. Several approaches try to overcome this problem, like LAF-Intel [5] that splits them into several single-byte comparisons at the compiler level, Vuzzer [139] and Angora [35] that identify portions of the input related to the values in the comparisons using taint analysis, or RedQueen [16] that heuristically replaces correspondences between input and comparisons.

Checksum checks. Checksums checks are a particularly hard version of comparison. While they can be solved by providing valid inputs, the problem is that any generic mutation invalidates it generating an invalid input, making the fuzzer unable to explore the code behind these checks. The most common way to handle these checks is to patch them out and restore them when the fuzzer finds a crash, manually or automatically, as proposed in several works [129, 16, 63].

Hashtable lookups. The third important code pattern identified as a problem for coverage-guided fuzzers is the hashtable lookups as the information needed to get the right item is not explicit in code coverage and eventual comparisons are between encoded versions of portions of the input. Current automatic solutions [16, 65] are quite naive because they can solve the roadblock if the lookup is in a specific helper function looking at its pointer arguments. While using a helper function is a common coding pattern for hashtable lookups, this is not an exhaustive solution and the state-of-the-art [14] requires manual annotations.

In addition, a popular technique to handle roadblocks is hybrid fuzzing, in which a concolic executor is used to aid the fuzzer [174, 132, 29]. For the checksum checks, the concolic engine can be used to repair the checksums on crashing testcases. For the other two kinds of roadblocks, in theory, it should be possible to solve them with this technique but concolic engines struggle to solve hashmap-like lookups due to the complexity of handling symbolic pointers.

2.2.2 Meaningful inputs generation

A problem that fuzzers face is the inability to stress code paths behind the parsing stage. While generic mutators are very good in stressing parsers with invalid inputs, in order to fuzz deep in the program we need mutators able to go beyond the parser.

A widely adopted solution is to guide the input generation with a model of the input format, that can be a grammar [13, 155] or a block-based model [130], using an internal representation, like the AST, that is easy to mutate while preserving validity.

An important field is the research into the automation of this process, as writing an input model is still a human task. So in the past year, some solutions were proposed to infer how the input bytes are handled by the program with the goal of mutating them accordingly [22, 63].

Another important property of inputs that some domain-based fuzzers may want to preserve is semantic validity. For instance, when fuzzing a compiler, a testcase that uses undefined variables is syntactically valid, but the compilation will fail. Fuzzers such as FUZZILLI [75] or CODEALCHEMIST [77] implement semantic-preserved mutations to fuzz JavaScript interpreters behind the compilation stage.

2.2.3 Hunting non-crashing faults

While the algorithmic improvements to fuzz testing play an important part in the game, the discrimination between testcases that trigger or not a fault is another important room for fuzzers' enhancement.

It is easy to spot bugs that cause a crash in the application just by observing the exit status for instance on UNIX systems, but many bugs are silent and do not corrupt the application state enough to cause a crash [119].

An example of this kind of bugs is many logic bugs related to integer arithmetic. While in some application the invalid state may be propagated further in the code and cause a crash, in many others this kind of faults remain silent.

2.3 Program Properties and Invariants

Property-based testing is a software testing methodology in which some form of specification of the program's properties drives the testing process. Such specification simultaneously defines what behaviors are valid and serves as basis for generating testcases [61].

The correctness oracle can be embedded in the target program itself in the form of a set of assertions that check the validity of each *invariant*, i.e., a property that according to the specification must always hold at that program point [58]. Testcases can then be generated by aiming at violating the invariant assertions.

QuickCheck [39] is probably the most well-known among such systems. Recently works such as Zest [125] and Hypothesis [108] borrows fuzzing concepts like feedback-driven mutations to improve their efficiency when testing, respectively, Java and Python codebases.

Since delegating the identification of program properties to the developers can be a daunting prospect, automation has been the subject of a large body of previous works in the field. Automated invariant learning is also a widely explored topic in other areas, for instance for memory error detection [140].

Invariants can be discovered by conducting static code analysis: for instance, rcore [70] builds on abstract interpretation [41] and monitors invariants at run-time to detect program state corruption from memory errors. Generally, such invariants are sound and incur limited false positives, yet the inherent over-approximation of static analysis may generate invariants too coarse to discriminate program states in an effective manner for high-level analysis.

Therefore, a more precise way to discover invariants, which also produces them in greater quantity, consists of inspecting the program state at run-time. For this reason, approaches like [78], [56], and [127] build on information gathered during the execution, in a dynamic fashion. The downside of dynamic approaches is that, unlike static ones, they produce *likely invariants*, i.e., invariants that hold for the analyzed traces but may not hold for all inputs. Hence, they may result in false positives when the learned invariants capture only local properties of the observed executions.

In Chapter 3 we build upon this well-known coverage problem [58] and turn it into an advantage for driving a fuzzer. We do that by starting from a corpus of testcases that—as it is the case with real applications—cannot be representative of all program states, then we modify a fuzzer to make it more sensitive to behaviors that diverge from the likely invariants obtained from the initial corpus. In this case, the fact that the learned invariants capture properties of the observed executions instead of properties of the program itself is the key intuition we use to generate a more diverse set of input values.

Invariants historically play a key role in many development tasks such as software testing, optimization, and maintenance [58]. In the context of security research, several works have explored invariants for other problems as well.

In the context of anomaly detection, invariants can act as oracles for program hardening. Works such as [70] and [153] instrument programs to block memory corruption exploits in production, as run-time checking costs

turn out to be modest. Web applications can benefit from similar protection as well, as explored in [42] with DAIKON and PHP code.

Fault localization is another popular twist. Whenever multiple invariants turn out to be violated, a typical workflow to locate the root cause is to study similar inputs to filter out non-relevant invariants. An example is Sahoo et al. [143], which uses dynamic backward slicing to remove more invariants, and AURORA [23], which employs a statistical analysis of the learned predicates.

2.4 Data Dependency Graphs

Data Dependency Graphs (DDGs) were first introduced by Ferrante et al. [60] in 1987 as a program representation to capture the data-flow relationship among each instruction in the program. More formally, the LLVM documentation defines a data dependence graph as a structure that "*represents data dependencies between individual instructions. Each node in such a graph represents a single instruction and is referred to as an 'atomic' node [...]*" [9] while edges are defined as "*def-use dependencies between the atomic nodes*".

The introduction of DDGs paved the way to new program analysis techniques, such as program slicing [167, 34] (i.e., the set of statements that affect the value of a certain variable) and reaching definitions [11] (i.e., the set of definitions that could hit a certain point in the code). Over the years, researchers have proposed DDG-based techniques to build new approaches in compiler optimizations, as reported by the seminal work in this field by Kuck et al. [94]. For instance, Heffernan et al. [81] used the data dependencies that exist inside a program to improve instruction reordering and increase the CPU pipeline performances. Other works [97, 82] proposed advanced scheduling approaches based on the adoption and transformation of the DDG to measure the dependencies among instructions and evaluate when to perform a reordering operation. Another common application of DDGs is dead code elimination, which aims at identifying which assignments in the code can be removed after checking that no subsequent operations depend on them [93, 24, 31].

In software security, DDGs are often used to verify if potentially unsafe or poorly sanitized data (the *source*) can propagate information inside the program until it reaches a certain statement that can trigger a vulnerability (the *sink*). This led to a set of applications of the DDG, especially in static software testing. For instance, in 1994 Kinloch et al. [91] suggested that the combination of the DDG with the Control Flow Graph (CFG) could

help programmers at detecting bugs. More recently, Yamaguchi et al. [169] proposed a program representation, known as the Code Property Graph, that combines the CFG, the DDG, and the Abstract Syntax Tree (AST) of a program. The authors then designed specific queries over this data structure to detect vulnerable patterns in the code. The popularity of DDGs for vulnerability discovery is confirmed by the comparison performed by Zhioua et al. [183] among static software testing tools. The authors found that 3 out of the 4 investigated frameworks implemented a data dependency analysis component when scanning C source code. However, data dependencies are not just used to analyze source code but play a very important role also in other scenarios. For instance, Cheng et al [37] use them to perform taint analysis on IOT firmware images with the goal of finding vulnerable flows, and several model checking techniques rely on them to detect unsafe program points [114, 165].

Possible applications of the Data Dependency Graph are not limited to unsafe languages such as C/C++. In 2009, Hammer et al. [76] proposed an approach based on path conditions in dependency graphs that can be used to reveal security-sensitive flows inside Java code. Moreover, in 2015 Qian et al. [135] developed a static analysis framework to detect vulnerabilities in Android applications by traversing the DDG, similarly to what already done in [169] for C source code.

Chapter 3

Introducing Likely Invariants as Feedback

Thanks to its success in discovering software bugs, *Fuzz Testing* (or *fuzzing*) has rapidly become one of the most popular forms of security testing. While its original goal was simply to randomly generate unexpected or invalid inputs, today's fuzzers always rely on some form of heuristics to *guide* their exploration. The most popular of these strategies is, by far, *Coverage-Guided Fuzzing* (CGF), in which the fuzzer selects inputs that try to increase some coverage metric computed over program code—typically, the number of unique edges in the control flow graph. Consequently, a large body of research has focused on overcoming the limitations of coverage-guided fuzzers, for instance by proposing techniques to solve complex path constraints [174] [132] [156] [138] [16], by reducing the large number of invalid testcases generated by random mutations [130] [125] [13] [22] [63], or by focusing the exploration on more 'promising' parts of the program [123] [120] [26].

While these improvements have considerably decreased the time required to visit different parts of the target application, it is important to understand that code coverage alone is a necessary but not sufficient condition to discover bugs. In fact, a bug is triggered only when i.) program execution reaches a given instruction, and ii.) the state of the application satisfies certain conditions. In rare cases, there are no conditions on the state, as it is the case for most of the bugs in the LAVA-M [51] dataset—which were artificially created to be triggered by simply reaching a certain point in the target applications [80].

On the one hand, this aspect is very important because the use of code coverage to reward the exploration results in the fact that fuzzers do not have any incentives to explore more states for an already observed set of

control-flow facts (e.g., branches and their frequencies). Thus, it is considerably harder for existing tools to detect bugs that involve complex constraints over the program state. On the other hand, the simple solution of rewarding fuzzers for exploring new states (state coverage) is also a poor strategy, which often *decreases* the bug detection rate. This is due to the fact that, for non-trivial applications, the number of possible program states is often infinite.

Therefore, special techniques are needed to reduce the program state into something more manageable to explore during testing, while still preserving the fuzzer's ability to trigger potential bugs. To date, few works have tried to find such compromise. For instance, some fuzzers approximate the program state by using more sensitive feedbacks, like code coverage enriched with call stack information, or even with values loaded and stored from memory. This second approach, as shown by Wang et al. [162], better approximates the program state coverage by taking into account not only the control flow but also the values in the program state, but is less efficient than others in finding bugs as it incurs into the state explosion problem mentioned above.

To capture richer state information while avoiding the state explosion problem, researchers have also looked at human-assisted solutions. For instance, FuzzFactory [126] lets the developers define their domain-specific objectives and then adds waypoints that reward a fuzzer when a generated testcase makes progress towards those objectives (e.g., when more bits are identical among two comparison operands).

At the time of writing, the most successful approximation of the program state coverage is achieved by targeting only certain program points selected by a human expert, as recently proposed in [15]. In the work, portions of the state space are manually annotated and the feedback function is modified to explore such space more thoroughly. We believe that the automation of this process may be a crucial topic in future research in this field.

Our Approach. In this first chapter, we propose a new feedback for Fuzz Testing that takes into account, alongside code coverage, also some interesting portions of the program states in a fully automated manner and without incurring state explosion.

The key idea is to augment edge coverage—the most widely-adopted and successful code coverage metric used by fuzzers—with information about local divergences from 'usual' variable values. To this end, we mine *likely invariants* on program variables by executing an input corpus (such as the queue extracted from a previous CGF campaign) and learning constraints on the values and relationships of those variables over all the observed ex-

ecutions. It is important to note that execution-based invariant mining produces constraints that do not necessarily model properties of the program, but rather local characteristics of the analyzed input corpus [57]: hence, constraints may be violated under different inputs.

Our intuition is that these local properties represent an interesting abstraction of the program state. We thus define a new feedback function that treats an edge differently when the incoming basic block sees one or more variable values that violate a likely invariant. This approach increases the sensitivity of a standard CGF system, rewarding the exploration of program states that code coverage alone would not be able to distinguish.

We develop a set of heuristics to produce and refine invariants, and techniques to effectively instrument programs with a low-performance overhead—a very important metric in fuzzing. We implement them into a prototype called InvsCov on top of LLVM [95] and the AFL++ [65] fuzzer.

Our experiments, conducted over a set of programs frequently tested by other fuzzers, suggest that our feedback, by succinctly taking into account information about usual program state in addition to control flows, can uncover both more and different bugs than classic CGF approaches.

Contributions. In summary, the main contributions of this chapter are:

- A new feedback that combines control flows with an abstraction of the program state from mined invariants;
- A prototype implementation of our approach based on LLVM and AFL++ called InvsCov;
- An evaluation of the effectiveness of our approach against classic and context-sensitive edge coverage.

We share the InvsCov prototype as Free and Open Source Software at

https://github.com/eurecom-s3/invscov

3.1 Methodology

```
1 int wavlike_msadpcm_init (SF_PRIVATE *psf, int blockalign, int
      samplesperblock)
2 { MSADPCM_PRIVATE *pms ;
3   unsigned int pmssize ;
4   // Likely Invariants:
5   //  - blockalign ∈ { 0, 2, 256 }
6   //  - blockalign < samplesperblock
```

```
7   ...
8   pmssize = sizeof (MSADPCM_PRIVATE) + blockalign + 3 *
      psf->sf.channels * samplesperblock ;
9   ...
10  pms->samples  = pms->dummydata ; // array in pms
11  pms->block    = (unsigned char*) (pms->dummydata +
      psf->sf.channels * samplesperblock) ;
12  pms->channels  = psf->sf.channels ;
13  pms->blocksize  = blockalign ;
14  ...
15 }
```

Listing 3.1: Excerpt of wavlike_msadpcm_init() initialization code.

```
1  static int msadpcm_decode_block (SF_PRIVATE *psf,
       MSADPCM_PRIVATE *pms)
2  {
3   ...
4   sampleindx = 2 * pms->channels ;
5   // Likely Invariants:
6   //   - pms->blocksize == 256
7   while (blockindx < pms->blocksize)
8   {   bytecode = pms->block [blockindx++] ;
9       pms->samples [sampleindx++] = (bytecode >> 4) & 0x0F ; //
        heap overflow bug
10      pms->samples [sampleindx++] = bytecode & 0x0F ;
11      } ;
12  ...
13 }
```

Listing 3.2: Vulnerable code found in msadpcm_decode_block().

In this section, we present the intuition behind our approach by using an example of a real-world vulnerability we discovered during our experiments. The vulnerability is a heap overflow in the WAV file format parsing of libsndfile, a popular library to operate on audio files. Listings 3.1 and 3.2 show the affected code. Specifically, the vulnerability is located in the msadpcm_decode_block function of file ms_adpcm.c, reported here at line 9 in Listing 3.2.

For our purpose, it is interesting to note that all the coverage-guided fuzzers we used in our experiments (§6.4) were able to reach the vulnerable

point in the code without, however, triggering the bug. Despite the fact that the vulnerable code is 'easy-to-reach' and that `libsndfile` is often used in fuzzing experiments (including the Google OSS-Fuzz project and recent research works such as [69] and [172]), the bug was still present when we ran our experiments.

This is likely due to the fact that to trigger the bug the loop should write outside the memory pointed by `pms->samples`, which references the C99 variable-size array field at the end of the `pms` structure. This only happens when the program is in a specific state, characterized by a small allocation size for the `pms` buffer (line 8 in Listing 3.1) and a `pms->blocksize` value (line 13 in Listing 3.1) sufficiently high to force the loop to write out of the bounds of the array.

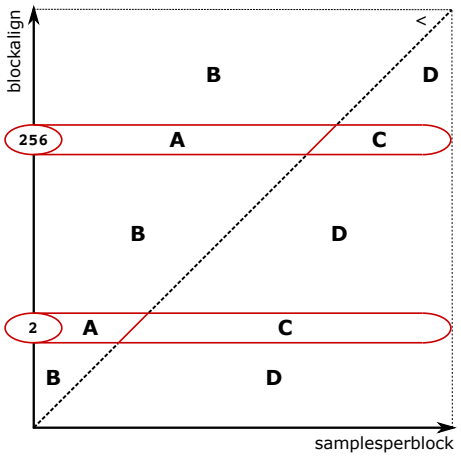
However, none of these requirements can be extracted from code coverage, as there are no branches in the program that involve these thresholds. Instead, they both depend on two input-derived values: `blockalign` and `samplesperblock`. Hence, a CGF-based exploration may easily satisfy one of the requirements but, without recognizing this as progress in the program exploration, it would unlikely satisfy both at the same time. In fact, any generated testcase satisfying either requirement would exercise an "intermediate" program state closer to the bug, but would not be seen as an interesting one to add to the queue for more mutations, because in the eyes of CGF it does not bring novel code coverage.

This example shows the challenge that modern fuzzers encounter when exploring the state of a program, even for code that does not entail difficult path conditions to be reached. State-of-the-art CGF systems can saturate in coverage while still missing bugs at program points touched in their operation. Also, they may fail to generate testcases to cover unseen program points whenever those are reachable only upon meeting conditions that do not depend on control flow alone.

3.1.1 Program State Partitions

The core idea of this chapter is that we can divide the program space in different partitions at multiple points in the application code, by learning likely invariants from executing the program under test over an initial corpus of inputs.

To continue with our example, let us imagine that we can fuzz `libsndfile` for a certain amount of time, e.g., 24h, with a standard CGF system (we will discuss in §3.1.4 the effect of different corpora on the extracted invariants). By investigating the values of the variables across all seeds saved by the



Invariant	Condition
LI_1	blockalign $\in \{0, 2, 256\}$
LI_2	blockalign $<$ samplesperblock

Invariant	A	B	C	D
LI_1	✓	✗	✓	✗
LI_2	✗	✗	✓	✓

Figure 3.1: State partitioning for wavlike_msadpcm_init() induced by the two likely invariants LI_1, LI_2. The bug can be exercised only when in partition B (LI_1, LI_2 both violated).

fuzzer, we would identify two likely invariants for the init function and one for the vulnerable decoding loop. All invariants are included as comments in Listings 3.1 and 3.2.

It is important to understand that these invariants are descriptive of the limited number of states that were induced by the corpus generated by the fuzzer. In other words, each invariant expresses a condition over the state of the program that the fuzzer was unable to violate during the testing experiment. Therefore, our intuition is that we can use these invariants to divide the program state into a number of partitions, as depicted in Figure 3.1 for the init function.

In this case, we can see that the two invariants partition the space in four non-contiguous areas (A to D in the figure), all but the first *unvisited* by the fuzzer. This information allows us to provide feedback to the fuzzer to explore new abstract states without incurring into the classic state explosion problem.

Moreover, since these states can be reached only by violating the invariants we learned over previous executions of the fuzzer, our intuition is that they are likely to bring the program into seldom-explored corner cases—where vulnerabilities may lie undetected for a long time.

To capture this information, the approach presented in this chapter augments the classic edge coverage feedback by using the violation of likely invariants learned over basic blocks. In an ideal world, we could learn exact invariants and transform them in terms of code coverage, allowing pure

coverage-based fuzzers to receive feedback to progress towards these areas. However, as described in §2.3, current invariant mining techniques lead to both over or under approximations.

3.1.2 Using Invariants as Feedback

The common limitation of dynamic invariant detection is that the resulting invariants often capture local properties of the test suite more than static properties of the program.

However, for our purpose, this is exactly what we want. In fact, likely invariants that represent only local properties of the corpus are interesting because their violation would tip fuzzers about what value combinations in the program state are unusual, and ideally the home of bugs.

Therefore, we define our invariant-based feedback as a combination of edge coverage with the information about which likely invariants are violated in the source basic block. To inform the fuzzer about the progress towards interesting states, we then tweak the classic novelty search algorithm adopted by most coverage-based systems. In particular, for each CGF-instrumented control flow graph edge, we make it generate a different value for the novelty search for each unique combination of violated invariants. As we will detail in Section 3.2.2, we track invariants individually and reward them independently at each basic block: this choice brings an unambiguous, implicit encoding of program state partitions.

The invariants ability to partition the program state space without incurring state explosion is also one of the key insights of our approach. At each basic block N invariants can partition the state locally just like N non-parallel lines can divide a plane into $N * (N + 1)/2 + 1$ regions. In practice, since each basic block typically manipulates only few variables, N is usually a very low value (statistics in Appendix 3.5).

Back to our example, for the `wavlike__msadpcm__init` function we have two variables involved in the learned invariants: `blockalign` and `samplesperblock`. The partition that triggers the vulnerability is B—the one that sees both invariants violated. Our fuzzer found the bug for a value assignment `{blockalign = 1280, samplesperblock = 8}`.

With the enriched sensitivity from our invariant-based feedback, the fuzzer can violate each invariant separately, save such testcases for partitions A and D, and for instance splice the two testcases to generate one that brings the state to B. More in general, our approach can generate inputs that violate multiple invariants by either combining or mutating previous seeds—each violating one or more distinct invariants.

As for the likely invariant involving `pms->blocksize` in the buggy function (Listing 3.2), we observe that violating it is not a sufficient condition to trigger the bug. The field is assigned equal to `blockalign` in Listing 3.1, but also `samplesperblock` has to contribute to expose the bug.

3.1.3 Pruning the Generated Checks

With our example, we showed how we can use invariants to partition the program state and how we can then provide this information as feedback to drive the fuzzer's exploration.

However, not all invariants are equally useful: while having more invariants does not affect our methodology (i.e., we do not lose sensitivity by exposing more partitions), the extra states they generate can pollute our feedback and the additional instrumentation can impact the run-time overhead.

Therefore, we designed three classes of pruning rules to remove invariants that would be fruitless to check either in light of other available information or because of the nature of their constituents.

1. The first class of invariants we discard are those that are **impossible** to violate. For instance, our likely-invariant mining system would often learn that unsigned integer variables are always greater than or equal to zero—which is not a very useful condition to drive a fuzzer. To identify these and alike cases, we perform a *Value Range Analysis* [79] for each function of the program under test. Arguments and global storage are initially seen unconstrained, and the analysis produces bounds for function variables that hold for any execution. Using range information, we instruct our miner to never generate likely invariants that are logically weaker than the ones found statically. Since these invariants cannot be violated, we can save the instrumentation cost required to monitor them.

2. The second class of fruitless invariants are those that combine **unrelated variables**. To remove these relationships, we compute *Comparability Sets* for each function of the program under test: each variable belongs to only one such set, and invariants combining variables across different sets are discarded. We initially create a separate set for each variable, then use a unification-based policy by iterating over function instructions and merging the sets of two variables whenever those occur as operands for the same statement. Eventually, a comparability set contains variables that take part in related computations. Few

exceptions apply: for instance, in an array pointer computation we do not merge the sets of the base and the index elements as they are not directly related.

3. Whenever different invariants have **overlapping conditions**, it is possible to optimize their run-time verifications by reusing previously computed values. In particular, we target pairs of likely invariants that share the same conditions on some of their variables. If the two invariants concern two program points p and p' where p' can execute only after p, we can use a standard flow-sensitive analysis to determine whether between p and p' there are no intervening re-definitions for any of the involved variable. In that case, we simply propagate the value computed at p and save the computation cost at p'.

The output of the value-range analysis and the comparability sets are computed beforehand and passed to the invariant miner, which takes them into account when generating the invariants. Overlapping conditions are instead dealt with when producing the program—augmented with code for checking invariants—that will undergo the testing process.

3.1.4 Corpus Selection

For our entire solution to work, we need to be able to learn likely invariants from a large number of executions of the program under test. Therefore, like for many other evolutionary fuzzing techniques, the choice of the initial corpus of inputs is critical.

An unwise choice can generate invariants that do not describe with sufficient generality the shape of the variables in the program state. For instance, it is a common practice in fuzzing to download many files of a given file format when testing a parser, but almost all those files are valid files. If we learn likely invariants from the program executions of such a corpus, we will bias our invariants on the validity of the file format and, in some cases, this can be a mistake because we might miss interesting partitions of the program state related to invalid inputs.

As we want to address the problem of finding bugs even when the fuzzer saturates in coverage [74], a natural choice is to use as corpus the queue of a coverage-guided fuzzer taken as soon as that fuzzer shows signs of slowing down in reaching new coverage points. A violation of an invariant learned over such corpus will lead to novel feedback for the fuzzer and desaturate the search.

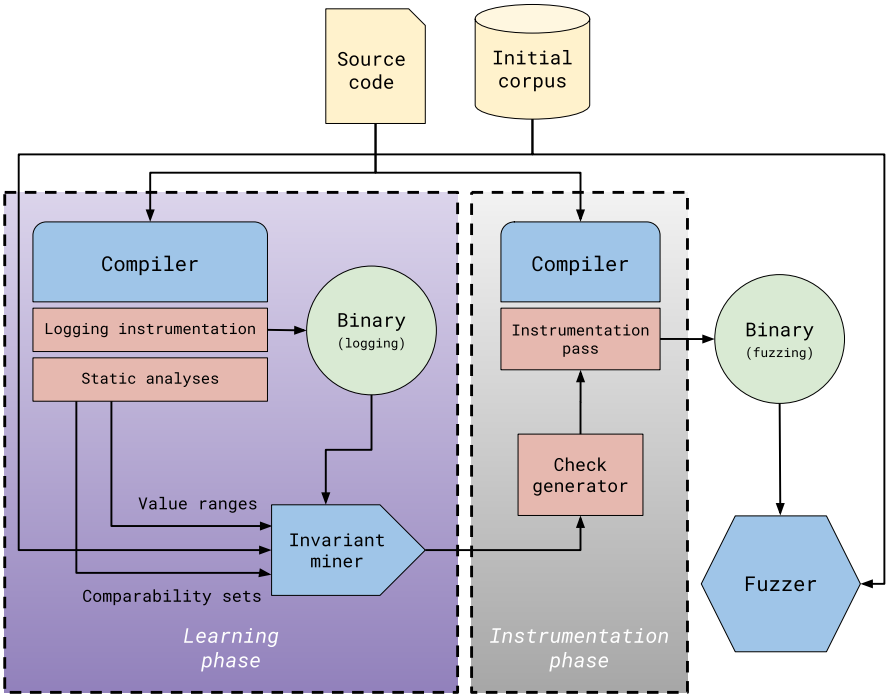


Figure 3.2: High-level workflow of invariant-based fuzzing.

To confirm our intuition we downloaded a dataset of valid files for the programs we tested in §6.4 and mined likely invariants by using such test-cases. We then compared the invariants extracted from these initial seeds with those obtained using the queue after a 24h run of a coverage-based fuzzer initially supplied with the same seeds. In our experiments, we observed that the invariants extracted only by using the valid files led to the discovery of 20% fewer unique bugs than with the invariants extracted from an initial run of a fuzzer.

3.2 Implementation

In the previous section, we introduced the motivation and the key ideas behind our approach. However, we intentionally avoided discussing two important aspects of our solution: i.) how we define the *state* we want to capture in our invariants, and ii.) how we perform the instrumentation of the program under test to collect the information required by our technique.

Our approach can be implemented in different ways, for instance by instrumenting the target source code, or by performing binary-level instrumentation via static rewriting [50] or dynamic translation [47]. While each approach has its own pros and cons, for our experiments we opted for a compiler-based implementation of our invariant-based fuzzing using LLVM [95] and the Daikon [57] likely-invariants system.

Our prototype is written in C++ and re-uses the fast intra-procedural integer range analysis of Pereira et al. [136] for LLVM, which takes an asymptotically linear time to complete. Figure 3.2 provides a high-level view of the complete architecture. We implemented two custom compile-time transformation phases (consisting of roughly 5 KLOC) for LLVM:

1. *Learning phase*, where we emit logging instrumentation for program state variables to feed the invariant miner;

2. *Instrumentation phase*, where we augment the code of the program under test to evaluate the likely invariants in a form directly suitable for coverage-guided fuzzers.

In short, during the *Learning phase* we record all the information about the program state required for invariant mining. We achieve this by running an augmented version of the program under test over a corpus of inputs, which can be obtained in several ways (§3.1.4; in the experiments described in §6.4 we use the seeds generated from a 24h coverage-guided fuzzing session). For invariant mining we use the Daikon dynamic invariant detector, one of the most used dynamic miners: first presented in 2007, Daikon is still under active development.

At each instrumentation place, invariant mining faces a cubic time complexity in the number of constituents (i.e., program variables) [58]. However, since our technique is applied at the level of basic blocks, the number of variables is practically a small constant, and the total computation cost for invariant mining becomes linear in the number of basic blocks in the program.

During the *Instrumentation phase*, we then encode likely-invariant information in program functions to expose them to coverage-guided fuzzers. Our transformed programs can execute out of the box on any AFL-based fuzzer but, as we elaborate in more details in §3.2.2, we foresee minimal adaptations to support coverage tracking schemes from other fuzzer families.

3.2.1 State Invariants Learning

In order to learn the likely invariants, we need to observe the values of the program state during the execution of the program over the initial corpus of inputs. To achieve that, we compile a dedicated version of the program under test that includes additional instrumentation to collect such values at run-time.

Since our prototype is implemented on top of the *Intermediate Representation* (IR) of LLVM, we can easily expose the state of the program at the level of each basic block. Also, the IR allows us to avoid issues with uninitialized values that affect tracing complex data types at the source code level [4]. For instance, a structure may contain a pointer, and to extract the present pointed value for tracing purposes its address must be valid. The original Kvasir front-end of DAIKON uses expensive dynamic binary instrumentation [47] to read variables and inspect memory. However, by working at the IR level, we can just wait until the address appears in a virtual register as the result of a load operation and use it for tracing.

Another advantage of using an Intermediate Representation is that, in an IR, instructions are typically expressed in a Single Static Assignment (SSA) form [142]. SSA entails that each variable can only be assigned once, and each use must be reached by a (unique) prior definition.

For simplicity, in our implementation we ignore floating-point instructions and model the program state by looking at SSA variables holding integer values. For local variables, since multiple SSA variables exist in the IR for a single source-level variable[1], we restrict our analysis to those SSA variables that can be directly connected to a source-level variable, by using debug metadata from the LLVM front-end.

When a program instead accesses non-local storage or a field of a non-primitive type, LLVM introduces an SSA variable as result of a load operation for the current contents. By instrumenting such IR variables, our invariant mining extends also to global variables, heap storage, and fields of structs.

Moreover, since our goal is not just to model the state of an application, but to improve the effectiveness of a security-oriented testing technique, we focus our analysis on those variables that can have security-related consequences, according to the following three rules:

- The variable is part of a `GetElementPtr` instruction[2] for pointer computation unless only constant indexes are involved;
- The variable value is loaded from or stored to memory by using a `Load` or `Store` instruction;

- The variable represents the return value of a function.

To collect the value of each variable we implemented an LLVM function pass that, alongside instrumenting the variables of interest with logging machinery, also dumps at compilation time the Comparability sets and the integer ranges [136] to support the pruning techniques described in §3.1.3.

The pass creates a JSON file for each code module to store information about program points and variables (type, comparability, and bounds). We then process and merge these intermediate files from all modules to produce the Daikon declaration file[3], adding also comparability and range bound information for the sake of invariant pruning (Section 3.1.3). We instruct Daikon to run the instrumented program over each input in the corpus and retrieve the values logged for its variables. We mine our invariants by using the on-demand mode of Daikon, which learns incrementally from each execution.

3.2.2 Program Instrumentation

In the second phase of our approach, we embed the likely invariants obtained from the Learning phase in the program under test and add the required AFL instrumentation to drive the fuzzer. For this, we turn each invariant into a C function that we compile to LLVM IR and invoke from the program point of interest. The function takes as arguments the IR values that are part of the invariant and evaluates them, returning a unique identifier when the invariant is violated, and zero otherwise. Listing 3.3 provides an example of such functions, generated for an invariant with identifier `123` that checks whether `varo > 1`.

```
unsigned ___daikon_constr_123(int varo) {
  if (!(varo > 1))
    return 123 << 1;
  return 0;
}
```

Listing 3.3: Example of generated C code from an invariant.

```
// Original AFL edge-coverage code
___afl_area_ptr[cur_loc ^ prev_loc]++;
```

[1]Special ϕ-*functions* regulate the currently visible assignment when it depends on the CFG basic blocks the program traversed.

[2]https://llvm.org/doxygen/classllvm_1_1GetElementPtrInst.html

[3]https://plse.cs.washington.edu/daikon/download/doc/developer/File-formats.html#Declarations

```
prev_loc = cur_loc >> 1;

// Extended to capture violations of invariants
___afl_area_ptr[cur_loc ^ prev_loc]++;
prev_loc = cur_loc >> 1;
prev_loc ^= ___daikon_constr_123(var0);
prev_loc ^= ___daikon_constr_321(var2, var3);
```

Listing 3.4: Classic and Extended AFL instrumentation for edge coverage.

To expose the violation of invariants as if there were a code coverage change, we modify few lines that are part of the classic AFL instrumentation, as depicted in Listing 3.4. In the original code, `cur_loc` represents the identifier assigned to the current block, and `prev_loc` is the right-shifted-by-one value of the previous block identifier. An edge coverage event is reported by XOR-ing these two variables and by incrementing the corresponding entry in the `___afl_area_ptr` coverage map. In this way, the code can also capture the number of times that the edge is executed modulo 256 (map values are 8-bit unsigned integers).

To include the information about the violated invariants into the AFL feedback, we encode the identifiers of the violated invariants into `prev_loc` by using the XOR operation. This allows each edge to also capture which invariants were violated in the source basic block. Listing 3.4 shows how we augment edge coverage with the combination of the outcome of the functions that check the invariants with identifiers 123 and 321. Note that zero is the identity element for XOR, so edge coverage is unaffected when an invariant is not violated (i.e., the invariant's function returns zero).

We insert our instrumentation by using an LLVM Function pass. During this phase, we also apply the optimization to remove overlapping conditions, as described in §3.1.3, by identifying those invariant evaluations in different blocks that perform the same checks on the same values. To minimize their number, and therefore avoid redundant instrumentation that could slow down the execution, we build the dominator tree [134] for each function of the target program and emit the check only at the top-level block in such tree that strictly dominates all the other blocks in which the same invariant appears. Thanks to the SSA form, the value returned for the check is guaranteed to be visible at its dominated blocks, and therefore we can avoid re-executing the evaluation function.

3.3 Evaluation

In our experiments we tackle the following research questions:

- **RQ1.** Are our invariant pruning heuristics effective in reducing the number of generated checks?
- **RQ2.** Does our new feedback incur state explosion?
- **RQ3.** Can our feedback lead a fuzzer to effectively exploring more program states than code coverage?
- **RQ4.** Can our feedback uncover more, or just different, bugs than code coverage?
- **RQ5.** What run-time overhead does our feedback introduce?

Target Programs

Program	Package	KLOC	Sanitizers
`catppt`	CATDOC 0.95	7	ASan, UBSan
`xls2csv`	CATDOC 0.95	7	ASan, UBSan
`jasper`	Jasper 2.0.16	176	ASan
`sndfile-info`	libsndfile 1.0.28	79	ASan, UBSan
`pcre2` (harness)	PCRE2 10.00	68	ASan, UBSan
`gm`	GraphicsMagick 1.3.31	251	ASan, UBSan
`exiv2`	Exiv2 0.27.1	80	ASan, UBSan
`bison`	Bison 3.3	100	ASan

Table 3.1: List of target programs used for the evaluation along with the corresponding package, the lines of C/C++ code, and the sanitizers used when compiling each program.

In order to answer these questions, we selected 8 real-world target programs as subjects for our experiments. We opted for programs that work on distinct file types and follow different strategies in the implementation of the parsing stage. In more detail, `cappt` and `xls2csv` look up tokens using large switch constructs, `jasper` works on a chunk-based format, `sndfile-info` is stream-oriented, `pcre2` uses lookup tables, `gm` combines different strategies, `exiv2` is chunk-based and uses C++ objects to represent chunks, and `bison` is an LR parser. The versions we selected are known to contain bugs as they are widely used in past works (e.g., [69] [112] [138]) to test fuzzers. For a rigorous evaluation we also manually de-duplicate crashes when assessing bug finding capabilities [92].

Program	Command line
catppt	@@
xls2csv	@@
jasper	-f @@ -t jp2 -T mif -F /dev/null
sndfile-info	–cart –instrument –broadcast @@
pcre2 (harness)	
gm	convert @@ /dev/null
exiv2	@@
bison	@@

Table 3.2: List of command line used for the each target program when run in the fuzzer.

Note that popular benchmarks like LAVA-M [51] are not suitable for evaluating our approach, as the bugs they contain depend exclusively on code reachability guarded by magic-value comparisons [80]. We also opted not to use the recent and appealing MAGMA [80] benchmarks, as their hardwired logging primitives (used to check for ground truth) split basic blocks and thus conflict with the granularity of our invariant construction and instrumentation.

To enable reproduction of our results, Table 3.1 lists the programs we used in our experiments, their software package and version, their lines of code, and the sanitizers [154] enabled at compilation time. The command line used to run them in the fuzzer is in Table 3.2. We applied both AddressSanitizer (ASAN) and UndefinedBehaviourSanitizer (UBSAN) compile-time instrumentation. However, we had to disable UBSAN for two applications as it introduced unwanted side-effects that made them crash even with the simplest test inputs.

Experimental Setup

We ran all experiments on a x86_64 machine equipped with an Intel® Xeon® Platinum 8260 CPU with a clock of 2.40 GHz. We used AFL++ version 2.65d as reference fuzzer to study the benefits of our approach and draw comparisons with the many configurations AFL++ offers (e.g., alternative mutation and seed scheduling policies, and context-sensitivity).

We ran each experiment 5 times to reduce the impact of fuzzing randomness, and report the median value to aggregate the results. Each experiment had a 48h budget.

Starting from an initial collection of valid files, we ran AFL++ for 24h

	Invariant pruning		
Program	**None**	**Learning**	**All**
catppt	137	137 (100%)	136 (99%)
xls2csv	453	400 (88%)	396 (87%)
jasper	11459	9144 (80%)	9144 (80%)
sndfile-info	3462	3013 (87%)	2996 (86%)
pcre2	4992	4803 (96%)	4497 (90%)
gm	16173	14362 (89%)	13278 (82%)
exiv2	6040	5534 (91%)	4943 (82%)
bison	9363	6263 (67%)	5983 (64%)
Total	52079	43556	41373
% (w.r.t. Unopt.)	100%	84%	79%

Table 3.3: Number of generated checks without any optimization, with optimizations for learning phase only, and with optimizations for learning & instrumentation phases.

and collected its queue as a corpus, which we used both as corpus for learning the likely invariants and as initial seeds for all the fuzzers we evaluate in our experiments. The same configuration was used in [22] for incremental fuzzing runs and allowed CGF fuzzers to approach saturation in our tests.

Throughout the rest of the section, we will denote with InvsCov a fuzzer that uses our invariant-based instrumentation as feedback, with CodeCov a fuzzer that uses classic edge coverage as feedback, and with CtxCov a fuzzer that augments edge coverage with context sensitivity.

3.3.1 RQ1: Invariant Pruning

To answer the first research question, we measured how the pruning rules introduced in §3.1.3 ultimately impact the number of tests for likely invariants that our system needs to insert into the program under test.

Table 3.3 reports the number of checks generated without any optimization enabled, with only those for the learning phase (comparability sets and removal of invariants impossible to violate) enabled, and with also the optimization applied at the instrumentation phase (overlapping conditions).

The optimizations from the learning phase reduce the amounts of checks by 14% on average. This resulted in an average of 1.4 likely invariants generated for each basic block that accesses one or more profiled variables (§3.2.1) in the LLVM IR. Upon adding the overlapping-conditions optimization from

	Testcases		Edges	
Program	InvsCov	CodeCov	InvsCov	CodeCov
catppt	213	119	404	404
xls2csv	1358	770	1013	1007
jasper	10831	3188	5452	5487
sndfile-info	1764	1297	8164	8074
pcre2	25534	15205	9831	9502
gm	12802	9488	25680	25216
exiv2	7016	5661	31201	31062
bison	5019	4419	6703	6700
Geo mean	3985	2466	5596	5548
% (w.r.t. CodeCov)	162%	100%	101%	100%

Table 3.4: Median number of testcases stored in the fuzzers' queues and edges covered over 5 trials of 48h.

the instrumentation phase, the total number of invariants decreased by 21%. While the overall reduction may seem small, according to our experiments the smaller number of invariants to check at run-time resulted in a 10% net increase in the performance of the fuzzer.

3.3.2 RQ2: State Explosion

The number of testcases maintained in the fuzzer's queue can serve well the purpose of verifying whether our technique would result in an explosion on the number of states the fuzzer has to track. In fact, the number of stored seeds is representative of the interesting testcases generated and therefore of distinct portions explored in the state space that is visible to the fuzzer. Table 3.4 reports the number of testcases in the fuzzer's queue after a 48h session. The growth due to the use of invariants is moderate, and only accounts for a 62% increase across all programs.

This is very important because an excessively large queue becomes unmanageable for a fuzzer. Wang et al. [162] studied queue sizes for two memory-based feedbacks (§2.2) and reported growth factors of 21x and 14x as geometric mean for the DARPA CGC benchmarks, and peaks of 196x and 512x. The authors also observed that the relative differences among most seeds were so small that they were very unlikely to lead to the discovery of new bugs. On the contrary, more moderate increases, such as ~8x over edge coverage for feedbacks focused on control flows (e.g., n-grams,

	Violated Checks		Exec / Sec	
Program	InvsCov	CodeCov	InvsCov	CodeCov
catppt	40	5	112	101
xls2csv	113	13	132	128
jasper	971	462	143	166
sndfile-info	558	214	151	152
pcre2	1524	286	2508	4381
gm	1874	715	63	65
exiv2	712	342	67	59
bison	387	234	57	65
Geo mean	458	134	145	156
% (w.r.t. CodeCov)	342%	100%	93%	100%

Table 3.5: Median number of checks violated by the testcases in the fuzzers' queues and average of the executions per second over 5 trials of 48h.

context-sensitivity), resulted in a profitable end-to-end bug finding.

In most of our programs we measured a growth factor below 2x, except for `jasper`, for which it was roughly 3x, yet far behind the numbers that were reported to cause state explosion in previous studies.

3.3.3 RQ3: Program State Exploration

Since our main goal is to help the fuzzer to explore various program states that can lead to bugs, we now look at how our proposed approach explores the program behaviors that would be visible to a pure code coverage-based approach.

First of all, we study the (cumulative) edge coverage on the original, un-instrumented program collectively exercised by executing the seeds (i.e., testcases) from the queues of InvsCov and CodeCov. Such coverage is a common metric in fuzzers evaluation, as a fuzzer cannot reveal a bug in a program point if it first does not explore it at least one time.

In Table 3.4 (column 'Edges') we report the median edge coverage of AFL++ when using, respectively, invariants or standard edge coverage as feedback. Overall, the differences are very small. For most targets, InvsCov results in a coverage comparable to CodeCov, showing that our technique does not result in a decrease of edge coverage. On some programs, our approach even helped the fuzzer to increase coverage over the saturated corpus, suggesting that some code paths may be reached only

with the right combination of conditions over some program state variables.

It is important to remember that the goal of our system is NOT to increase code coverage, but instead to increase the state coverage along the paths reached by a fuzzer. Therefore, we study the number of invariants violated by using our feedback mechanism compared to the traditional CodeCov, as a proxy of the improved program state coverage. The 'Violated Checks' column in Table 3.5 shows that AFL++ with InvsCov, thanks to our instrumentation mechanism (§3.2.2), maintains a set of testcases that violate more invariants than AFL++ with just CodeCov. Overall, our approach was 3.4x more effective than pure CodeCov at helping the fuzzer to visit different partitions of the program state.

3.3.4 RQ4: Bug Detection

Program	Default			MOpt			Rare		
	Invs Cov	Code Cov	⋂	Invs Cov	Code Cov	⋂	Invs Cov	Code Cov	⋂
catppt	3	3	3	3	3	3	3	3	3
xls2csv	17	15	13	18	17	15	17	16	14
jasper	7	5	5	8	5	4	8	4	4
sndfile-info	11	10	10	10	10	10	11	10	10
pcre2	77	35	28	81	52	36	80	48	38
gm	19	14	13	18	14	13	20	14	13
exiv2	8	7	7	8	7	7	8	7	7
bison	5	5	5	5	5	5	5	5	5
Total	147	94	84	151	113	93	152	107	94
% (w.r.t. CodeCov)	156 %	100 %	89 %	134 %	100 %	82 %	142 %	100 %	88 %

Table 3.6: Median unique bugs found with and without invariant-based feedback over 5 trials of 48h for each target program and three different fuzzers (Default, MOpt and Rare).

As the ultimate goal of Fuzz Testing is to detect bugs in programs we now analyze in more details the bugs InvsCov could find in our experiments and study their properties.

To compare InvsCov against classic edge coverage, we consider additional AFL++ configurations that exercise different designs in other components of the fuzzer, such as the scheduling strategies for mutations or seed selection. These strategies are orthogonal to the feedback function in

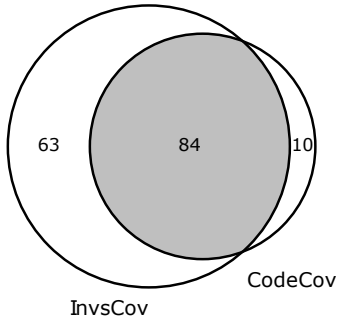


Figure 3.3: Venn Diagram showing the bugs DEFAULT found by either INVSCOV or CODECOV, and by both (gray).

use. Therefore, in the end, we expect INVSCOV to outperform CODECOV independently of other parameters. We believe this type of multi-pronged experiments allows for a more fair evaluation to isolate the contribution of the feedback technique alone.

In particular, we selected three AFL++ configurations for our tests:

- DEFAULT, i.e., the standard configuration of AFL++ used also for the other research questions;
- MOPT, i.e., AFL++ equipped with the MOPT [107] mutation scheduler, a powerful technique that dynamically prioritizes mutations according to their expected efficiency at any time in the execution;
- RARE, i.e., AFL++ scheduling that prioritizes seeds that exercise paths not along the 'hot' regions traversed by most seeds in the queue. Different, complex embodiments of this idea proved to be effective in a number of previous works [27] [98] [25].

We run these three state-of-the-art fuzzers on all target programs for 48h, to simulate a fuzzing campaign with a medium-small length, as previously used to evaluate fuzzers in works such as [123] and [22].

For crash de-duplication, we first grouped the reported crashes by using a standard call-stack hash from the stack trace. However, as automatic deduplication with stack hashes is generally unsound (it can both under- and over-count [92] depending on the case), we decided to manually inspect and triage each testcase.

Table 3.6 reports how many *unique manually deduplicated* bugs each fuzzer found over our set of subject programs[4]. The table also reports the

Program	Reached	InvsCov \ CodeCov
catppt	0	0
xls2csv	0	4
jasper	1	2
sndfile-info	1	1
pcre2	41	51
gm	0	6
exiv2	0	1
bison	0	0
Total	43	65

Table 3.7: Median number of bugs in the set difference between the InvsCov and CodeCov bugs (see Table 3.6) that are reached in coverage by CodeCov but not triggered.

intersection between the bugs found with our approach and with classic edge coverage. This relationship, summarized for all programs in the Venn diagram of Figure 3.3, highlights that guiding fuzzers by using state invariants not only results in more bugs being discovered, but also in *different* bugs[5].

Notably, the fuzzers that use our InvsCov feedback never underperformed with respect to the corresponding CodeCov versions, in all configurations. For two targets, `catppt` and `bison`, all fuzzers found the very same number of bugs, suggesting that these bugs are easy to trigger without particular requirements over the program state. Notably, on some of the targets (`sndfile-info`, `xls2csv`, `gm`, `exiv2`), the use of invariants allowed the fuzzer to also discover previously unknown bugs and vulnerabilities, like the one we used as running example in §3.1.

To better understand the bugs that only InvsCov was able to uncover, we classify them according to whether or not CodeCov was able to reach the crash point (obviously, without triggering it). We report the number of 'covered but not triggered' bugs for CodeCov in Table 3.7. It is interesting to observe that the instructions responsible for 43 of the 65 bugs discovered by InvsCov were reached by CodeCov, but not triggered due to the lack of the correct combination of state conditions required to trigger the bug upon reaching the flawed program point. These types of bugs are particularly common for `pcre2`. Since the program is essentially a parser that makes use of lookup tables, its program states are heavily data-dependent. The importance of the program state is also confirmed by the fact that some of the crashing locations were reached already by the initial input corpus we

supplied to the fuzzers. Thus, our approach shows a clear advantage for those programs that contain data dependencies in their flows.

For the remaining 21 bugs for which traditional fuzzers were unable to even reach the vulnerable location, a possible reason—as we discussed already for Table 3.4 (§3.3.3)—could be the fact that the use of invariants also allowed the fuzzer to achieve a slightly better code coverage. Indeed, specific conditions on program state values may be needed not only to uncover a fault but also to progress the exploration towards some code regions of the program under test.

As an additional set of experiments, we analyzed the default configuration of AFL++ with edge coverage augmented by context-sensitivity[6] (CtxCov), firstly introduced by Chen et al. [35], which turned out to be the form of feedback that revealed more bugs in the recent analysis of Wang et al. [162]. We report the number of triaged bugs for InvsCov and CtxCov in Table 3.8, running five 48-h trials with the same initial corpus of the other experiments.

Our experiments confirm that CtxCov performs better than CodeCov (+11%), revealing more unique bugs on four targets (2 on `xls2csv`, 6 on `pcre2`, 1 on `exiv2` and `gm`). Nonetheless, call-stack information for the context does not contain explicit information on program data, and InvsCov consistently finds more or different bugs than CtxCov as well (e.g., +47 on `pcre2`, +6 on `gm2`). The number of bugs found by both slightly improves for two subjects (2 bugs on `xls2cov` and `pcre2`) compared to CodeCov, suggesting that calling-context information offered AFL++ a different angle based on call paths to exercise the program states that trigger such bugs.

Finally, we also explored the hybrid scenario (marked as *Combined* in Table 3.8) in which we augmented edge coverage with both our invariants and context-sensitivity at once. This combined approach led to the discovery of another heap vulnerability in `libsndfile` (function `wavlike_ima_decode_block`). While this solution performs overall slightly worse than invariants alone, we observed promising peaks on single runs of `pcre2` (119 for Combined, 92 for InvsCov, 47 for CtxCov), `jasper` (12-8-6), and `sndfile-info`

[4]The classes we observe are the typical ones from sanitization with ASan and UBSan (e.g., heap and stack overflow, division by zero). As the bugs are many, we omit tedious information on their types for brevity.

[5]InvsCov may also miss bugs reported by CodeCov within a fixed time budget because of fuzzing entropy and different seed scheduling choices over different queues. However, those bugs are still within reach for InvsCov.

[6]Fuzzers can use calling-context information, i.e., the sequence of routine calls concurrently active on the stack when reaching a program location [48].

Program	InvsCov	CtxCov	∩	Combined
catppt	3	3	3	4
xls2csv	17	17	15	18
jasper	7	5	5	6
sndfile-info	11	10	10	11
pcre2	77	41	30	65
gm	19	15	13	21
exiv2	8	8	7	8
bison	5	5	5	5
Total	147	104	88	138
% (w.r.t. CtxCov)	141 %	100 %	85 %	133%
% (w.r.t. CodeCov)	156 %	111 %	94 %	147%

Table 3.8: Median number of bugs found with InvsCov and CtxCov, their intersection, and the bugs found with a fuzzer Combined that uses both feedbacks simultaneously.

(12-11-10). The downside of combining multiple feedback refinements is, in fact, that with larger queues (e.g., +79% on `pcre2` w.r.t. InvsCov) the randomness in seed scheduling impacts which program portions, and ultimately bugs, get explored in a limited time budget. We report the complete experimental data in Appendix §3.5, and leave the investigation of how to optimize combinations of this kind to future work.

3.3.5 RQ5: Run-Time Overhead

As our technique requires adding a more complex instrumentation to the program under test, it is reasonable to expect a higher run-time overhead with respect to CodeCov. Following the approach of the authors of RedQueen [16], we measured the average execution speed of AFL++ when executed on our target programs for 48h. Table 3.5 details (in column 'Exec / Sec') how many executions per second InvsCov and CodeCov were able to perform. The experiments show that our technique introduced on average a slowdown of 8%. We believe this to be a moderate price to pay to increase the ability of fuzzers to explore more (and more diverse) states of the programs under test.

A counter-intuitive result here is that for some programs the execution speed measured for InvsCov is higher than for CodeCov. The reason is that in some programs many invariant violations were triggered along

fast code paths: as InvsCov causes the fuzzer to spend more time on the same code path if one or more invariants are violated along it, the fuzzer ultimately focused on those parts and executed the other, slower paths less often than when using CodeCov, thus benefiting from shorter executions.

3.3.6 Discussion

The results of our experiments confirmed that our feedback, by distinguishing when program variables deviate from their 'usual' values, improves the sensitivity of a fuzzer for program states that code coverage alone fails to reward. Out of the 65 buggy program points that only our approach could drive to a crash, edge coverage alone was able to reach 43 of them, without however exposing the bug because the program was not in the correct state. Even when using refined code-based feedbacks like context-sensitivity, our approach continued to reveal more and different bugs than CGF.

Our tests also show that our instrumentation is tenable: it introduces only a moderate 62% growth on the fuzzer's queue size (orders of magnitude less than memory feedbacks, and still smaller than several code-based feedbacks [162]) and it slows down testcase execution by 8% on average. These costs are clearly amortized in our experiments by the many more unique bugs reported by our technique.

Finally, our feedback is not decremental in terms of code coverage compared to edge coverage and, in some cases, it can also 'unlock' more state-dependent program portions for further exploration. As briefly experimented in the *Combined* scenario, InvsCov and fine-grained forms of code feedbacks may also complement each other. Such a fuzzer would be able to better differentiate and explore those local state properties that are influenced by control-flow facts (e.g., the call path).

3.4 Limitations and Future Directions

Our approach augments the classic edge coverage feedback with information on violated likely invariants. As we emit AFL-compliant instrumentation for the sake of compatibility, there is a possibility of hash collisions—just like with AFL—when indexing the shared map for coverage updates. AFL++ offers an alternate link-time instrumentation scheme [86] that is collision-free but breaks compatibility. We may devise an InvsCov variant that benefits from such design, devising a solution that combines LTO with invariants like Borrello et al. [28] did for CtxCov or exploring split maps for the invariant and edge feedbacks.

On the methodological side, an intrinsic limitation of our approach is that it is not adaptive. We learn invariants once, while there could be potential to explore by refining them as the exploration advances and new value conditions are observed. A follow-up of our approach would then be to devise an online invariant mining module. Recent machine learning advances in anomaly detection like [46] could offer valid support to this end. We believe that a fuzzer that adaptively learns the state space partitions over variables can have a positive practical impact, and potentially help in desaturating fuzzing campaigns like OSS-Fuzz to catch more bugs in software already well-tested with CGF solutions.

Finally, as we study data facts at basic-block level, our likely invariants cannot capture 'implicit' relations between variables that do not get processed together in any block.

3.5 Appendix

In the following we report supplementary data for some of the experiments that we discussed throughout §6.4.

For the first research question, alongside the total number of invariants we also collected frequency metrics for probes at function and basic-block level. Due to the dynamic nature of the invariant mining process, only code actually reached in any execution from the input corpus can feature invariant checks. Table 3.9 reports figures computed over reached code only and with all our pruning optimizations enabled. While the code characteristics (most prominently, the varying complexity of individual functions) are reflected by heterogeneous values for invariants checked by a single function, when considering basic blocks we observe rather regular trends, with two peaks for `bison` and `jasper`, due to their basic blocks typically longer and richer of LLVM IR virtual register manipulations involving variables of interest for our method.

For the last set of experiments that we conducted for studying our bug finding capabilities, here we report statistics on the median queue size for the CtxCov and Combined fuzzers (Table 3.10), and the peak number of unique bugs identified among the 5 runs we made for each program under test (Table 3.11). The context-sensitive feedback benefited from our invariants as well, yet a larger queue impacts the program states explored by different runs within the 48h budget.

Program	Per Function	Per Block	Total
catppt	9.78 (14)	1 (137)	137
xls2csv	12.12 (33)	1.15 (349)	400
jasper	29.88 (306)	2.94 (3106)	9144
sndfile-info	20.09 (150)	1.01 (2979)	3013
pcre2	106.73 (45)	1.32 (3651)	4803
gm	28.78 (499)	1.38 (10391)	14362
exiv2	5.31 (1042)	1.17 (4708)	5534
bison	27.23 (230)	2.14 (2922)	6263
Geo mean	20.53	1.41	2784

Table 3.9: Average number of produced invariants for each function and basic block with at least one invariant. The reference baseline is reported between parentheses.

Program	InvsCov	CtxCov	Combined
catppt	213	149	281
xls2csv	1358	950	1766
jasper	10831	3528	18057
sndfile-info	1764	1525	2096
pcre2	25534	27227	45705
gm	12802	10928	14302
exiv2	7016	8457	9073
bison	5019	4975	6076
Geo mean	3985	3143	5356

Table 3.10: Median number of testcases in the queues of InvsCov, CtxCov, and Combined.

Program	InvsCov	CtxCov	Combined
catppt	4	3	4
xls2csv	19	19	19
jasper	8	6	12
sndfile-info	11	10	12
pcre2	92	47	119
gm	22	17	21
exiv2	8	8	8
bison	6	5	6
Total	170	115	201

Table 3.11: Peak number of unique bugs found by InvsCov, CtxCov, and Combined among the 5 runs.

Chapter 4

Fuzzing with Data Dependency Information

In the context of dynamic analysis, researchers have proposed many approaches to measure the coverage that a certain input produces in the software under testing. One of the possible metrics is *path coverage*, which refers to all independent paths present in a program. For example, in software testing, the community has focused on path coverage for tests generation [151, 137] with the goal of automatically producing inputs that can reach nested code locations. The main limitation of path coverage is that any non trivial application can contain a huge, and potentially infinite, number of paths [141, 106] thus making this approach a poor choice for a large set of programs. Because of this, Fuzzing mostly rely on simpler forms of coverage, such as the popular *edge coverage*.

As a necessary condition for a fuzzer to discover a vulnerability is the ability to generate an input that reaches the location of the flaw in the target program, code-coverage has become one of the most common metrics to gauge the effectiveness of a fuzzer and the success of a fuzzing campaign. In this context, *edge coverage* became the de-facto standard to measure code coverage. According to this paradigm, all basic blocks in the control flow graph (CFG) of the binary are instrumented to reward the fuzzer with a feedback when new edges connecting two basic blocks are discovered. In this way, the fuzzing engine can keep track of the new discoveries in terms of program points and mutate the generated inputs so that they evolve towards the exploration of nested portions of code. This gave origin to what is commonly referred as *coverage-guided* fuzzing.

A common goal among researchers is to develop new techniques to increase edge coverage, which has led current state-of-the-art fuzzers to be

very effective at visiting difficult-to-trigger code locations. For instance, Redqueen [16] can generate inputs that satisfy complicated conditions, like the ones that involve a comparison with some magic bytes. However, high code coverage alone does not always translate in a better ability to discover bugs, and therefore the fuzzing community had also explored other approaches based on alternative coverage metrics. In this direction, Parmesan [123] relies on the code locations instrumented with AddressSanitizer [148] and UndefinedBehaviorSanitizer [6] to build a directed fuzzer that tries to uncover and stress such locations. Ankou [112] adopted instead a different fitness function to guide the fuzzer by considering different combinations of the branches during the execution of the target. These approaches have shown promising results at detecting new bugs, thus suggesting that looking for alternative coverage approximations could help fuzzers to find different vulnerabilities compared to those normally found by employing edge-coverage.

Historically fuzz testing has taken inspiration from program analysis techniques to implement novel solutions that, in the past, have allowed to increase the fuzzers' performances. In 2019, Chowdhury et al. [38] rely on static analysis to make the target easier to be fuzzed, by simplifying the loop exit conditions and determining the ranges of valid input that can reach some error locations. Another example of the combination of fuzzing and program analysis approaches is the use of taint analysis to support fuzzing. This idea was adopted, for instance, by Rawat et al. in 2017 [139] and Chen et al. in 2018[35] to infer properties of the application that were used to generate more suitable input values, thus increasing the amount of visited code.

A common representation used in program analysis is the *Data Dependency Graph* (DDG), which could be used as a possible approximations of code coverage. DDG are very often adopted in other fields, such as compilers [81, 124], mostly used for code transformations or optimisations, and static software testing [169, 170] where interestingly it is used to determine the existence of vulnerable patterns in the source code. The fact that the DDG is already used for vulnerability discovery purposes suggests that it can be a good feedback candidate to drive a fuzzer to discover vulnerable paths. Our intuition behind this, is that the set of information contained in the DDG can provide the fuzzer with an additional feedback that the CFG alone cannot capture.

To verify our hypothesis we implemented a custom instrumentation approach, built on top of AFL++ [65] and LLVM [96], where the fuzzer is rewarded, on top of the traditional CFG instrumentation, every time a new

significant DDG edge is explored. We encountered several challenges to make our instrumentation lightweight, incremental with respect to the edge coverage, and effective in terms of discovered bugs. We tested our prototype implementation, DDFuzz, over three different datasets including a custom one of 10 real world applications, the popular benchmarking service Fuzzbench [116] and a third suite of programs previously used in the state-of-the-art [22].

The findings show that adopting the DDG coverage to guide fuzzers can lead to the discovery of additional (and different) bugs compared with traditional approaches. For instance, our data-dependency empowered fuzzer revealed 26 bugs that the traditional AFL++ strategy missed in our custom evaluation, by introducing a modest overhead of 10% compared to a normal edge-coverage instrumented binary. Moreover, our approach results to be incremental also to other state-of-the-art approaches such as Context Sensitivity and Ngrams. The second evaluation on Fuzzbench [116] that contains a larger variety of programs and bugs emphasizes these aspects as our approach performs better than AFL++ in 5 cases while for 3 further applications it is still able to find different bugs. As a confirmation of our methodology, DDFuzz discovers 12 more bugs compared to AFL++ in the third and final evaluation.

Contributions. In short, this chapter provides the following contributions:

- We propose a novel instrumentation method, DDFuzz, and we show its benefits as well as its limitations on a total of 38 target applications.
- We establish a reliable criteria based on the codebase structure which allows to predict when the use of our approach should be adopted for a fuzzing campaign.
- We test DDFuzz against several other state-of-the-art instrumentation approaches as well as a large range of targets demonstrating how, for each case, our custom instrumentation differs in terms of detected vulnerabilities.

The code of our prototype is available at

https://github.com/elManto/DDFuzz

4.1 Methodology and Implementation

We now illustrate the methodology and the implementation challenges that we encountered while developing our solution. The technique presented in this chapter is implemented as an LLVM [96] pass. While the code is compatible with different versions of the LLVM APIs, for our experiments we used LLVM 13. We chose LLVM for two main reasons: first, its intermediate representation is emitted in SSA [142] (Single Static Assignment) form, which means that each variable is assigned only once, and all variables must be defined before their first use. As we will see in the rest of the section, this simplifies the recovery of dependencies between LLVM IR variables with respect to other code representations where multiple definitions are allowed. Second, the LLVM toolchain is already well integrated into popular fuzzing projects, thus making our solution easy to plug into existing fuzzers' implementations, such as AFL++ [65]. This gives us the possibility to deploy our solution in an effective way, as well as to compare it with other instrumentation approaches that are already shipped with the AFL++ package. The following three subsections describe how our pass works by dividing the process into three main parts: DDG construction, dependency filtering, and target instrumentation.

4.1.1 DDG construction

The first decision we had make, when we started to develop our static analysis part, was the choice of the proper LLVM IR variables to construct the data dependency graph. The first intuitive approach was to recover the dependencies of *each* variable present in the LLVM bitcode by relying on the def-use edges to represent a dependency. However, we immediately noticed that such a technique does not fit well with our context. Indeed, because of the SSA form of the bitcode, this would produce too many dependencies to account for, which in turn would result in poor feedback for the fuzzer and in a large overhead in the execution of the target binary.

The LLVM framework applies some optimizations to construct its internal *Dependence Graph* [8], such as considering strongly connected components as a single node (the so-called *P-Node*). From now on, we will refer to this graph as $\texttt{DDG}_{\texttt{raw}}$, to indicate that we obtained it by the default LLVM implementation without applying any further transformations/optimizations. Although $\texttt{DDG}_{\texttt{raw}}$ is already an improvement with respect to the base strategy, it was still insufficient to overcome the performance issue. Nevertheless, before completely abandoning this road, we performed a benchmark where we evaluated this approach against the one that we ended

Algorithm 1: Data Dependency Graph builder

```
1  function BuildDDG(module)
2      blocks ← GetBasicBlocks(module)
3      DDG_b ← {}, ∀ b ∈ blocks
4      DFT ← {}
5      for b ∈ blocks do
6          instructions ← GetInstructions(b)
7          for i ∈ instructions do
8              if IsDefinition(i) then
9                  val ← GetValue(i)
10                 DFT_val ← {}
11             end
12             if IsGeneric(i) then
13                 val ← GetValue(i)
14                 ops ← GetOperands(i)
15                 InsertDataFlow(DFT, val, ops)
16             end
17             for u ∈ GetUses(i) do
18                 def ← QueryDataFlow(DFT, u)
19                 src ← GetParentBlock(def)
20                 DDG_b ← AddToSet(src)
21             end
22         end
23     end
24 end
```

up selecting. This was needed to ensure not to discard valid possibilities for our final prototype tool. Results of this preliminary experiment are reported along with the other evaluations in Section 4.2.3.

Our intuition then was to consider only a subset of LLVM variables, depending on how they are defined and used in the bitcode, and to recover the data dependencies that involve only this subset. This implies that we had to choose which instructions to consider as definitions and which one to retain as uses. The first observation was that, at the binary level, the actual data flow happens only when the memory is read or written. At the IR level, this led us to adopt the *Load* and *Store* instructions as a possible source and sink of our data flows. In addition to this, we added the *Call* instructions, and we considered them as uses of the variables, i.e., we track the dependency that reaches the function call parameters. Finally, we selected the *Alloca* instructions as a potential source of the def-use edge. Even though the compiler optimization passes would remove the majority of the *Alloca* defined variables, it can still be useful to track the dependencies that come from these variables when they are maintained in the emitted bitcode.

Algorithm 1 shows our solution. The two main data structures that we

use are the DDG itself (a map of sets) and what we call the *Data Flow Tracker* (DFT), initialized at the beginning of the algorithm. The DFT is as a map of sets, where the key is a LLVM *Value* and for each key we get a set of LLVM *Values* the key depends on. Our approach iterates over all the instructions present in each basic block (line 7) and, when we meet a *defining instruction* (i.e., *Load* and *Alloca*), we add an entry in the DFT as shown in the first `if` block (lines 8-10). For general purpose instructions, i.e., those instructions that are neither a source nor a sink, we first extract the operands and subsequently the return value of the instruction (12-14). The primitive `InsertDataFlow` is then responsible to track the fact that the return value is actually depending on the operands variables (line 15). To achieve this, it stores the Value `val` in the corresponding DFT set, whose key has a dependency with the operands `ops` that are involved in the instruction. The inner `for` loop iterates over all the uses in the instruction (line 17). For each of them, we extract the defining instructions by means of the DFT (with the primitive `QueryDataFlow`) and we add a new edge to the DDG (lines 18-20). `QueryDataFlow` iterates over the keys of DFT, looking for a match between `u` (the use) and one of the elements in the corresponding set, thus performing a recovery of all definitions that `u` depends upon. The final output is a data dependency graph, where an edge connects respectively a definition of a variable D and a use of a variable that depends on D. From now on, we will refer to such a graph as $\texttt{DDG}_{\texttt{full}}$.

4.1.2 Filtering

Given the goal of producing a lightweight instrumentation that has a limited impact on the performance of the compiled binary, we introduce a set of optimizations and filters to reduce the number of locations to instrument. This filtering phase helps us to discard dependencies that would not add any additional feedback to the fuzzer, as the associated transition is already captured by edge coverage. First of all, since our reference granularity is the basic block, any dependency within the same block of code is not significant. Similarly, multiple dependencies that connect the same two basic blocks are merged into a single one.

It is important to remember that the purpose of a DDG coverage instrumentation is not to help the fuzzer to reach complicated nested regions of code, but rather to revisit a determined program point by examining the different dependencies of the variables involved in such a path. In other words, not by visiting more code, but by triggering additional paths in already-explored code. Our hypothesis is that by exploring these additional dependencies we can uncover new flaws which would normally go unde-

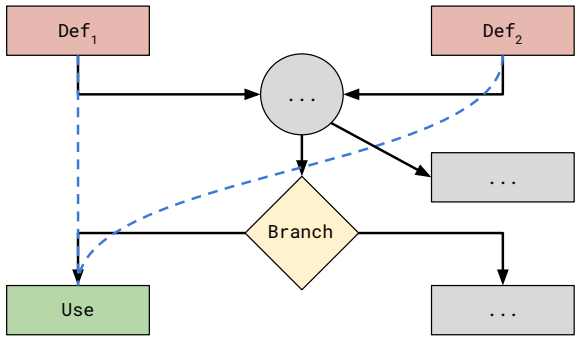


Figure 4.1: The configuration of the Definition and Uses that we want to isolate

tected. Because of that, we designed our instrumentation to co-exist with the classic edge coverage mechanism. This allows our fuzzing engine to receive two different feedbacks, the former useful to test different dependencies and the latter to further explore the application code. However, this also means that we had to reason about potential intersections of DDG and CFG coverage, which would lead to duplicate feedback. Thus, we now clarify which data dependencies we track and which ones we discard with our pass.

Essentially, we implemented two main rules to filter out redundant data dependencies. The first one is to check if a dependency is among two connected basic blocks, i.e., in which one is the successor of the other in the CFG. In this case, the dependency would not add any additional information that is not captured already by the edge coverage, and therefore would just increase the overhead without providing any useful feedback to the fuzzer. Therefore, in this condition we discard the data flow.

The second rule is an extension of the previous one and covers other scenarios where a data-dependency is already captured by edge coverage. In particular, we identified two additional types of data-flow. In the first one, the use U of a variable depends on a **single** definition D that can be located many basic blocks before U. In this scenario, we noticed that it is not worth maintaining the edge connecting the two basic blocks in our data structure because by definition if the program reaches the code block of U, it must have passed through the definition D already (since it was the only definition). Traditional edge coverage instrumentation already rewards the fuzzer when following this path, and therefore, for these situations, we do not track the data flow and discard the edge from our DDG.

Differently, we can have a configuration in which the use U of a variable depends on **more** than one definition, for instance the use of a ϕ-node variable [1]. As an example, we can consider Def_1 and Def_2, as represented in Figure 4.1. In the graph, the black arrows represent CFG edges while the blue arrows represent DDG edges. In this case, a fuzzer can reach 100% edge coverage while still triggering only one of the two def-use pairs. In fact, because of the mechanism used to log the edge coverage (i.e., the XOR of the current and previous BBs), the execution can reach the use always from the same edge, and therefore the fuzzer would not consider the two paths as two separate discoveries. As a result, this type of dependency is the only type of flow that we keep, as it is fundamental to reward the fuzzer in a different way compared to standard approaches (edge-coverage).

In the rest of the chapter, we will refer to the DDG obtained after the filtering phase as $\texttt{DDG}_{\texttt{filtered}}$. To summarize, $\texttt{DDG}_{\texttt{filtered}}$ is a graph that contains only flows represented with a def-use relationship and that have at least two definitions for the same used variable.

4.1.3 Instrumentation

As we explained in Section 2.1, typically AFL-based fuzzers log an edge visit by computing the XOR between the IDs of the current and previous locations and use this value as an index to access a bitmap that stores the number of times a certain edge has been hit. For our purposes, this is still necessary, because the DDG does not add any information to make the fuzzer explore deeper code but only to improve *how* to fuzz some specific code locations. Therefore, as a first instrumentation layer, we keep the traditional approach that is used in off-the-shelf fuzzers to log new edges inside a bitmap.

However, in the context of the DDG this solution does not work, since the two locations that we want to use as input for the XOR are not consecutive and the execution could go through many intermediate BBs before reaching the one that contains the data-dependent use. To solve this issue, we add an additional marker variable, originally set to 0, for each basic block that contains a definition we want to keep track of. Our instrumentation then changes the marker value to the ID of the block when it reaches the block itself. Finally, we instrument each basic block containing a use by generating a new bitmap index obtained by XORing the corresponding marker variables with the ID of the destination block. Because $N \oplus 0 = N$,

[1] A ϕ-node merges many versions of a variable into a new one in relationship with the incoming control flow. https://gcc.gnu.org/onlinedocs/gccint/SSA.html

```
// X, Y, Z are random IDs set at compile time
void function() {
    u16 on_block_X = 0;
    u16 on_block_Y = 0;

    int val;

    // this if-statement leads to two distinct definitions of the
     variable 'val' that is then used in the last if-statement,
     producing a dependency that we want to instrument
    if (...) {
        // block X
        on_block_X = X;
        int a = load_a();
        val = a;
    } else {
        // block Y
        on_block_Y = Y;
        int b = load_b();
        val = b;
    }

    // without these branches here the DDG edges would be
     included in the CFG and so pruned by our filtering pass
    if (...) {
        ...
    } else {
        ...
    }

    if (val == 0) {
        // block Z
        u16 idx = on_block_X ^
                  on_block_Y ^
                  Z;
        __afl_area_ptr[idx]++;
        use(val);
    } else {
        ...
    }
}
```

Listing 4.1: A simple example of how our instrumentation looks like.

Table 4.1: Dataset of the applications used for our study, along with their version, lines of code, command line and compilation information

Application	Package	Commit	KLOC	Command line	Sanitizers	Initial Queue
bison	bison	0ac1584	100	@@	ASAN	3
pcre2(*)	pcre2	65457aa	68		ASAN, UBSAN	3
c2m	mir	7670d7e	200	@@	ASAN, UBSAN	65
qbe	qbe	e0b94a3	10	@@	ASAN	52
faust	faust	f9aac26	115	@@	ASAN, UBSAN	27
readelf	binutils	68b975a	120	-s @@	ASAN	3
objdump	binutils	68b975a	120	-d @@	ASAN	3
libmagic(*)	file	d1ff3af	14		ASAN, UBSAN	3
tiff2pdf	libtiff	7d3b9da	63	@@	ASAN, UBSAN	10
openssl(*)	openssl	0d87763	234		ASAN	9

(*) indicates that an harness was used

the result will account only for the definition that was actually executed, thus resulting in different values when the same basic block is reached by following different data-flow paths.

For instance, we can consider the code in Listing 4.1, in which we highlighted the instrumentation inserted at the IR level. At the beginning of the function, we insert a marker variable for each definition block that we want to track, `on_block_X` and `on_block_Y` in our example. In the basic blocks containing the definitions, we set the corresponding marker to the ID of the block (`X` and `Y` respectively). Then, the basic block that contains the use is instrumented to compute the hash of the markers linked to `val` with the ID of the current block (`Z`). This hash is used as an index in the AFL bitmap.

4.2 Evaluation

To assess the validity of our approach, we performed a number of experiments to measure and compare the effects of our instrumentation in terms of bug detection, performance overhead, achieved code coverage, and the

Table 4.2: Median number of unique bugs detected in 5 trials over 24h of our evaluation against edge-coverage based fuzzing along with the intersection of bugs detected in the median case. Results include also the DD ratio for each of the targets, the slowdown introduced due to our DDG instrumentation and the p-value obtained with Mann-Whitney

Target	DD ratio	DDFuzz	Edge	DDFuzz ∩ Edge	Slowdown	P-value
bison	15%	5	3	3	2%	0.05
pcre2	14%	30	28	13	6%	0.46
c2m	23%	26	26	23	21%	0.55
qbe	15%	5	3	2	2%	0.04
faust	20%	4	2	2	18%	0.03
Total	-	70	62	43	-	-
Geomean	17.06%	9.51	6.66	5.14	6%	-
readelf	7%	4	4	4	2%	0.45
objdump	7%	5	5	4	1%	0.56
file	5%	1	1	1	1%	0.38
tiff2pdf	8%	9	13	9	4%	0.002
openssl	5%	2	2	2	5%	0.45
Total	-	21	25	20	-	-
Geomean	6.28%	3.49	3.24	3.10	2%	-

possibility to cause queue explosion. We believe that all these points are equally important to evaluate when proposing a new fuzzing approach, as they play a role in the overall effectiveness of a solution. Based on the results of our experiments, we try to understand when the adoption of our approach is useful for a fuzzing campaign and when, instead, it does not add any clear advantage with respect to the traditional edge coverage solution.

For our evaluation, we used 3 datasets. The first includes old versions of 10 real-world open-source projects known to contain bugs. For the first five applications, we selected software that we expected to contain a large number of data dependencies. For instance, we decided to include the `pcre2` library because of its frequent use of lookup tables, a C compiler frontend (`c2m`) and backend (`qbe`), a popular parser generator (`bison`), and `faust`, a compiler for a functional programming language used mostly for real-time signal processing. For the remaining five applications, we selected instead parser-related projects in which we expect data dependencies to be less

relevant. Three of them (namely `readelf`, `objdump` and `libmagic`) operate on the ELF file format while `tiff2pdf` performs image parsing and `openssl` is used to perform cryptographic operations.

The goal of this initial distinction, based purely on our expectation of the amount and impact of data dependencies, is to select some applications where our approach can provide an advantage and others where probably it is less useful. This can help us to better assess in which scenario an analyst should deploy a fuzzing campaign with our approach and when instead the traditional edge-coverage can provide better results.

Table 4.1 reports the list of applications in our first dataset, along with the hash commit that we tested and the command line that we used for the fuzzer. The fourth column reports the lines of code and clearly shows how our dataset contains software of different sizes, ranging from 10 KLOCs (`qbe`) to 234 KLOCs (`openssl`). In three cases, namely `pcre2`, `file` and `openssl`, we had to build an harness that invokes the core functionalities of the libraries. Furthermore, for each fuzzing campaign, we enabled the ASAN sanitizer, and, when it was possible, we also added the UBSAN sanitizer (some of the applications failed at building when UBSAN was enabled). We did not modify the compiler optimization level adopted by AFL++ (by default set to `O3`), to make the code execution more performant.

In addition to this first set of experiments, we also wanted to extend our evaluation to other datasets, to confirm our findings and prove that our approach can work for different codebases. However, many popular benchmark datasets which are used in other studies for fuzzing experiments are not suitable for our purpose. This is due to the fact that such benchmarks were designed to evaluate the amount of code covered by different fuzzers, often by inserting artificial bugs. Although code reachability is a fundamental aspect of a fuzzing approach, our solution does not add any improvement in terms of code coverage (at least not directly). Rather, it tries to use data dependencies to augment the amount of discovered bugs – which does not necessarily imply exploring new program points. As a consequence, we believe that datasets such as LAVA-M [51] are not ideal candidates for a proper evaluation. On the other hand, both MAGMA [80] and Fuzzbench [116] satisfy our requirements as they focus on a number of bugs and on a large variety of applications. Among the two, we selected Fuzzbench [116] because it contains a larger number of programs. In total, this second dataset includes 22 different programs (we had to disable three applications because they did not compile under clang-based instrumentations).

Finally, we added a third dataset to our evaluation, this time taken from

a recent paper by Blazytko et al. [22]. This last benchmark is composed of targets not selected by us, thus without any a-priori knowledge about the structure of the programs, but that may still contain targets with relevant data-dependencies as the focus of the original paper was to deal with highly-structured inputs and complex parsing code. This final dataset included 6 applications.

4.2.1 Experiment Setup

We performed the experiments with the first and the third dataset on a `x86_64` server containing an Intel Xeon Platinum 8260 CPU with a clock frequency of 2.40GHz. As already explained, we adopted AFL++ (version 3.14) as a reference fuzzer, both for our implementation and evaluation. One of the advantages that come with using this project is that it already implements many feedback approaches, such as Edge Coverage, Context Sensitivity, and NGram coverage (from now on respectively Edge, Ctx and Ngram). Thus, it provides the ideal comparison for our evaluation purposes. In the follow-up of the paper, we will use interchangeably Edge and AFL++ to indicate the use of standard edge coverage mechanism whereas we will specify Ctx and Ngram to identify the adoption of alternative feedback techniques.

Each experiment was made of 5 trials of 24 hours to limit the randomness of the fuzzer. The initial seeds were mostly taken from the `test` directories that were already present in the repositories of the projects to ensure that they represented valid input files for that application. The initial queue size is reported in Table 4.1 for each target program in our custom evaluation set.

For the experiments on the Fuzzbench targets, we adopted the default configurations described on the public website page [10]. In this case, each test consists of 20 trials of 23 hours for each fuzzer.

Moreover, as the authors of [92] suggest, we computed the p-value resulting from the Mann-Whitney U test, a nonparametric test used to compare differences between two independent randomly selected sample values that originate from two distinct populations. For our purposes, the test checks that our prototype fuzzer is statistically different from the competitor one, i.e., the default AFL++, in terms of bugs detected during the trials.

4.2.2 Comparison against edge coverage

For our initial evaluation, we want to compare our data dependency coverage instrumentation against the traditional edge coverage approach. The

first question that we want to answer is whether our approach is helpful in terms of discovered bugs. We also report a simple metric to capture the impact of our pass by counting the amount of data dependencies in the target application. This can tell the analyst if it makes sense to enable our instrumentation for a new fuzzing campaign. This metric, which we call *DD ratio*, is the ratio between the basic blocks that we instrumented with the dependencies information (as described in Section 4.1) over the total number of basic blocks in the program.

We computed such a value for the ten applications in our dataset. Results are reported in the first column of Table 4.2. We can observe that the value of *DD ratio* changes quite sharply from application to application, with a minimum of 5% for `openssl` to a maximum of 23% for `c2m`. The other interesting and more important aspect is that the two sets of applications are clearly separated by their `DD ratio` value, confirming our intuition that the first five were more data-flow intensive. For instance, if we adopt a threshold of 10%, that derives from the geometrical mean of all ratios for each target, we can easily classify our codebases into *strongly* (above the threshold) and *weakly* (below the threshold) data-dependent applications.

We then ran our fuzzer evaluation against the traditional edge coverage instrumentation. Results are reported in the remaining columns of Table 4.2, where DDFuzz and Edge indicate respectively the median of the unique bugs found by the two approaches while the column with label DDFuzz ∩ Edge represents the findings that are common among the two (i.e., the intersection). Such numbers are the result of a careful triage phase that was conducted both in an automated fashion along with a rigorous manual check. As a first step, we de-duplicated bugs by grouping them according to their call-stack hash collected from the stack trace that we obtained by the sanitizers. However, this simple approach is generally error-prone (as shown in [92]) and thus we manually inspected all remaining test cases to avoid any possible duplicates.

The first way to interpret the results is by comparing the number of discovered bugs by the two approaches. Overall, DDFuzz found 8 vulnerable flaws more than Edge in the 5 strongly data-dependent applications. With the exception of `c2m`, where the median is the same, for the other four cases, the fuzzer equipped with our instrumentation discovered two additional bugs each. Viceversa, for the remaining five targets that do not exhibit a high dependency in their code base, the number of discovered bugs remained mostly the same. The only exception is `tiff2pdf`, where DDFuzz found 4 bugs less than Edge – probably due to the fact that triggering the bugs was mostly a code reachability problem.

While these results are promising for strongly data-dependent programs, the intersections between the discovered bugs can tell us even more. In fact, not only our approach allows to detect more bugs on average, but it can find some that are never discovered by EDGE in any of the trials. For instance, for `c2m`, even if the number of unique bugs is the same, there are three vulnerable points that are only detected by our approach. This is even more evident if we consider the intersections for `pcre2`, where DDFUZZ finds a total of 17 unique bugs which are not revealed by the vanilla AFL++. For the remaining cases instead (`bison`, `qbe` and `faust`) the unique bugs identified by DDFUZZ are mostly a super set of the ones individuated by EDGE. This trend holds until a sufficiently high data dependency exists in the analyzed code. Indeed, if we consider the last five rows, where the `DD ratio` is always under 8%, the intersections converge towards the total number of bugs – i.e., the additional instrumentation of DDFUZZ had a negligible impact on the results.

We also measured the overhead introduced by our additional instrumentation, compared again with that introduced by EDGE alone. The results, reported in Table 4.2, show that the overhead introduced by our approach is overall modest. The `slowdown` factor was computed as the difference between 1 and the ratio among the DDFUZZ average executions per second over the EDGE executions per second (and then transformed in percentage). For the five strongly data-dependent applications, the average slowdown is 10% (6% if we consider the geometrical mean), despite the fact that we observed a non-negligible variation of this value. Such a variation is only in part justified by the `DD ratio`, because other factors influence it, such as the execution path, the control flow executed and the fuzzer execution mode (i.e., if we used an harness or we make use of the fork system call to spawn the target process). For the remaining five applications instead, the executions per second are almost the same between the two fuzzing approaches, demonstrating again that, in the case of a weakly data-dependent code structure, our methodology converges towards the default AFL++. The last column of Table 4.2 concludes our measurements about this experiment with the p-value resulting from the Mann-Whitney U test computed over the bugs found for each trial. The p-value is significant ($<= 0.05$) in 3 out of the 5 cases for the strongly-dependent programs while only one weakly-dependent application reports a p-value statistically significant. This again suggests that for high values of `DD ratio`, DDFUZZ behaves differently from EDGE.

Table 4.3: Comparison of the effects of our filtering strategies in terms of median number of bugs over 5 trials of 24h each

Target	DDFuzz	$\text{DDFuzz}_{\text{full}}$	Intersection	Decrease
bison	5	4	4	67%
pcre2	30	29	24	41%
c2m	26	16	13	44%
qbe	5	4	4	49%
faust	5	1	1	52%
Total	71	54	46	-
Geomean	9.95	5.94	5.49	49.85%

4.2.3 Effects of our instrumentation filters

As presented in Section 4.1, before performing the actual instrumentation of the DDG, we obtain three different versions of the data dependency graph. The first is $\texttt{DDG}_{\texttt{raw}}$ which is the result of the default LLVM dependence graph implementation. The second instead is $\texttt{DDG}_{\texttt{full}}$ which is the output of Algorithm 1 and serves as base graph to produce our final version, $\texttt{DDG}_{\texttt{filtered}}$. As already explained, we ended up instrumenting this last version of the DDG in our final implementation of DDFuzz, which is used across the whole paper. However, in the current section, we want to measure the effects of our optimizations compared to the other two flavors of the DDG ($\texttt{DDG}_{\texttt{raw}}$ and $\texttt{DDG}_{\texttt{full}}$).

As a first step, we compare the performances that result from the instrumentation of $\texttt{DDG}_{\texttt{raw}}$ (the LLVM default implementation) against the final version of the DDG ($\texttt{DDG}_{\texttt{filtered}}$). For this, we selected an applications in our dataset (qbe) for which our approach had an average performance and instrumented it by using either the default LLVM data dependency graph (i.e., $\texttt{DDG}_{\texttt{raw}}$) or our optimized version. We then launched two fuzzing campaigns according to the experiment setup described in the previous subsection.

We immediately noticed that the instrumentation based on $\texttt{DDG}_{\texttt{raw}}$ was introducing a major overhead since we had to increase the timeout of AFL++ due to the fact that the instrumented program was not terminating within the default time interval. After 24 hours, the average executions per second was 50% higher with our optimization (362 vs. 240), and this resulted in a median of five discovered bugs for DDFuzz versus two (always a subset of the five) when using $\texttt{DDG}_{\texttt{raw}}$.

Table 4.4: A comparison of Data Dependency coverage against Ngram2, Ngram4 and Ctx instrumentation approaches in terms of median number of bugs over 5 trials of 24h each

Target	DDFuzz	Ngram2	Ngram4	Ctx	DDFuzz ∩ Ngram2	DDFuzz ∩ Ngram4	DDFuzz ∩ Ctx
bison	5	5	4	3	3	3	3
pcre2	30	29	23	33	15	14	17
c2m	26	27	27	17	23	22	12
qbe	5	3	4	5	3	4	4
faust	5	2	2	1	2	2	1
Total	71	66	60	59	46	45	37
Geomean	9.94	7.48	7.23	6.09	5.73	5.93	4.76

For the second measurement, we wanted to quantify the effects of our pruning strategies. As described in Section 4.1.2, we filtered $\texttt{DDG}_{\texttt{full}}$ to exclude the edges which are already covered with Edge, producing as a final output what we call $\texttt{DDG}_{\texttt{filtered}}$. To assess to which extent these filtering strategies impacted the performances of our final implementation, we performed another experiment with the same setup described in Section 4.2.1, comparing the filtered with the unfiltered graphs. In this case we selected only the five applications that exhibit a sufficient amount of data dependency as we are interested at measuring the effects of our filter strategies, and it would not be meaningful to consider applications where our approach instruments only a small amount of locations. Results of this evaluation are listed in Table 4.3 where we refer to DDFuzz as our prototype implementation after the application of the filters and to $\texttt{DDFuzz}_{\texttt{full}}$ as the implementation obtained from our pass without the filtering steps (i.e., the instrumentation of $\texttt{DDG}_{\texttt{full}}$). Moreover, we computed the intersections of bugs between $\texttt{DDFuzz}_{\texttt{full}}$ and DDFuzz (third column) and the percentage decrease of the instrumented locations (fourth column).

We can immediately notice that bugs discovered by DDFuzz are mostly a superset of those which are detected with $\texttt{DDG}_{\texttt{full}}$. This is because the feedback produced by the more dense instrumentation is less informative in terms of the dependencies that were reached. However, the approach can still find some interesting program points as demonstrated by the intersections with DDFuzz. In facts, for `pcre2` and `c2m`, 8 distinct bugs are detected by the more naive approach and missed by DDFuzz in the median

case.

Overall, the instrumented locations decreased by 52% in average (49% geometrical mean) with respect to the version without the filters enabled. This resulted in better performances and indeed we observed an average slowdown of roughly 20% compared to EDGE (while, as already shown, DDFUZZ slowdown was 10%). Therefore, we can conclude that our filtering strategies represent an improvement of the standard instrumentation of $\texttt{DDG}_{\texttt{full}}$ and justify our design choices.

4.2.4 Comparison against different instrumentation strategies

For the second evaluation of our approach on the custom dataset, we want to compare DDFUZZ against different instrumentation approaches. As we have shown in Section 4.1, one of the major points of our instrumentation technique is that the edges in the DDG can connect basic blocks which are not necessarily consecutive, i.e., when other intermediate basic blocks exist between the source and the sink. Thus, our first hypothesis was to examine instrumentation passes such as the ones that rely on Ngram coverage (NGRAM). The simple idea behind these is that when computing the XOR of the edges to find the bitmap index where to log the new discovered program point, NGRAM approaches take into account multiple edges. For instance, by adopting NGRAM2, the index of the bitmap involves the XOR of the previous 2 locations, while EDGE would consider only the previous location. In other words, EDGE is equivalent to a NGRAM1 instrumentation. For our tests, we chose to compile our target binaries with NGRAM2 and NGRAM4 because, according to Wang et al. [162], these are the two approaches that provide the best results in terms of number of discovered bugs. We also compared against Context Sensitivity [162] (CTX), a recent approach that takes into account the callstack in addition to the reached basic blocks. According to its authors, together with the two aforementioned NGRAM instrumentations context sensitivity was the solution that provided better results.

The results of our second evaluation are showed in Table 4.4, limited again to only the five strongly data-dependent binaries. As in the previous experiment, we report the median of unique bugs, de-duplicated as explained in 4.2.2. The first four columns contain the median detected bugs depending on each of the four instrumentations that we wanted to include, while the remaining three columns instead show the intersections with DDFUZZ. There is no solution that outperforms all the others for all

Table 4.5: A comparison of the median values of line and function coverage among the programs according to each different instrumentation over 5 trials of 24h each

Target	DDFuzz		Edge		Ngram2		Ngram4		Ctx	
	Lines	Func	Lines	Func	Lines	Func	Lines	Func	Lines	Func
bison	46.1%	49.7%	43.9%	47.6%	47.5%	50.3%	47.5%	50.3%	47.5%	50.3%
pcre2	53.2%	32.2%	53.8%	32.4%	54.1%	32.4%	54.2%	32.4%	54.2%	32.4%
c2m	48.8%	55.2%	48.8%	55.2%	48.9%	55.2%	48.9%	55.2%	49.2%	55.8%
qbe	76.9%	85.2%	76.6%	85.2%	77.0%	85.0%	77.1%	85.0%	77.0%	85.0%
faust	26.3%	28.5%	26.7%	28.4%	26.7%	28.5%	26.8%	28.5%	26.8%	28.5%

Table 4.6: Comparison among the average queue sizes over 5 trials of 24h each when data dependency coverage and edge coverage are applied, along with their ratio

Target	DDFuzz	Edge	Ratio
bison	5,173	2,986	x1.7
pcre2	119,180	16,282	x7.3
c2m	14,323	13,269	x1.1
qbe	2,748	1,814	x1.5
faust	2,934	2,633	x1.1

programs, but DDFuzz results are the best for three out of five applications and the best overall – with 71 discovered bugs vs. 66 of Ngram2 (the second-best performer).

Again, by looking at the intersection, we can see that the bugs detected thanks to the use of our data dependency instrumentation are very different for each of the tested applications. In each one, DDFuzz finds at least one different bug that the other approaches could not find. It total, it was able to detect 25 bugs that were never triggered by Ngram2, 26 more than Ngram4, and 34 not captured by the Ctx instrumentation. This shows once more that our extension could provide a very clear benefit for applications that have a rich set of data dependencies and that even when DDFuzz is not the approach that finds more bugs overall, it always leads to detect some different ones. Moreover, it is important to underline that we believe that all these approaches can (and should) be combined during a fuzzing campaign.

4.2.5 Queue Explosion

If the number of vulnerable points detected is an important metric for a fuzzer effectiveness, the number of inputs generated influences its usability and could result in poor feedback for the fuzzer.

The authors of [162] found that an increase factor up to ∼8x is still manageable by the fuzzer and would allow to provide relevant feedback without incurring in a queue explosion problem. These results were derived from experiments with different coverage instrumentations, such as NGRAM and CTX. On the other hand, the authors observed growth factors of 21x and 14x when using two types of memory feedbacks and concluded that such values were potentially leading to the explosion in the queue size.

For this, we compute the average size of the queue for our five strongly data-dependent applications, both in the baseline case with only edge coverage instrumentation and with our data dependency instrumentation. Table 4.6 shows the results along with the ratio obtained by dividing the DDFUZZ queue size by the EDGE size. The numbers show that the overall increase is quite moderate. With the exception of `pcre2`, where the increase accounts for a factor of 7.3x, all other factors are below 2x, thus showing that our technique results in additional feedback but not large enough to cause problems in the input queue.

4.2.6 Code Coverage

As a last experiment on our custom benchmark, we decided to compare the values of the code coverage that we obtained for our strongly data-dependent applications. For this test, we used afl-cov [7] which represents the default solution to measure this metric and is compatible with AFL++ based fuzzers. The tool was launched against the different instrumentation types that we tested and produced two values for each one: line coverage and function coverage. Results are reported in Table 4.5 and show the median values that we observed in our experiments.

If we look at the first two columns that correspond to DDFUZZ and EDGE, we observe that there is not one of the two which dominates the other. In two cases (`bison` and `qbe`) DDFUZZ performs better whereas for `c2m` we registered the same line coverage. For the remaining two projects instead, EDGE reaches a major number of program points.

Other forms of instrumentation led instead to better code coverage. However, in the worst case (`bison`), the line coverage difference with the best approaches is 1.4% while for the other projects is always less than 1%. This result is in line with our expectations since our approach is not de-

Table 4.7: Total unique bugs found accross all 20 trials of 23h by DDFuzz and Edge on FuzzBench

Benchmark	Total bugs	Edge	DDFuzz	DDFuzz ∩ Edge	DD ratio
arrow_parquet-arrow-fuzz	105	93	86	74	1%
proj4_standard_fuzzer	0	0	0	0	2%
muparser_set_eval_fuzzer	0	0	0	0	3%
openh264_decoder_fuzzer	10	10	8	8	3%
aspell_aspell_fuzzer	0	0	0	0	4%
systemd_fuzz-varlink	0	0	0	0	4%
tpm2_tpm2_execute_command_fuzzer	0	0	0	0	4%
file_magic_fuzzer	0	0	0	0	7%
libgit2_objects_fuzzer	2	2	2	2	5%
grok_grk_decompress_fuzzer	4	4	2	2	7%
stb_stbi_read_fuzzer	11	11	11	11	8%
njs_njs_process_script_fuzzer	0	0	0	0	11%
php_php-fuzz-execute	21	13	16	8	11%
libxml2_libxml2_xml_reader_for_file_fuzzer	13	11	12	10	12%
libhtp_fuzz_htp	7	6	7	6	13%
matio_matio_fuzzer	22	20	21	19	14%
php_php-fuzz-parser-2020-07-25	13	12	12	10	14%
poppler_pdf_fuzzer	5	4	3	2	14%
libarchive_libarchive_fuzzer	0	0	0	0	15%
zstd_stream_decompress	0	0	0	0	15%
usrsctp_fuzzer_connect	0	0	0	0	17%
libhevc_hevc_dec_fuzzer	3	1	3	1	19%
Total	214	187	183	150	-
Geomean	9.72	7.98	8.20	6.38	7%

signed to increase coverage, but only to increase paths on already-covered areas of code.

4.2.7 FuzzBench

We now look at the results we obtained from the experiments we performed on Fuzzbench [116]. The main goal was to extend our approach to a broader range of applications and bugs to verify whether our findings could be generalized beyond our test programs.

In these experiments we limited our comparison to DDFuzz and Edge, because the reports generated by Fuzzbench do not include the information

Table 4.8: Median relative code-coverage accross all 20 trials of 23h by DDFuzz and Edge on FuzzBench

Benchmark	Edge	DDFuzz
arrow_parquet-arrow-fuzz	96.43	95.02
proj4_standard_fuzzer	100.00	100.00
muparser_set_eval_fuzzer	97.91	97.91
openh264_decoder_fuzzer	99.38	98.91
aspell_aspell_fuzzer	99.91	99.86
systemd_fuzz-varlink	100.00	100.00
tpm2_tpm2_execute_command_fuzzer	96.16	88.62
file_magic_fuzzer	81.65	78.34
libgit2_objects_fuzzer	99.75	99.63
grok_grk_decompress_fuzzer	94.58	91.41
stb_stbi_read_fuzzer	94.81	93.01
njs_njs_process_script_fuzzer	96.22	93.66
php_php-fuzz-execute	96.15	92.77
libxml2_libxml2_xml_reader_for_file_-fuzzer	94.73	92.85
libhtp_fuzz_htp	99.94	99.85
matio_matio_fuzzer	99.33	99.29
php_php-fuzz-parser-2020-07-25	99.12	98.08
poppler_pdf_fuzzer	97.95	97.96
libarchive_libarchive_fuzzer	96.51	81.32
zstd_stream_decompress	98.12	97.84
usrsctp_fuzzer_connect	99.58	99.65
libhevc_hevc_dec_fuzzer	96.14	93.28

about the intersections of discovered found but only an aggregated value representing the sum of the unique bugs for all fuzzers. Therefore, if we had included other instrumentation approaches (as, for instance, Ctx and Ngram) we would have not been able to distinguish among the bugs found by each technique. The experiment consisted of 20 trials of 23 hours over a total of 22 real-world projects. We report the results in Table 4.7, where, for each target, we list the bugs revealed by the two instrumentation approaches, the sum of the two, the intersection as well as the DD ratio, that we introduced in Section 4.2.2. To compute the DD ratio, we built the 22 projects on our local machine, according to the docker specifications reported by Fuzzbench to mimic the same environment setup. This allowed us to run our instrumentation pass and log the instrumented locations and the total number of basic blocks necessary to compute the ratio. The targets in Table 4.7 are sorted by the DD ratio with the weakly data-dependent

applications in the first half and the strongly data-dependent projects in the second.

If we look at the total unique bugs detected by the standard AFL++, we notice that they are more than the amount of vulnerabilities triggered by DDFuzz (respectively 187 and 183). On the other hand, the geometrical mean, which indicates a central tendency by flattening the outlier values, tells us that DDFuzz finds in the mean case 8.20 bugs while Edge stops at 7.98 vulnerabilities.

By looking at individual applications, we can notice that Edge performs better in 4 cases while DDCov in 5. For the remaining 13 benchmarks, in 10 cases none of the fuzzers reported any interesting finding (both fuzzers properly ran but did not trigger any vulnerable location) whereas for 3 cases the bugs discovered are the same (`libgit2_objects_fuzzer`, `stb_stbi_read_fuzzer`, `php_php-fuzz-parser-2020-07-25`), despite the fact that for `php_php-fuzz-parser-2020-07-25` the two fuzzers find 2 different bugs. Moreover, it is interesting that in two cases where Edge finds more vulnerabilities, our data dependency instrumentation can still trigger different buggy locations that were not detected by the edge coverage. More specifically, for `arrow_parquet-arrow_fuzz` the intersection of bugs is 74 which means that DDFuzz finds 12 different bugs, and the same happens for one bug discovered in `poppler_pdf_fuzzer`. Overall the outcomes of this experiment confirms that the feedback produced by our instrumentation is different from the standard edge coverage and results in different program points reached during the fuzzing session. This inherently does not always imply more bugs but can result in some different ones.

For the 5 projects where DDFuzz works better, the `DD ratio` indicates a high value of data dependency (above the 10% threshold we defined in Section 4.2.2) whereas for 3 out of the 4 projects where Edge wins, the `DD ratio` suggests that the application has a lower amount of data dependencies. The only exception is `poppler_pdf_fuzzer`, where Edge detected 4 bugs vs 3 of DDFuzz, despite a data-dependency ratio of 14%.

Overall, this confirms once more that the `DD ratio` can be used as a criteria to predict the type of instrumentation that would provide better results. To conclude our overview of the bugs, we extracted the statistical significance data included in the Fuzzbench report. Interestingly, both for *DDFuzz* and Edge the bug coverage is statistically significant (i.e., p-value $\leq \sim 0.05$) in 7 cases out of the 12 projects where they can detect at least one vulnerability.

As a parallel consideration, we computed the coverage that our prototype fuzzer reached in the Fuzzbench targets. Table 4.8 shows for AFL++

Table 4.9: Outcome of the third evaluation over a dataset of 6 programs. The results include the DD ratio, the median number of bugs found with DDFuzz and Edge coverage, their median intersection, the Slowdown introduced with our instrumentation and the p-value that we obtained with the Mann-Whitney test.

Target	DD ratio	DDFuzz	Edge	DDFuzz ∩ Edge	Slowdown	P-value
nasm	3%	4	4	4	1%	0.12
sqlite	7%	1	2	1	15%	0.001
lua	11%	9	2	2	20%	0.005
boolector	14%	3	1	1	6%	0.13
mruby	17%	3	3	3	35%	0.45
tcc	12%	11	8	8	7%	0.05
Total	-	31	20	19	-	-
Geomean	9.3%	3.9	2.6	2.4	8.7%	-

and DDFuzz alike, the median relative code coverage that we obtained across the 20 trials for each target. According to the Fuzzbench documentation, the relative code coverage for a trial is computed as the ratio between the single-trial coverage over the maximum coverage obtained during the experiment. It is quite evident that for the majority of the programs (17 out of 22), Edge results in a better code coverage while DDFuzz can do better only in 2 cases. However, it is interesting to observe that for all targets where DDFuzz finds more bugs, it also reaches a lower code coverage. This confirms once more our intuition that fuzzing the data dependency edges does not help to discover new program points but suggests how to "stress" the already discovered code locations in a different way.

4.2.8 Third Dataset

As a final proof of our approach efficacy, we opted to select another dataset, made of real-world programs, and adopt it to compare against Edge. We reviewed recent state-of-the-art papers looking for other fuzzing solutions dedicated to specific classes of target applications (since *DDFuzz* works best with highly data-dependent binaries). At the end we settled for the dataset of Blazytko et al. [22], who evaluated their structure-aware fuzzer on a set of 8 programs, that we report in Table 4.9 (note that we excluded `PHP` and `libxml2` as they are already included in the experiment we ran on

the Fuzzbench dataset, and a second evaluation of these targets would be unfair, as they both resulted in more bugs for *DDFuzz*, see Tab. 4.7). For the experiment setup, we used the same configuration described in Section 4.2.1.

DDFuzz outperformed Edge in 3 targets out of the 4 with a DD ratio above the threshold (10%), with 11 more bugs overall. On the two targets with a low DD ratio, our technique performs like the baseline on `nasm` while on `sqlite` it underperforms Edge missing one bug. The Mann-Whitney U test results in a significative p-value in 3 of the 6 considered cases. With the exception of `boolector`, where the test produces a p-value major than 0.05, the two remaining applications (`nasm` and `mruby`) that result in a high p-value justify this due to the similar number of bugs during the 5 trials. This evaluation further confirms that the DD ratio is a good predictor of the efficacy of DDFuzz and thus the insight that our technique should be applied to a specific class of data-dependent programs.

4.2.9 Classes of Bugs

Another interesting measurement that we carried out on top of our experiments is to study the classes of bugs that DDFuzz revealed during the several trials. For each of the detected bugs in the median case, we analysed the reports generated with ASAN for the two datasets that we tested on our servers. Table 4.10 reports the results in the column **Total** while the last column shows the bugs that only DDFuzz could find.

The table shows that DDFuzz can detect several types of bugs, with a prevalence for Heap Buffer Overflows (38) and Undefined Behaviors that generate the signal `ILL` (36). However, many of these bugs are the same that also the vanilla version of AFL++ can spot during our tests. Thus, we isolated the ones that are different among the two fuzzers, and found that the majority accounts for Stack Overflows (11 instances) and Heap Buffer Overflows (11) while all the other cases are almost equally distributed between the other classes of bugs.

4.3 A Bug Case Study

In this section, we present a case study about a sample bug that we found while triaging the crashes that we obtained during our experiments. Our goal is to show an example of how `DDFuzz` succeeds in real life and differs from previous approaches at inspecting the program state.

The application that we consider is `tcc`, a project that implements a fast and small C compiler (roughly 50K lines of code). During the triage

Table 4.10: Classes of bugs detected with DDFuzz in the median case

Bug Class	Total	DDFuzz Only
Heap BOF	38	11
Global BOF	5	3
Stack BOF	1	0
Heap UAF	4	2
Stack Overflow	19	11
Invalid Ptr Deref	11	3
Invalid Allocation Size	1	0
ABRT	2	1
ILL	36	5

phase, `AddressSanitizer` reports the presence of a Global Buffer Overflow. Interestingly, no edge-coverage trial reports the same bug and therefore we opted to re-implement the approach described in the paper by Wang et al. [162] to investigate the crash and understand the reason why only one of the two fuzzers was able to detect it.

The first step consists of reconstructing the testcase tree that originated the crash. This is possible because AFL++ stores the newly generated testcases in the queue by naming them with additional info such as the time, the mutation strategy and the previous testcase whose the mutation originated the current one. Moreover, when the fuzzer applies a splicing strategy, the two parents' names are preserved, thus allowing to recover all original testcases also in this second scenario. After recovering the testcase tree, the next step is to show if each mutation generates novelty according to the fuzzer bitmap. Indeed, in case a certain testcase in the tree does not generate novelty for the edge-coverage fuzzer, it means that the fuzzer would have skipped the testcase, loosing one step towards the crashing input. Note that for the bug we show, we found only one input that generated the corresponding crash.

After running our set of scripts that implements the previously described technique, we find that the Global Buffer Overflow is the result of 168 mutations deriving from 2 initial seeds and divided into 133 havoc and 35 splicing. More importantly, for 3 havoc mutations, the resulting testcase does not generate any novelty according to edge-coverage, while `DDFuzz` classifies it as interesting. The first of these intermediate testcases appears already after the first 10 mutations of the tree that lead to the bug. This demonstrates how our Data Dependency instrumentation can affect the findings

of the fuzzer already after few initial mutations. The other two mutations that were retained because of the DDG instrumentation come later in the tree, respectively at the 18th and 105th mutation round. This case is a good example of how the augmented sensitivity caused by our data-dependency feedback can help the fuzzer to retain input that can later help to discover new bugs.

4.4 Discussion

Overall DDFuzz experimented on three different datasets for a total of 38 different target applications. Our numbers and experiments show that embedding data flow information in coverage-guided fuzzing is a useful practice. The feedback produced by such an instrumentation can reward the fuzzer in a different way compared to current state-of-the-art techniques and lead to reaching different program points that would remain unexplored with coverage-guided approaches proposed so far. This is evident when we compare the intersections of vulnerabilities triggered by our own approach against the other instrumentations that we tested. In our custom dataset, it helped to reveal 27 new bugs compared to Edge. Moreover, in Fuzzbench, it triggered 33 different buggy locations and in our third real-world evaluation set it was able to find 12 different vulnerabilities. However, DDFuzz functioning is strictly related to the internals of the tested codebase. Indeed, as we have seen across our evaluation it is able to spot interesting and different bugs only in those cases where the tested application exhibits an high data-dependent structure of the code. Therefore we tried to develop a reliable heuristic (that we referred as `DD ratio`) that helps at recognizing these cases before running the actual fuzzing session. Moreover, we tested the impact of our technique from the point of view of the growth in the fuzzer queue, demonstrating that in the average case the increase is minor than x2, while in the worst case, it is smaller than many coverage guided approaches. Finally, the code coverage reached by DDFuzz is affected negatively only in a minimal part (-1.4% in the worst case against Ctx) while in some cases, it can allow to trigger different program points. All these aspects come at the expense of a moderate slowdown of 10-14% in the average case (6-8% if we use the geometric mean) compared to traditional Edge.

In total our dataset includes 38 programs, therefore respecting the fuzzing guidelines indicated by the authors of [92]. While we cannot make sure to cover all possible scenarios we thing that the targets we selected for our experiments are quite representative from different points of view. For instance, they provide a good variety of weakly and highly data-dependent

applications as well as they show different trends in terms of performances and discovered bugs. In particular, we found that the highest amount of data-dependency present in an app corresponds, in our set, to 23% (`c2m`) that resulted in a slowdown of 21%. This hints that in a worst-case scenario with higher values of the DD ratio, our instrumentation could penalize the fuzzing session by injecting too much instrumentation. However, during our research of the targets, especially w.r.t. the first custom dataset, we did not meet any application producing a DD ratio so high to hinder the fuzzing process.

That being said, our approach is first of all a sub-approximation of the path coverage that allows the fuzzer to reach new program points. As we described in the Introduction, path coverage is not a good solution for many applications where it results in state explosion. On the other hand, the fact itself that DDFuzz is an under-approximation in part justifies why our approach can only work in a subset of cases. However, we believe that relying on such approximations to produce alternative feedbacks could, and should, represent a possible road rather than just focusing on edge coverage based instrumentations and we hope, with our work, to put more emphasis on this aspect for future research.

4.5 Limitations and Future Work

Although we believe that our implementation well describes the potential of the data flow information as feedback for fuzzing, there are some points that we did not investigate, and that could additionally extend and improve our approach for future uses.

Firstly, our approach is useful particularly for what we called strongly data-dependent applications. This is at the same time a limitation and a feature, because it restricts the range of programs for which our fuzzing approach should be deployed.

Another point of improvement is that our current implementation does not avoid edge collisions. This could result in some paths that are ignored as the result of the `XOR` computation returns the same value for two different sets of edges. Note that this is similar to what happens with the Ngram instrumentation that solves the collisions problem only in part, introducing a more sensitive feedback that results in more collisions in the bitmap. Since the scope of our study was to show the efficacy of the DDG instrumentation, we did not implement a mechanism to avoid this issue. However, we plan to address this point in future work on this topic. For instance, a possible solution could consist of two bitmaps, one collision-free

for edge coverage that implements the AFL++ approach transforming the program by breaking the critical edges in the CFG [1] and a second smaller one with collisions for DDG coverage.

For the same reason, we did not try to adapt our approach to binary-only fuzzing. Although this is technically possible, it would require a different approach to recover the DDG of the binary, and instrumentation should be injected either by binary rewriting [54] or by emulation [20]. In any case, we believe this could represent a promising future research direction, and we hope it will be considered by researchers on this topic.

Finally, our implementation is based on AFL++ and we did not adapt it to other fuzzing engines. Given the large number of fuzzers, it is possible that our approach could work in a different way depending on the underlying implementation of the engine.

Chapter 5

Understanding American Fuzzy Lop

Recent research in software vulnerability discovery has identified *fuzzing*, or *fuzz testing*, as a key technology to efficiently detect bugs in different types of applications, including classical user-space programs [111, 116], OS kernels [161, 146, 159] and virtual machine hypervisors [145].

The high demand for more and more advanced fuzzers has resulted in a large proliferation of new prototype implementations. Some of these solutions have become well-known and largely adopted tools. Others have contributed to the research process, by studying new ideas that help fuzzers to uncover new vulnerabilities faster or with higher precision. Although every new tool comes with new features that distinguish it from existing fuzzers, a considerable amount of the functionalities is usually inherited from its "parent" project, which is often a well-established tool in the community.

Over the past five years, both industrial and academic research on fuzz testing has reached a consensus on a de-facto standard for fuzzing – the American Fuzzy Lop (AFL) [175] released in 2013 by Michał Zalewski. Two main aspects can explain AFL's success. On the one hand, its usability allows researchers to run the fuzzer out-of-the-box against several programs without any specific domain knowledge of the target itself. On the other hand, AFL excels at finding vulnerabilities fully automated, with low manual effort for security analysts. While these two factors are essential to explain the large success of this project, its development process passed through many phases of implementation and optimization. Often, new features are developed by multiple external contributors, with the inherent consequence that many design choices are not documented in a single and accessible resource.

This chpater provides an accurate analysis of internal mechanisms, parameters, and algorithms, that determine the final behavior of the American Fuzzy Lop. In other words, we shed light on the design choices that have been implemented over the years and on their impact. In many cases, improvements came from contributions outside the academic ecosystem, thus lacking experiments and clear results to demonstrate why the author chose a specific technique over alternative options. As a result, today, everybody uses AFL without a complete understanding of its internals. However, we found that even minor modifications of the inner parameters affect the results of a fuzzing experiment, both positively and negatively.

More importantly, this lack of documentation prevents researchers from identifying, in a rigorous way, what the root causes behind the excellent performance of AFL are. We believe that this deep understanding is a fundamental step to guide future work in the field.

It is also important to understand that not all design choices are related to the effectiveness of the vulnerability discovery process. Some may instead improve other aspects of the fuzzing workflow, such as usability and reproducibility of results. In this chapter, we also study whether these features are still beneficial in modern fuzzing campaigns or if they should now be considered outdated.

Our work's primary focus is on the algorithmic components that AFL embeds and that we can still find in other modern fuzzers, like its scheduler, the mutation engine, and the feedback mechanism. We exclude other specific engineering decisions, such as AFL's original solution to scale over multiple cores and machines. To verify the impact of each component, we performed a dedicated set of experiments in which we compare the vanilla AFL solution with a carefully-designed patched version of the project that replaces the feature under analysis. For instance, one of the mechanisms that captured our interest since the beginning was AFL use of *hitcounts* to encode the feedback in the coverage map. To study this aspect we patched AFL to include an alternative approach to measure the coverage, namely *plain edge coverage*.

Overall, we identify *nine* unique aspects that represent, to the best of our knowledge, the core design choices of modern fuzzers. We independently evaluate each feature and its patched counterpart(s), through a set of experiments performed on the popular FuzzBench benchmarking service [116]. We mainly used the bug-based dataset but also included the coverage-based one to clarify some cases where only one dataset was insufficient to draw conclusions. This allowed us to study each aspect in terms of its direct effects on the fuzzing campaign, as captured by the number of bugs and

increased coverage. While these are the two metrics that researchers have settled upon to evaluate the overall performances of a fuzzer, in this thesis we argue that these two values are often insufficient to gain insights about the impact of a certain feature or internal parameter. In fact, in our experiments, we often found that these two metrics alone did not provide enough information to fully capture the subtle difference between different implementations, which would allow security researchers and fuzzers' developers to debug and fine-tune their tools.

Therefore, for some of our final remarks, we limit our takeaways to qualitative findings, just relying on what we can learn by looking at general metrics such as bugs and coverage, and losing the necessary precision to measure deeper consequences of using a particular technique.

Overall, we can split our findings into two main groups. The results in the first group show that some features commonly adopted by off-the-shelf fuzzers root their origin in pragmatical or historical reasons, rather than scientific ones. For instance, we observed that a simple *random* energy assignment policy is capable in many cases to outperform the default AFL's energy assignment scheme. Similarly, we found that splicing implemented as a stage was less effective than splicing as a mutation, even though this comes with the caveat that the generated testcases are possibly more complicated to debug, thus affecting the usability of the system.

The second set of findings confirms the effectiveness of some historical design attributes when compared to modern alternatives. This is the case for novelty search, one of the major, and often forgotten contributions of AFL. It regularly outperforms the use of other genetic algorithms in terms of discovered bugs.

To conclude, we believe that the main contribution of this chapter is to show how even apparently minor aspects can impact, both positively or negatively, the performance and outcome of a fuzzing campaign. We hope that our evaluation can pave the way for more sound and complete comparisons so that security practitioners and researchers can refine their tools to obtain the best results from their efforts. Therefore, in the spirit of open science, we release all code and artifacts to reproduce the evaluations in this chapter at

https://github.com/eurecom-s3/dissecting_afl

5.1 American Fuzzy Lop

American Fuzzy Lop is a mutational coverage-guided fuzzer with a suite of additional tools [175]. These include testcase- and corpus-minimizers, a fault-triggering allocator, and a file format analyzer. Its latest available version at the time of writing is the 2.57b [1], released in 2020, but the fuzzer is unmaintained by its original author since 2.52b [2], released in 2017.

In this section, we will discuss the inner working of the fuzzer, `afl-fuzz`, and the design choices behind it.

5.1.1 General Design

As stated by Zalewski in a technical whitepaper [180] written in 2016, the main design principles behind AFL are *speed*, *reliability*, and *ease of use*. While important, these metrics are no longer the predominant principles that drive recent research on fuzz testing. Instead, researchers now predominantly focus on the time required to uncover bugs and on the amount of coverage reached. While some choices in AFL improve these two metrics, the principle of ease of use is often forgotten, even though it is the reason behind many aspects of AFL. For instance, the corpus is represented as a queue for ease of use: by making AFL mutate simpler testcases first, shallow crashing testcases will have only minor changes over the original, "human-friendly" testcases. In addition, the fuzzer keeps track of the parent testcases of each corpus entry, allowing the user to reconstruct the genealogy of each corpus entry or crashing testcase.

The actions of the fuzzer are divided into *stages* that correspond to several tasks applied on a single testcase taken from the queue. Users may configure the behavior of these stages in different ways, for instance by disabling the deterministic stage with the `-d` parameter [182]. The testcase delivery to the target program is performed via standard input or through a file. Finally, the target execution is controlled by using a *forkserver* [177], a mechanism that uses pipes to request copy-on-write clones of the target programs with `fork(2)` for each execution to avoid the overhead of `execve(2)`.

5.1.2 Coverage Feedback

The main difference between AFL and previous solutions is the code coverage of the target program used as feedback. Although not the first to

[1] https://github.com/google/AFL/releases/tag/v2.57b
[2] https://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz

introduce this approach [157, 49], AFL took coverage guidance to the next level with an effective evolutionary algorithm based on this feedback.

However, the coverage metric AFL uses is not a classic path coverage. In fact, like many symbolic executors [19], AFL aims at a trade-off between precision and path explosion. Therefore, instead of simple basic block coverage, it uses edge coverage augmented with counters (*hitcounts*) to track the number of times an edge was executed. According to Zalewski [180], the use of hitcount buckets allows AFL to effectively tackle the path explosion problem.

Implementation-wise, AFL keeps a shared bitmap between the target and the fuzzer of 64kb (a value chosen to match the L2 cache size at the time AFL was first developed) with each entry of one byte. When an edge is executed, the corresponding entry is incremented by 1, wrapping around the byte in case of overflow. The instrumentation is at the level of basic blocks, so the ID used for each edge is the result of a hash function that combines the current block with the previous. This approach introduces collisions in the bitmap. Starting from version 2.37b (released in 2017), AFL adopted the `trace-pc-guard` option of SanitizerCoverage [102] for source-based instrumentation, an approximation of edge coverage that uses precise block coverage after breaking critical edges. After each traced execution, AFL post-processes the map and buckets the entries, thus reducing the possible values from 256 to 9. This mechanism is at the core of many fuzzers derived from AFL, such as AFL++ [65] and LibFuzzer [105].

This coverage information is used in the fuzzer by different algorithms. Its most important use is to decide if a testcase is *interesting*, and, therefore, whether it is worth adding it to the corpus for future mutations. For this, AFL uses a *novelty search* algorithm that considers as interesting an input that uncovers a new entry in the map or a value that reaches a previously unseen bucket.

The use of hitcounts allows AFL to encode each possible bucket as a bit in a single byte. Thanks to this optimization, AFL implements a very fast novelty search by using only a loop of DWORD/QWORD bit-wise operations. The choice of using 8 buckets allows AFL to avoid a path explosion, and, at the same time, increases execution speed, as it allows for a highly optimized processing of the resulting coverage map.

5.1.3 Scheduling

Like many other fuzzers, AFL makes use of multiple scheduling policies for various components.

First, it schedules which testcase in the corpus should be selected next. As described before, the corpus is represented as a queue and the base policy is FIFO. On top of that, AFL uses heuristics to decide to skip a testcase for various reasons. The first applies when there are some *favored* testcases in the corpus. The fuzzer marks a subset of the corpus as favored in the process of re-evaluating the queue and choosing a small subset of testcases that cover all the coverage seen so far, the so-called *corpus culling*. The main purpose of this operation is to give priority to testcases that are smaller and faster to execute. If there is at least one corpus entry in the favored set, a non-favored testcase is skipped with a 99% probability. Otherwise, the probability goes down to 95% in the case of a non-favored, but previously fuzzed entry, and 75% for never selected cases.

Another scheduling application is the so-called *energy assignment* [27]. For each corpus entry, AFL calculates a score that is used to compute how many executions must be performed in each stage in which mutator is used. The policy employed in the fuzzer, implemented in the `calculate_score` routine, is based on several parameters. The first is the execution time of the testcase, which can alter the score if slower, or faster than the global average from 0.1x up to 3x. Another parameter is the number of filled entries in the coverage map when executing the testcase, this time applying a multiplier from 0.25x to 3x. The intuition is that testcases with greater coverage trigger more interesting states. Additionally, the score is increased for newly discovered entries to allow the fuzzer to focus on novelties. Following the same spirit, the *depth* of the entry in the genealogical tree is taken into account as a multiplier to fuzz derived inputs, that could have been difficult to discover by blackbox approaches, for a longer time.

5.1.4 Mutators

AFL relies on generic, target-agnostic, byte-level mutators [176]. These are used in several stages, many of which are deterministic. The fuzzer sequentially bitflips the current input starting from one to 32 bits at a time. During this process, as optimization, AFL records the bits that do not contribute to a change in coverage to avoid mutating them in subsequent deterministic stages. After that, the fuzzer walks each byte by adding and subtracting integers in the range from -35 to +35. The next stage is the replacement of each part of the input with numbers from a set of interesting values, such as INT_MAX, 0, and 1. This is done iteratively on the input first at the byte level, and then by using 16 and 32 bits integers.

The last of the deterministic stages uses a dictionary [179] of tokens related to the input format, for instance, `\x7fELF` if the target is an ELF

parser. These tokens can be specified by the user (with the `-x` parameter) to help the fuzzer to generate testcases that are otherwise impossible to create by using generic bit-level mutations. AFL can also auto-detect tokens during the bit flips stage by looking for groups of bits that, when changed, always produce the same coverage, a sign that they might be part of a magic value. The dictionary stages then mutates the testcases, replacing and inserting tokens from both, the user-specified and the generated list.

The first non-deterministic mutation stage is *random havoc*. It applies several mutations, including the ones used during the previous stages and some block-based mutations such as overwriting and inserting blocks of inputs. The mutations are applied at random locations of the input and are stacked. The number of applied mutations is chosen at random between 2 and 128 and the iteration of the stage is regulated by using the score of the testcase.

The last stage, *splicing*, by default is activated only after the fuzzer goes through a full cycle of the entire queue without any new finding (but it is always enabled in FidgetyAFL [182]). It selects an entry from the corpus and recombines it with the current testcase, then it applies the havoc mutator to this child testcase. This is an important stage that allows AFL to generate testcases derived from two parents.

In AFL, for ease of use, each testcase saved in the corpus or the crash folder keeps the information about its one or two parent testcases, as well as the mutations that were applied. This allows AFL's users to reconstruct the entire process of derivation of a testcase, information that helps them during crash analysis.

5.1.5 Minimization

Some mutations can increase the size of a testcase and, especially for inputs discovered later in a testing campaign, can result in very lage files. These large, slow-to-parse, inputs can decrease execution speed and decrease the likelyhood of a mutation of the correct bytes. Therefore, the fuzzer tries to minimize their impact by using a testcase minimization algorithm.

After requesting a testcase from the corpus, AFL passes it through its *trimming* stage. The key idea is to mutate the testcase by trying to obtain a smaller testcase that still achieves the very same coverage. The algorithm consists of removing blocks from the inputs while checking if the coverage map remains the same. If successful, the process is repeated several times by increasing the size of the blocks to remove. This technique reduces the complexity of the items in the corpus, but it also requires additional executions for each testcase that is saved in the queue.

5.1.6 Instrumentation

To obtain the coverage information from each execution of the target, AFL employs several instrumentation options. First of all, it can instrument the x86 compiler to intercept and modify the assembly code, to log each basic block by relying on functions available through an injected runtime. In addition, AFL also provides an LLVM pass [95] which assigns a random block ID at compile time and adds the instrumentation to hash the blocks and write to the shared memory, thus resulting in a more efficient instrumentation than the one provided by the legacy x86-only solution. With LLVM, the runtime is also more mature as it provides not only the forkserver option but also the so-called *persistent mode* to avoid forking when fuzzing stateless code, resulting in increased performance.

Alongside compiler-based approaches, AFL comes with a binary-only QEMU mode. QEMU mode uses a patched QEMU 2.10 usermode to inject a forkserver at the guest entrypoint, and to add instrumentation between each executed basic block, through a logger routine executed after each basic block.

5.2 Methodology and Experiments Design

By reviewing the implementation and the internals of AFL, we identified nine characteristics to assess in our tests. For each of them, we also looked for alternative solutions proposed in other works to serve as a comparison in our experiments. We have not selected the trivial comparison between AFL and FidgetyAFL [182] as it is covered in the FuzzBench paper [116], which highlights that FidgetyAFL always outperforms AFL in terms of code coverage over time.

Our aim is to assess the contribution of each feature on the performance of AFL in terms of uncovered bugs and code coverage using FuzzBench [116] over a 23h campaign. If the results depend highly on the structure of the target program, we will try to classify manually which kind of program is influenced by the tested feature. Finally, when the results do not show a significant difference, we will provide a qualitative investigation of the possible impact of that feature on usability.

We now introduce the nine aspects to be covered in our study.

Hitcounts. Hitcounts are adopted by other fuzzers today [105, 158], but AFL was the first to introduce this concept. Despite its wide adoption, the impact of this optimization (over plain edge coverage) has never been measured in isolation on a large set of targets.

To fill this gap, we modified AFL not to increase each entry in the coverage map while instrumenting the target. Instead, we always set the value to 1. We expect hitcounts to improve the coverage and especially the bug detection capabilities by introducing additional information about the program state, like loop counts. We want to quantify this improvement and potentially discover target-specific corner cases.

Novelty search vs. maximization of fitness. While AFL considers every newly discovered hitcount as interesting, both, other early fuzzing solutions [181], and more recent tools [138] instead only consider testcases that maximize a given metric as interesting. For instance, VUZZER uses the sum of all the weights of the executed basic blocks [138].

We think that a big part of the success of AFL in terms of performance is the novelty search-based approach it uses to evaluate interesting testcases. To evaluate this assumption, we implemented [3] a simplified version of the VUZZER fitness maximization without the need for static analysis, in which each basic block has weight 1:

$$f(i) = |\mathrm{BB}(i)| \begin{cases} \frac{\sum_{b \in \mathrm{BB}(i)} \log_2(\mathrm{freq}(b))}{\log_2(\mathrm{len}(i))} & \text{if } \mathrm{len}(i) > 50000 \\ \sum_{b \in \mathrm{BB}(i)} \log_2(\mathrm{freq}(b)) & \text{otherwise} \end{cases}$$

We chose to borrow the VUZZER fitness function as it is a simple one based on just code coverage, avoiding introducing a fitness from scratch as, to the best of our knowledge, VUZZER is the only academic work proposing a simple fitness. Other approaches in the literature that use a fitness employ heavy static analysis or complex approaches based on many features, not just code coverage [115]. While it would be interesting to benchmark them too, it is not fair to compare such complex techniques with a fuzzer that only uses code coverage like AFL. More complex novelty search solutions are present in literature [166] that can be used as a competing approach in future works.

In this experiment, we benchmark the AFL approach versus a fitness maximization and the combination of the two approaches, as proposed by VUZZER [138]. We expect novelty search to outperform both of the competing algorithms, as the maximization saves testcases in the corpus that are not small and fast (one of the key elements in the design of AFL), and might end up in local maxima. A set of diverse testcases, like the ones saved

[3] Note that the input length is bound to 50,000 bytes (to address input bloating) and the log base is taken from the VUZZER code.

by AFL, is likely better in the corpus during fuzzing.

Corpus culling. The prioritization of small and fast testcases in the AFL corpus selection algorithm improves the speed at the cost of avoiding more complex testcases that might trigger more complex program states. We selected this feature for our benchmark because the set of *favored* testcases in AFL was a major addition to the fuzzing algorithm, and it is used even as a metric in following works such as Driller [156].

In this experiment, we want to assess the difference between using the AFL corpus culling mechanism and using the entire corpus. We expect culling to result in faster coverage growth over time and, potentially, more bugs triggered in the same time window. On the other hand, the fuzzer without corpus culling might be able to discover new bugs that standard AFL is unable to trigger.

Score calculation. The performance score used to calculate how many times to mutate and execute the input in the havoc and splice stages are derived from many variables, mainly testcase size and execution time. This score is an essential part of AFL and the focus on many derived works (e.g. [27, 173, 25]).

In our experiment, we measure the difference between the AFL solution and two baselines, represented by a constant and by random scores. As picking a constant is a sensitive operation, we opted to create two AFL variants, one with the minimum score possible for AFL (25) and another with the maximum (1600). The random variant selects instead a random number within these boundaries. In addition, we include in the experiment a version of `calculate_score` that does not prioritize novel corpus entries, as this was a significant optimization introduced in AFL. Intuitively, we expect that the major contribution comes from the prioritization of the novelties, thus resulting in small differences between the baselines and the patched AFL with the naive score calculation.

Corpus scheduling. The FIFO policy used by AFL is only one of the possible policies that a fuzzer can adopt, to select the next testcase. However, derived works tend to take for granted that the corpus structure is represented by a queue.

While we know that this feature has its root in usability, in this experiment we assess whether it also contributes to the performance of the fuzzer. Thus, we evaluate AFL versus a modified version that implements the baseline (i.e., random selection) and the opposite approach (i.e., a LIFO scheduler). We expect the random approach to perform equal to, or even better than the original embodiment of AFL, while the LIFO approach may

help in gaining coverage faster on some targets.

Splicing as stage vs. splicing as mutation. Splicing refers to the operation that merges two different testcases into a new one. There are two possible ways to apply this mechanism. The first, adopted by AFL, considers splicing as a stage. In this case, the actual merge happens only once at some point in the execution of a specific testcase, when it is joined with a randomly chosen input among the other ones present in the queue. However, other fuzzers (e.g., Libfuzzer [105]) often implement splicing as a mutation rather than a stage, thus applying it many more times for each testcase during their havoc stage.

To compare the two solutions, we modified the AFL codebase to implement splicing as a mutation operator. This choice can also have an impact on the usability of the fuzzer. Indeed, we expect that a major adoption of splicing as mutation can increase the exploration of the fuzzer while reducing the simplicity of the testcases and, therefore, complicating the a-posteriori triaging phase.

Trimming. Trimming testcases allows the fuzzer to reduce their size and consequently give priority to small inputs, under the assumption that large inputs slow down the execution and that the mutations would be less likely to modify an important portion of the binary structure. In AFL, the component in charge of this task tries to discard blocks of data with variable length and stepover. When the removal results in the same checksum of the original trace map, the new minimized testcase is stored.

Even though this algorithm can bring the two important benefits described above, we argue that reducing the size of the testcases could reduce state coverage. Additionally, the trimming phase could become a bottleneck for slow targets. Therefore, in our evaluation, we compare the default version of AFL against a modified one, in which we disabled trimming. We expect trimming to be either beneficial or detrimental, depending on the type of target program and the structure of its input.

Timeout. The timeout regulates the maximum amount of time the target program runs for. This greatly influences the execution time of the target and in turn the number of executions per second. While the user can specify an arbitrary value by passing a command line argument (`-t`), AFL can also automatically compute a timeout for the program under test. More specifically, as a first step, AFL calibrates the execution speed during an initial phase by running the target several times and computing an average of the execution times. After that, the default heuristic applies a constant factor (x5) to this average value and rounds it up to 20 ms. In our experiments,

we modify the multiplicative factor to measure its effect on the fuzzing session. We expect that a higher timeout can lead to higher coverage, but also degrade the performance of the fuzzer.

Collisions. As explained in section 5.1.6, AFL assigns an identifier for each basic block at compile-time. When using SanitizerCoverage [102]'s *pcguard*, critical edges are split into basic blocks and thus AFL assigns a random identifier to each edge. Unlike the classic instrumentation that combines the IDs of the current and the previous block, however, this technique is unable to track edges related to indirect jumps. For both variants, since identifiers are chosen at random, this causes collisions between two different edges in the bitmap, that in turn can affect the novelty of a testcase. Although the number of collisions depends on the number of instrumented locations, for an average size program the actual collisions are typically between 750 and 18,000 [86].

In our evaluation, we want to compare the AFL instrumentation approach against a version that is collision-free. As SanitizerCoverage traces each block by calling a function with a guard parameter, and this guard is contained in a per-module table initialized in a constructor, we can easily patch AFL to assign values to the guards by using a global incremental counter in the constructor instead of random values. This allows the instrumentation to generate edge encodings that do not result in collisions during the fuzzing session. Other fuzzers indeed make use of the guard variable as the index to access the fuzzer bitmap.

We want to benchmark this feature as the collision-free variant is simpler than the original implementation with *pcguard*, raising the question of why random identifiers are even used in AFL. In addition, it is unclear if the lack of feedback from the indirect jumps affects the performance more than the collisions, so we include in our test also the classic approach to benchmark this impact.

Please note that in this experiment, unlike the collision-free coverage based on *pcguard* present in AFL++ (since 2.66c), we do not adapt the size of the map to the detected number of blocks – a feature that significantly improves the speed of the fuzzer – as we want to evaluate the impact of the collisions in isolation.

5.3 Experiments

In this section, we present the results of our experiments, conducted by using the FuzzBench service [116], and we discuss them to understand the

Table 5.1: Hitcounts vs. plain edge coverage bug-based experiment score

Fuzzer	Average normalized score
AFL edge coverage	88.09
AFL	74.36

impact of each selected feature. We mainly use the bug benchmark of FuzzBench, which consists of 25 targets known to contain bugs, as we believe that discovered bugs are the ultimate metric in fuzzing evaluation [92]. In addition, we also report the coverage over time as another important metric to understand the performance of each variant of AFL. Each program was executed for 23 hours and the reported results are median values computed over 20 trials to mitigate the effects of randomness in fuzzing. In addition, we use the Mann-Whitney U test to verify the statistical significance of the results by comparing differences between two independent groups that in our case are the original AFL and its variants. The aggregation of the results is done by using an average normalized score [116]. Finally, we executed all variants with the `trace-pc-guard` instrumentation and persistent mode to mitigate the well-known impact [168] of `fork(2)`.

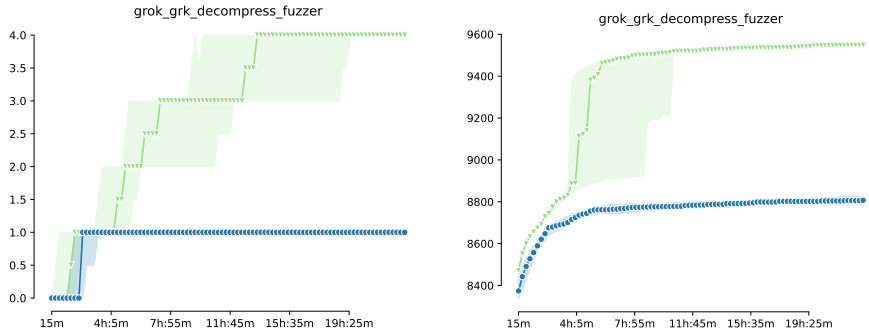
For the sake of brevity, we only report the results of interesting benchmarks and avoid discussing each individual benchmark for each experiment. For the interested reader, the graphs with the complete data of all the 9 experiments are available online at https://anon-afl.github.io/dissecting_-afl_reports/.

For each set of experiments, we also highlight in gray our discovered insights. We hope this can help users to better understand AFL and improve the design of new fuzzing approaches.

5.3.1 Hitcounts

In this first set of experiments, we compare vanilla AFL against a modified version that does not use hitcounts. Table 5.1 reports the average normalized score of the number of uncovered bugs in our experiments. Quite surprisingly, the AFL variant without hitcounts discovered more bugs than the unmodified AFL, a counter-intuitive result as hitcounts should allow AFL to bypass coverage roadblocks that depend on loop counts.

In particular, vanilla AFL performed better on 6/25 benchmarks in terms of median discovered bugs, out of which only two are statistically significant for the Mann-Whitney U test. The variant with only edge coverage was better on 5/25 benchmarks, of which four are statistically significant.



(a) Bug Detection (b) Coverage

Figure 5.1: Comparison of AFL and AFL-edge-coverage on Grok grk decompress (■ `AFL`, ■ `AFL edge coverage`)

It is interesting to note how for some targets edge coverage clearly outperformed vanilla AFL, as in the case of the grok and the PHP benchmarks. For instance, in the case of `grok_grk_decompress_fuzzer` we can observe that the graphs reporting bugs discovered over time and coverage over time (Figure 5.1) are correlated. This might suggest that the use of hitcounts prevents the fuzzer from discovering new code paths, a behavior that can be explained by the augmented sensitivity, up to 8x as the hitcounts introduce 8 different states for each edge.

As shown by previous studies [162, 163, 62], the increase of sensitivity introduces testcases in the saved corpus that are too similar to one another, causing internal wastage of the exploration of the program. AFL is therefore focusing on fuzzing testcases that are not frontiers in terms of unexplored coverage areas. This behavior is, of course, highly target dependent, as the states that AFL can reach by using the hitcounts in its feedback may contain bugs that otherwise cannot be easily discovered with edge coverage only.

Table 5.2: Hitcounts vs. plain edge coverage code coverage-based experiment score

Fuzzer	**Average normalized score**
`AFL`	99.63
`AFL edge coverage`	97.99

To further confirm our intuition that hitcounts introduce a benefit only

Table 5.3: Novelty search vs. maximization of a fitness bug-based experiment score

Fuzzer	Average normalized score
AFL	83.32
AFL fitness	83.08
AFL fitness only	70.17

on some targets, we run another set of experiments on FuzzBench on a different set of 22 benchmarks that FuzzBench uses to evaluate fuzzers using only code coverage as a metric[4]. The score reported in Table 5.2 shows that on this set of different subjects classic AFL outperforms the variant with only edge coverage, confirming that hitcounts can either increase or decrease the effectiveness of the fuzzer depending on the target application.

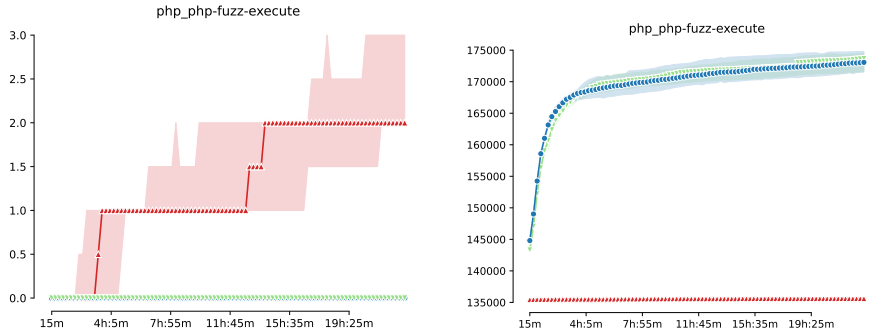
Our conclusion after this experiment is that AFL, and follow-ups fuzzers like AFL++, should provide an option to disable hitcounts. AFL++ provides many different options, and the users are suggested to run an instance of each variant when doing parallel fuzzing, a common use-case in real-world setups. The fact that in our experiments, hitcounts have shown very different results on different targets suggests that users should include a variant without hitcounts when doing parallel or ensemble fuzzing like OSS-Fuzz [3].

5.3.2 Novelty search vs. maximization of a fitness

In this second experiment, we compare three fuzzers: vanilla AFL (with its novelty search algorithm), a variant with only fitness maximization, and a hybrid variant with both maximization and novelty search. In line with our expectations, the bug-based benchmark shows that, in average, vanilla AFL performs best. Table 5.3 reports the average normalized score of the number of uncovered bugs.

The usage of the fitness only is clearly detrimental and the combination of both techniques does not introduce a valuable increment in bug-discovery. AFL and the combined variant perform almost the same, with the exception of `libhtp_fuzz_htp` in which the fitness variant is better and `poppler_pdf_fuzzer`, in which vanilla AFL is best. While this result

[4] https://www.fuzzbench.com/reports/experimental/2021-12-17-afl-edges/index.html



(a) Bug Detection (b) Coverage

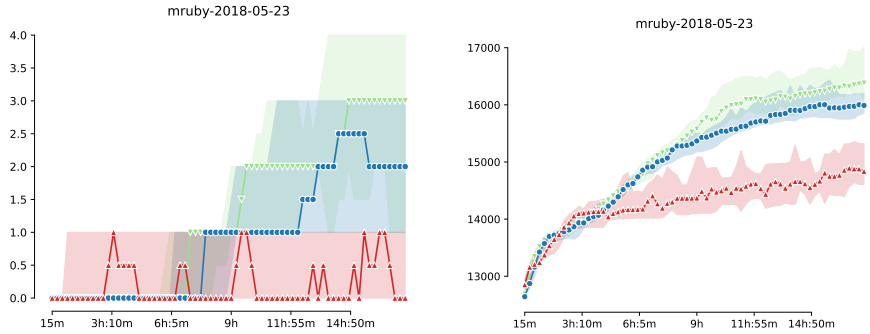
Figure 5.2: Novelty search vs. fitness experiment on the PHP application (■ `AFL`, ■ `AFL + fitness`, ■ `AFL fitness only`)

was expected, there are some surprising results on specific targets such as `php_php-fuzz-execute`, placing the variant with only the fitness maximization as the best fuzzer on 4/25 benchmarks, all statistically significant.

Unlike in the previous experiment, this time there is also no correlation between uncovered code and bugs. For instance, Figure 5.2 shows that for PHP the variants with fitness only are unable to increase the coverage of the target application, but at the same time, it is the only variant able to discover bugs. The saved testcases in the corpus cover the same regions as the initial testcases so we can observe that, on this target, the fuzzer is behaving like a blackbox fuzzer without any coverage tracking capability.

A possible explanation is that the novelty search fuzzers are spending time exploring more program behaviors, while the bugs are in the initial code regions behind constraints that cannot be solved immediately. On large programs, this is a well-known behaviour [30] which explains why random testing can outperform more complex solutions.

The conclusion we can draw from this experiment is that it would be a mistake to underestimate the impact of the novelty search. In particular, researchers proposing new approaches that also modify this aspect should carefully evaluate – in isolation – the benefit of a different mechanism to decide if an input is interesting, as AFL's novelty search provides a strong baseline.



(a) Bug Detection (b) Code Coverage

Figure 5.3: Corpus culling comparison on the mruby application (■ `AFL`, ■ `AFL w/o fav_factor`, ■ `AFL no culling`)

5.3.3 Corpus culling

In this third experiment, we evaluate AFL versus two other variants, one without corpus culling and one with culling without any prioritization based on the `fav_factor` (a function of execution speed and testcase size). In Table 5.4 we report the average normalized score for each fuzzer in terms of discovered bugs.

Table 5.4: Corpus culling experiment score

Fuzzer	Average normalized score
AFL w/o fav_factor	90.14
AFL	87.00
AFL no culling	81.94

The usage of corpus culling is clearly a benefit looking at the total numbers, but the prioritization given by using `fav_factor` as weight is not. The reason behind this is that corpus culling favors faster testcases while maintaining the same code coverage, but loses the state triggered by more complex testcases that cannot be easily observed by looking at edge coverage alone. While this optimization is effective as it reduces the number of testcases in the queue, prioritizing always the smaller and faster inputs may be decremental in findings bugs. In fact, our experiments show that the more naive version of culling seems to be the most effective in practice.

Taking `mruby-2018-05-23` as case study (Figure 5.3), the variant without the fav_factor provide the best results. The graphs show how the ex-

ploration of the program states not related to code coverage can help the fuzzer to increase coverage faster. While this may seem counter-intuitive, there are programs states blind to edge coverage (e.g. loop counter values) that are roadblocks in terms of exploration of the control flow graph. Therefore, fuzzing more complex testcases can help to bypass them and achieve new coverage. While this variant is generally better, the results are only statistically significant for the `mruby` and the `openh264_decoder_fuzzer` applications.

Investigating the results of the variant without culling also provides interesting insights. The outcome is highly variable, with benchmarks like `php_php-fuzz-execute` and `poppler_pdf_fuzzer` in which it discover more bugs and more code coverage in a statistically significant way, and others like `grok_grk_decompress_fuzzer` in which it is only able to discover two bugs while the others discover six. This can be explained by looking at the number of testcases in the queue, as the fuzzer got stuck fuzzing testcases too similar to one another if culling is disabled.

Our experiments show that complex testcases are useful to uncover bugs and AFL should not discard them a priori. However, it is unclear how to reach the right trade-off between complexity and speed and we foresee future works that try to improve the prioritization algorithm of corpus culling to fuzz faster without discarding interesting testcases.

5.3.4 Score calculation

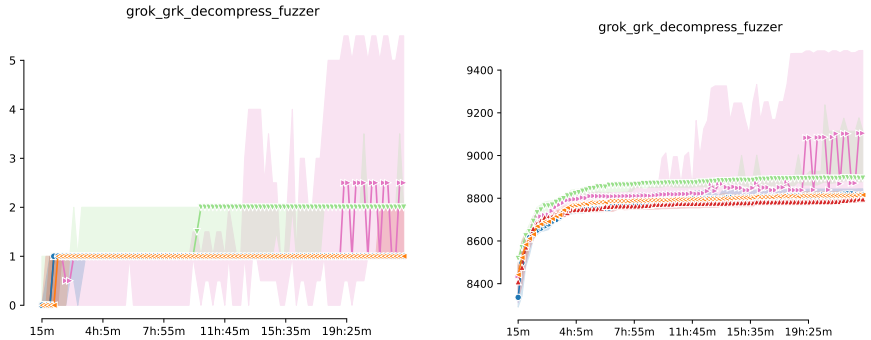
To evaluate the algorithm used in AFL to assign the energy to a testcase, we compare it versus several alternative implementations: two simply adopting a constant score (respectively the maximum and the minimum possible score in AFL), and one that assigns a random score between the valid range. Additionally, we include a variant of the AFL scoring algorithm without the multiplicative factor that prioritize the novel inputs recently saved in the queue.

In our tests, vanilla AFL was not able to outperform the random baseline in terms of an average normalized score of uncovered bugs (Table 5.5), while it was better than the variants using a constant score. The results confirm instead that the prioritization of novel testcases is an improvement.

However, we can observe that the normalized scores of AFL and the random variants are really close and the random version is the best performer only in three benchmarks (`arrow_parquet-arrow-fuzz`, `grok_grk_decompress_fuzzer`,

Table 5.5: Score calculation experiment score

Fuzzer	Average normalized score
AFL random score	93.52
AFL	90.51
AFL max score	88.88
AFL no novel	87.63
AFL min score	81.06



(a) Bug Detection (b) Code Coverage

Figure 5.4: Score computation comparison on grok (■ AFL, ■ Max, ■ Min, ■ Random, ■ No novel)

and `php_php-fuzz-execute`), out of which only the results on PHP are statistically significant.

The high variability of the random score fuzzer can be observed for instance when testing `grok`. Figure 5.4 shows that in some runs this fuzzer outperformed all other variants by a large margin (both according to bugs and coverage), but in other runs, it did not.

It is worth observing in this case that the high variability in code coverage (Figure 5.9b) is always above the curve of the other variants, suggesting that the random fuzzer consistently outperforms the others while the number of bugs is more aleatory. This highlights that uncovering a bug is more susceptible to randomness.

The result of the random variant is particularly important as in recent years energy assignment was the focus of a large number of studies, most of which used AFL as a baseline for the experiments. However, if even a random score can often perform better than the algorithm implemented in AFL, it is difficult to say whether a new energy assignment algorithm

that beats AFL is really an improvement that can increase the ability of the fuzzer to discover bugs if not compared against the real baseline. The high variance in the results of the random algorithm also suggests that it might be a useful adoption in parallel fuzzing. Multiple instances of the fuzzer using this score calculation algorithm will increase the chance to hit the best performer random distribution.

A possible threat to the validity of this experiment is the biased nature of the benchmarks, which contain applications with a medium-small sized codebase as they are libraries. With complex targets, the score calculations with a non-naive algorithm may become more important, and we can see a hint of this result by looking at the results of this experiment for the `ffmpeg_ffmpeg_demuxer_fuzzer`, a complex and slow program in which AFL triggers more median number of bugs than the random variant.

In conclusion, our experiments show that for simpler targets the energy assignment problem may be less important than for complex programs. On the one hand, this might suggest that the development effort in creating faster and more effective fuzzers can make this allocation problem less relevant for generic fuzzers. On the other hand, the fastest possible fuzzer cannot compensate for a slow and complex to execute system under test (like program interpreters or even entire operating systems), highlighting the need to benchmark new energy assignment algorithms, with a dataset of complex targets. Finally, we suggest using the baselines we introduced in this chapter to avoid the mistake of considering AFL's implementation as a baseline.

5.3.5 Corpus scheduling

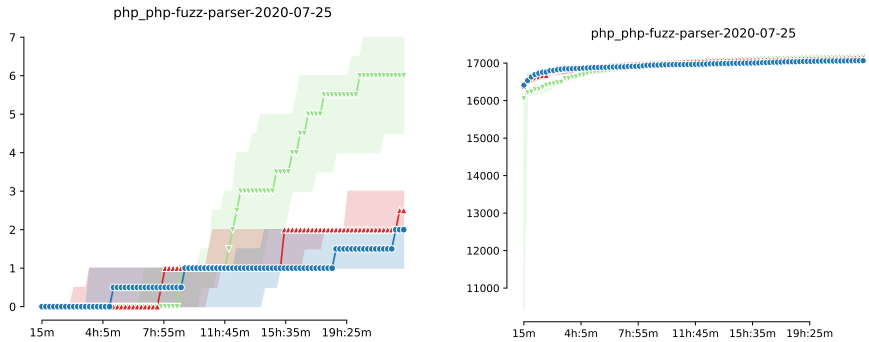
In this experiment, we tested two alternatives to the FIFO policy used in AFL to select the next testcase in the queue to fuzz: a random selection, and a LIFO policy. The results show that in terms of ability to discover bugs, vanilla AFL is better than the random baseline, but the LIFO variant is the best among the three, as reported in Table 5.6.

can we zoom the top part of the Y axis on the coverage plot

This time, the random baseline is not superior to vanilla AFL, which is better than random in 6 benchmarks, making the results of the previous works in this field, corpus scheduling or *seed scheduling*, robust even if they use AFL as baseline. However, a simple variation such as LIFO, shows a boost in performance on 7 targets, two of them in a statistically significant way.

Table 5.6: Corpus scheduling experiment score

Fuzzer	Average normalized score
AFL scheduling LIFO	90.33
AFL	82.94
AFL scheduling random	82.94



(a) Bug Detection (b) Code Coverage

Figure 5.5: PHP fuzz-parser for the corpus scheduling experiment (■ AFL, ■ LIFO, ■ Random)

For instance, on `php_php-fuzz-parser-2020-07-25` (Figure 5.5), fuzzing later discovered testcases first with FIFO gives a boost in the bug discovery ability while maintaining the uncovered coverage regions at the same level of the other AFL variants. Another benchmark that benefited from the LIFO approach is `grok_grk_decompress_fuzzer`, but in this case the performance boost is affect both bugs and code coverage.

On the other hand, when the scheduling policy decreases code coverage (like in `matio_matio_fuzzer` and `mruby-2018-05-23`), it negatively affects the results and performs worse than the vanilla AFL and the random baseline. The improvement of LIFO over AFL seems to not be related to the type of the input, for instance, it performs better on the PHP benchmarks but worst on `mruby`, both of which are textual programming languages parsers. Thus, we were not able to reach a clear conclusion on which policy is better in general, as fuzzing the newly generated testcases first is not always the best choice even if LIFO is the top performer on average.

However, the boost in performance can be easily observed once that LIFO starts diverging from AFL, making possible to learn during the fuzzing campaign which policy is best suited for a given target, even with a simple

approach that alternates between the two in the first hours and then select the best performer.

In conclusion, our experiments show that FIFO is generally better than random, not just because of usability but also because it often provides better results. However, on many targets, the alternative approach (LIFO) provided better results. Thus, we believe that more research is needed to learn the best policy for corpus scheduling (such as the recent AFL-Hier [163]). Even if the random baseline seems weaker than AFL, we believe future works on the topic should still include it as a baseline in their evaluation alongside other simple policies like FIFO and LIFO.

5.3.6 Splicing

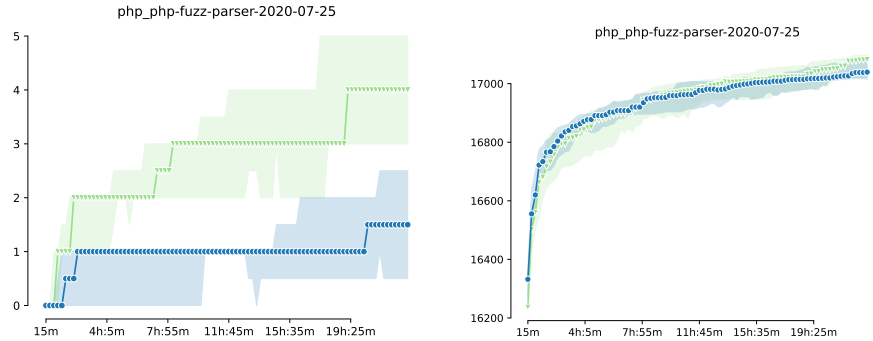
In this experiment, we evaluate AFL splicing stage (in which the currently fuzzed testcase is combined with another taken from the corpus, and then fuzzed with the havoc stage) versus a variant in which the combination of two or more testcases is implemented as one of the mutations included in the havoc stage. Table 5.7 reports the average normalized score in terms of uncovered bugs for this experiment.

Table 5.7: Corpus scheduling experiment score

Fuzzer	Average normalized score
AFL splicing mutation	97.15
AFL	94.66

The results show that the AFL variant that uses splicing as part of the havoc stage outperforms vanilla AFL. In particular, it is better in 6 benchmarks – two of which show a statistically significant difference from AFL.

In this experiment, the insight is clear, the recombination of different inputs in the fuzzer corpus leads to a benefit especially on highly structured inputs parsers such as `php_php-fuzz-parser-2020-07-25`, as shown in Figure 5.6, in which the improvement in terms of bugs finding is very large. Even on other benchmarks in which the difference in terms of bugs is smaller, like `mruby-2018-05-23`, splicing as a mutation improved code coverage. This result is not surprising, as recent works [22, 63] have already shown that an enhanced mutator that can recombine and merge different



(a) Bug Detection (b) Code Coverage

Figure 5.6: Splicing Comparison on `libxml2` (■ `AFL`, ■ `AFL Splicing Mutation`)

inputs in the corpus provides a net benefit for a fuzzer testing highly structured inputs parsers.

The downside is that frequent re-combination increases the testcase complexity, making triaging harder. In fact, by introducing splicing as mutation we lose the ability of AFL to keep track of which testcases were involved in the recombination by simply looking at the file names. Moreover, with this new variant, the number of parent nodes for each testcase can be greater than two, thus reducing the ability of users to easily keep track of the transformations applied by the fuzzer.

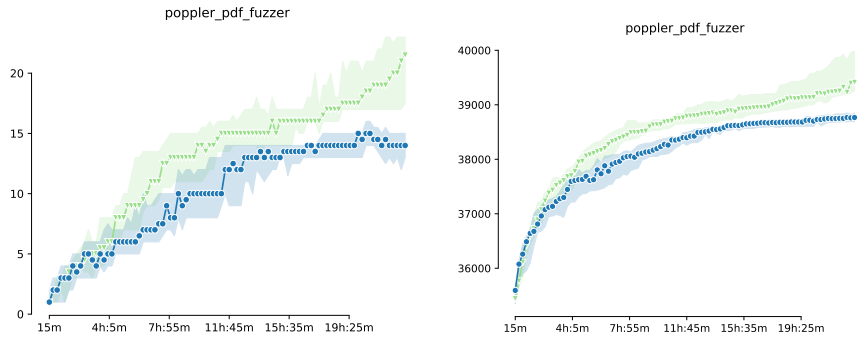
The results of this experiment confirm our intuition about the effectiveness of splicing. Modern fuzzers focus on performance as bugs are becoming harder and harder to find, and therefore we believe they should use splicing as a mutation. In the specific case of splicing, both versions can co-exist in the same fuzzer without conflicts and enabled according to user preferences.

5.3.7 Trimming

In this experiment, the comparison is between AFL (which uses trimming to reduce the size of a testcase while maintaining the same code coverage) and a variant without the trim stage. The overall result in terms of average normalized score of the bugs found is reported in Table 5.8, which highlights that the variant without trim stage outperforms vanilla AFL.

Table 5.8: Trimming experiment score

Fuzzer	Average normalized score
AFL no trim	99.25
AFL	88.81



(a) Bug Detection (b) Code Coverage

Figure 5.7: Trimming Comparison on `poppler` (■ `AFL`, ■ `AFL no trim`)

The aim of the trim stage of to speedup the fuzzer by reducing the size of testcases while maintaining intact the coverage. This can negatively affect the effectiveness of a fuzzer in the cases in which code coverage alone is not sufficient to describe the program state. So it is not surprising that on many targets the trim stage decreased the performance of AFL.

In particular, the variant without trimming was the best performer against eight targets (five of which with statistical significant results). It is worth noting that all these five programs process structured inputs and therefore the trim stage may be detrimental. In fact, attempts to shrink the testcases would most likely result in invalid inputs, and therefore AFL ends up using valuable resources to try to trim the corpus, without actually succeeding.

As an example, the variant without trimming provided great results on a complex structured input program such as `poppler_pdf_fuzzer` (Figure 5.7) in which it can discover more bugs and explore more code without reaching saturation (as suggested by the flattening graph of AFL). The application parses PDF, a complex format in which bit-level modification can easily destroy the validity of an input resulting in different coverage and so a useless (and time-consuming) trimming stage. In addition, the longer the fuzzing campaign the slower it becomes to execute testcases, making trim-

ming more and more costly without any advantage as the fuzzer is unable to alter a testcase without maintaining the same code coverage.

The insight from this experiment is that the trim stage can be useless or even detrimental when the tested codebase is large or when the target input has a complex format, as every bit-level modification will unlikely lead to a testcase that maintains the same coverage of the original. This behavior wastes resources, as the time spent trimming (without any benefit) could be better used for fuzzing. Therefore, in these cases we believe that a fuzzer without the trim stage outperforms AFL simply because it fuzz the target at a higher speed.

5.3.8 Timeouts

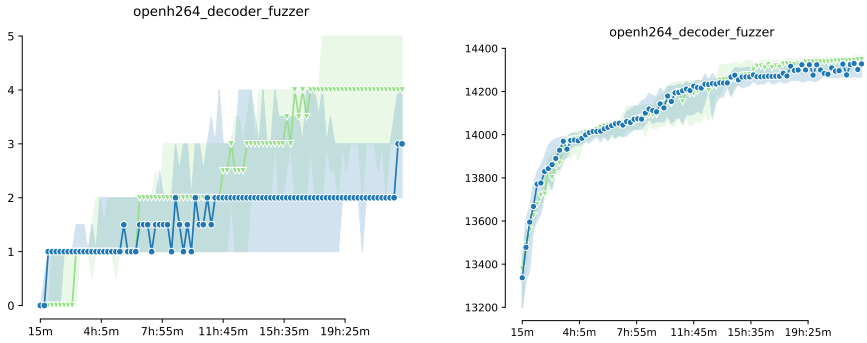
In this test we compare AFL with a variant that double the timeout chosen by the timeout detection algorithm. In terms of average normalized score, reported in Table 5.9, this variant seems to perform better than vanilla AFL.

Table 5.9: Timeout experiment score

Fuzzer	Average normalized score
AFL double timeout	97.43
AFL	94.70

The variant finds a higher median number of bugs than AFL in seven benchmarks and performs worst than AFL in two – but none of the results are statistically significant. However, we can see how on some targets, like `openh264_decoder_fuzzer` shown in Figure 5.8, doubling the computed timeout helps the fuzzer to find bugs faster. This small difference is due to the environment in which the fuzzers were run, a preemtible VM on the cloud, the default environment of the FuzzBench service. Usually, a fuzzer decreases its execution per second with time as more complex code paths are discovered, and slow targets are not an exception. A slow target like `openh264_decoder_fuzzer` on a slow machine may cause the fuzzer to generate inputs triggering timeouts virtually at every execution if the timeout is too strict.

This experiment, however, may not suggest a generally valid insight about AFL, as the speed of a target program depends on the complexity of such program but also on the method used to fuzz it. AFL in forkserver



(a) Bug Detection (b) Code Coverage

Figure 5.8: Timeout comparison on `openh64decoder` (■ `AFL`, ■ `AFL double timeout`)

mode, for instance, may behave in a different way than the setup used in this experiment (which uses persistent mode), or other alternative solutions used by AFL-based fuzzers to execute a system under test (such as full-system fuzzers [146, 85] or network fuzzers [131, 147]).

In conclusion, this experiment suggests that changing the timeout calculation algorithm of AFL with a similar one does not change much the performance of the fuzzer. This aspect is too much dependent on the platform in which the fuzzer runs to get a generic insight and the user should carefully tune the timeout based on the slowdown introduced by using different types of machines or virtual environments, such as cloud VMs or containers.

5.3.9 Collisions

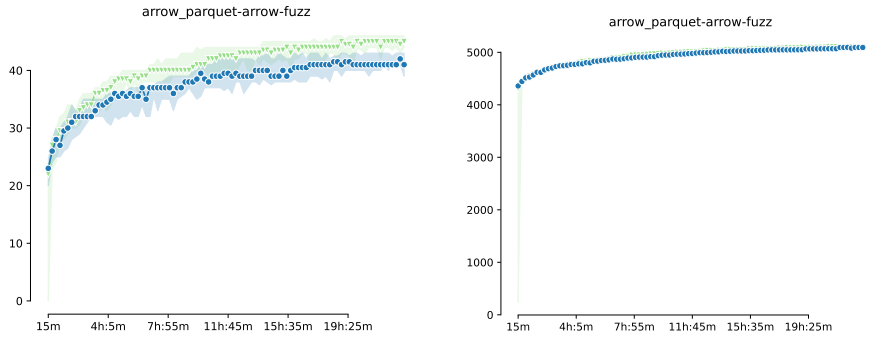
In the final experiment, we tested AFL versus a collision-free variant that uses the `trace-pc-guard` instrumentation option of LLVM to track edges in the target program. Table 5.10 shows that, overall, collisions decrease the ability of AFL to discover bugs.

Taking `arrow_parquet-arrow-fuzz` as showcase for this experiment (Figure 5.9), we can see how the collision-free variant performs better, even if by a small small number of bugs.

The collision free variant is better than vanilla AFL on six benchmarks (only two with statistically significant results) and worse on two benchmarks (none statistically significant).

Table 5.10: Collisions experiment score

Fuzzer	Average normalized score
AFL collisions free	96.52
AFL	89.44



(a) Bug Detection (b) Code Coverage

Figure 5.9: Collisions comparison on `arrow` (■ `AFL`, ■ `AFL collisions free`)

It is very interesting to note that the only two targets in which vanilla AFL performs better are large programs: `wireshark_fuzzshark_ip` and `ffmpeg_ffmpeg_demuxer_fuzzer`. In both cases, the number of edges is greater than the shared map size, thus causing collisions also in our variant. While these collisions are considerably less (as they are only the ones due to the overflow of the map, which also affects AFL), AFL uses a hash function and therefore its collisions are equally distributed across the map. Our variant uses instead sequential identifiers, and therefore all colliding blocks are located in the same area of code – basically eclipsing an entire area of the target from the fuzzer.

In AFL++, this problem is solved by increasing dynamically the map size, which however introduces a significant slowdown as the processing of the shared memory is the second most expensive operation in AFL.

This experiment shows how the simpler approach that reduces collisions by using edges enumeration is marginally better. We think that the classic hash-based index is still used in AFL as a legacy indexing scheme borrowed from `afl-gcc` and never changed. Researchers that want to build new fuzzers upon AFL should therefore consider replacing such

an indexing scheme.

5.3.10 Discussion

AFL includes multiple features whose impact has never been properly evaluated. Many of them, such as hitcounts, are commonly re-used by other fuzzers and taken for granted for historical reasons. Our work highlights the importance of benchmarking each aspect of a fuzzer in isolation to fully understand if, and when, it is beneficial for a testing campaign.

For instance, the results of our experiment show that researchers, designing new fuzzers, need to think twice before replacing certain features of AFL, such as novelty-search, with their own. On the other hand, some features, such as hitcounts, may not be beneficial for certain targets, and better options may be available. Our findings also suggest that usability-oriented features (such as using splicing as a stage, instead of as a mutator) should be carefully considered and, if they are decreasing the fuzzer effectiveness, should probably not be used by default in fuzzers.

Our results also show that AFL is a complex tool, and therefore it might not be the correct baseline to use when evaluating novel ideas. In reality simpler approaches may perform better due to a variety of reasons. Therefore, new fuzzers (even if based on AFL) should adopt such approaches as true baselines, since outperforming AFL can be relatively easy.

Finally, our experiments emphasize the difficulty of drawing general conclusions. Simply reporting the *'average case'* can cover up some exceptional performance for single test cases or runs. In fact, our effort to precisely benchmark different aspects of AFL encountered three main problems:

- **Randomness**. Despite the fact that FuzzBench repeats each experiment up to 20 times to mitigate the random nature of fuzzing, most of the results were not statistically significant. This affected some tests more than others, depending on the actual magnitude of the impact of a given feature. The worse case we observed was in the experiments on timeouts, where even though results seem to suggest that longer timeouts are beneficial for slow targets, *none* of the results were statistically significant. Thus, conclusive experiments to fine-tune a fuzzer might require very large numbers of runs.

- **Target-dependency**. Even when results were statistically significant, the conclusions were often target-specific. In other words, the *"common sense"* the community derives from specific targets can be

misleading and difficult to generalize, and specific targets often highlight how a feature that is often beneficial can be largely outperformed by other configurations in a specific case (novelty search is a great example of this behavior).

- **Introspection.** A final takeaway is that looking only at bugs and code coverage is often insufficient to really understand the effect and impact of a given technique. More fine-grained ways to introspect the operation of a fuzzer in benchmarking tools (such as by reporting the numer of timing-out inputs) can greatly help the community to perform more quantitative measurements.

Chapter 6

The LibAFL Fuzzing Framework

Fuzzers are tools designed to execute a target application with a large number of automatically-generated inputs. Their goal is to discover problematic states, often associated with the presence of security vulnerabilities. Because of their effectiveness, fuzzers have become an essential asset in the arsenal of both developers and security researchers.

Many off-the-shelf fuzzers are available to the public, some of which are now considered de-facto standards for general-purpose applications: AFL [180, 67], AFL++ [65], HonggFuzz [158], and LibFuzzer [105]. These fuzzers are very popular among security testers and, for example, routinely discover thousands of bugs on OSS-Fuzz [3], an extensive fuzzing effort for open-source software.

Unfortunately, while off-the-shelf fuzzers are great tools that are easy to set up and use for non-experts, they often show their limitations for experienced users. In fact, to test complex applications or to adapt to different types of targets, such as operating systems kernels, device drivers, or embedded devices, experts often resort to creating new fuzzers, or modifying existing ones to fit their needs. For instance, academic researchers often implement algorithmic improvements and new ideas in small prototypes, often built on top of AFL or AFL++. While this satisfies the need of the reviewers in terms of reproducibility of the results, it also resulted in an incredible number of mostly-incompatible forks.

This is due to the fact that all existing fuzzing frameworks are not designed to be extensible. Thus, researchers are forced to reinvent the wheel over and over when implementing their prototypes, often missing out on features that are present in other forks and that are too complex to port

or re-implement. Some projects, notably AFL++ [65], proposed highly configurable architectures for fuzzing. However, they are not sufficiently generic (e.g., all inputs are represented as byte arrays, thus requiring hacks and workarounds to integrate structured and grammar fuzzing techniques) nor properly compartmentalized (thus requiring forking the project to adapt it to new techniques).

This problem is not only an engineering issue, but it also highlights the lack of a standard definition of the entities that define a modern fuzzer. Manes et al. [111] published an academic survey that covers all fuzz testing efforts until 2019. The authors highlight the enormous number of public fuzzers and categorize some common high-level concepts in a generic fuzzing algorithm. While this high-level categorization is sufficient for a systematization, the entities, and their relationships are too coarse-grained to develop a fuzzer framework according to this definition.

The fragmentation of the fuzzing landscape has three critical consequences on the research in the field:

1. Orthogonal contributions are difficult to combine.
   Several hundred, if not thousands, of different improvements have been proposed in the last decade to increase the effectiveness of fuzz testing. However, a new corpus scheduler implemented on top of AFL cannot be easily combined with a new mutator implemented in a custom fuzzer. As we mentioned before, this hinders the progress of fuzzing as a whole. Each individual tool focuses on a few advanced techniques but cannot take advantage of other orthogonal approaches proposed by other researchers.

2. Individual contributions are difficult to assess.
   A common drawback of many papers on fuzzing is that the authors compare their technique (for instance, a scheduler) which they implemented on a certain fuzzer, with previously-proposed solutions implemented in different tools. Thus, it is often difficult to understand whether better results are only due to the novel algorithm and not the result of other components of the fuzzer.

3. Different solutions are difficult to compare.
   While dozens of different techniques exist for every aspect of fuzzing, a third-party comparison would require a considerable re-implementation effort, typically reserved for surveys and systematization of knowledge papers. As a result, only a selected amount of solutions have been properly tested and compared on the same datasets.

We believe that these three issues are essential roadblocks that significantly slow down the progress of fuzzing, the transition of new techniques from academia to industry, and the development of new solutions, due to an extensive duplication of work.

Our Approach. Therefore, in this chapter we propose LibAFL, a novel fuzzing framework written from scratch in Rust. LibAFL consists of a collection of libraries that can be used to build custom fuzzers by combining components based on extensible entities. It achieves this goal thanks to several factors: (a) it is easily extensible; (b) it is based on a categorization of components of modern fuzzers; (c) it is designed to exploit the features of Rust, such as easy and fast serialization of objects and component slotting at compile-time; (d) it already implements a wide range of fuzzing algorithms, features, and instrumentation options proposed by recent works in the field.

LibAFL's building blocks can be used to recreate several modern fuzzing solutions. Thanks to the extensible design, researchers can combine blocks and experiment with compositions of beneficial techniques. In this chapter, we use a range of building blocks implemented in LibAFL to combine and test compelling fuzzing approaches that were never before evaluated, and others that were never evaluated on top of the same baseline. To the best of our knowledge, we are the first work to conduct such an extensive re-implementation (we integrated techniques from 20 previous works) and evaluation (15 different techniques) by using the same baseline. Additionally, we evaluate combinations of these techniques and provide insights into the effectiveness of these combinations.

We also show how LibAFL is a robust base to develop standard and more exotic fuzzers. In the first category, we build a generic bit-level fuzzer that uses an optimal combination of known techniques and show how the result outperforms all state-of-the art generic fuzzers like AFL++, LibFuzzer, and HonggFuzz. We then re-implemented a differential fuzzer for the Ethereum virtual machines [110] by using a custom feedback based on the VM state. We compare this fuzzer with its original implementation, and show how our version outperforms it in terms of uncovered differences in the two tested VMs.

Contributions. In short, in this chapter, we propose the following contributions:

- We identify and model common building blocks used by modern fuzzers;

- We present LibAFL, a novel open-source fuzzing framework written from scratch in Rust;
- We implement state-of-the-art building blocks and techniques;
- Based on these building blocks, we evaluate 15 techniques proposed in prior work, as well as a range of novel combinations;
- We present a case study that re-implements a differential fuzzer using custom feedbacks;
- Our generic fuzzer outperforms all off-the-shelf fuzzers;

LibAFL is a widely used Free and Open Source Software available at

https://github.com/AFLplusplus/LibAFL

6.1 American Fuzzy Lop ++

In the landscape of existing solutions, our previous work, American Fuzzy Lop ++, stands out as a community-driven fork of AFL. AFL++ serves as a comprehensive platform that amalgamates a diverse range of fuzzing techniques while offering a certain degree of extensibility and adaptability. Since 2019, we contributed significantly to AFL++ by incorporating various noteworthy techniques previously explored in the realm of fuzzing and performed some comparative experiments. This includes notable implementations like MOpt [107] and AFLFast [27].

One of the key innovations introduced by AFL++ is the development of a plugin interface known as "custom mutators." This interface empowers the users to tailor the tool to their specific needs by creating custom mutations and testcase minimization. Additionally, AFL++ offers a range of hooks that trigger during the fuzzer's lifecycle, allowing users to intervene, for instance, when a testcase is retrieved from the corpus.

While AFL++ marked an initial step towards achieving our goals, it also inherited inherent limitations from its predecessor, AFL. Notably, AFL++ retained a monolithic C codebase for many tasks unrelated to mutator development, lacking clear separation of components based on sound software engineering principles. This architectural constraint presented challenges.

In recent years, the fuzzing community has witnessed the emergence of several forks of AFL++ [164, 109, 155, 62], each continuing the tradition of AFL by introducing incompatible forks implementing orthogonal

techniques. Additionally, since the publication of its academic paper, the AFL++ codebase has grown in both size and scope as more techniques have been incorporated. This expansion has made the software increasingly difficult to maintain effectively.

To address these concerns and to ensure a more robust and maintainable platform, we made a deliberate choice to embark on the development of LibAFL from the ground up. This approach involved designing LibAFL with a fresh perspective, steering clear of extending AFL++ even though it provided a solid foundation to build upon. Our aim is to establish LibAFL as a modern and versatile fuzzing framework that addresses the limitations inherited from AFL++ and provides researchers and security professionals with a flexible and sustainable tool for conducting state-of-the-art fuzzing research.

Eventually, AFL++ will be a frontend fuzzer using LibAFL as backbone, as described later in Chapter 7.

6.2 Entities in Modern Fuzzing

To support the design of our framework, we first identified a set of 9 basic entities that are commonly present in most modern fuzzers. In this section, we present these entities and provide some examples by using state-of-the-art fuzzers.

Input – Formally, the input of a program, or a system in general, is the data taken from external sources that affect its behavior. In our model of an abstract fuzzer, we define Input as the internal representation of the program input (or a part of it). In the simplest case, the input of the program is a single-byte array. Fuzzers such as AFL store and manipulate this byte array directly, delivering the result to the target upon execution.

However, there are cases in which a byte array is not an ideal representation of an Input, e.g., when the target expects a sequence of system calls [161]. In this case, a fuzzer does not internally represent the Input in the same way that the program consumes it. Another example is the inputs for grammar fuzzers like NAUTILUS by Aschermann et. al. [13]. Here, the fuzzer internally stores Inputs as Abstract Syntax Trees, a data structure that can be easily manipulated while maintaining validity according to the grammar. Since the target expects a byte array as input, the tree is serialized to a sequence of bytes just before the execution. Other fuzzers may also use other input representations, such as sequences of tokens encoded as integers [144], or the intermediate representation of a programming lan-

guage [75].

Corpus – The Corpus is a storage for inputs and their associated metadata. Different kind of storage affects the capabilities of a fuzzer, for instance, a corpus that lives entirely in memory makes the fuzzer faster but can quickly exhaust the available memory when fuzzing large targets, while a corpus stored on disk allows the user to inspect the state of the fuzzer but introduce a bottleneck on disk operations.

Most mainstream fuzzers [180, 105, 27] store the corpus on disk, but this choice affects the scalability of parallel fuzzing and requires a standard library to perform the file IO operations.

In our model, a fuzzer requires at least two separate corpora: one that is used to store *interesting* testcases (6.2) that are used as component of the evolutionary algorithm of the fuzzer, and another one used to store the *solutions*, i.e., the testcases that fulfill the objective of the fuzzer (e.g., program crashes).

Scheduler – The Scheduler is a component tied with the corpus. It is the way the fuzzer asks for the next testcase to fuzz, typically by selecting one entry from the corpus. Naive schedulers implement, for instance, a simple FIFO policy or a random selection. More complex schedulers may use probabilistic algorithms based on introspection statistics about the fuzzer [25] or apply other schedulers to a subset of the corpus, as AFL does when calculating the "favored" minset.

Other examples include schedulers that try to mitigate the explosion of the corpus caused by too sensitive feedback [164] or to prioritize testcases with interesting properties [166].

Stage – The Stage is a component defining an action to perform on a single testcase from the corpus. Usually, the scheduler selects a testcase and then the fuzzer executes every stage on that given input. The Stage is a very broad entity and in existing fuzzers, it is usually the component that invokes one or more times a mutator on the input (e.g., the *random havoc* stage in AFL) or an analysis stage that, for instance, perform taint tracking to gather information in a white-box fuzzer [35].

Another widely known stage adopted by many fuzzers is the minimization phase, introduced in AFL, reduces the size of a testcase obtained from the corpus while maintaining the triggered coverage points.

Observer – The Observer is an entity that provides information from a single execution of the target. To reason on an execution of an input, the fuzzer executes it and then looks at the observers. A snapshot of the

observers state after an execution is equivalent to the execution itself in terms of effects on the fuzzer state. Defining the observers in this way is particularly useful when a distributed fuzzer can send the observers state across multiple nodes. This avoids the need for re-execution with the same input when fuzzing a very slow target.

An example observer is the coverage map, used by common coverage-guided fuzzers such as AFL or HonggFuzz. The map is filled during execution to report the executed edges. This information is not preserved across runs and it is an observation of a dynamic property of the program.

Other fuzzers, such as Ijon [15] or FuzzFactory [126], use different forms of observers but always rely on a map to keep track of additional metrics beyond code coverage.

Executor – The Executor is the component responsible to execute the target system given an input from the fuzzer. In different fuzzers, the embodiment of this entity may change a lot. For instance, for in-memory fuzzers like LibFuzzer an execution is a call to a harness function, while for hypervisor-based fuzzers like Nyx [145] it requires an entire operating system to re-start from a snapshot at each run.

In our model, the Executor is the entity that defines not only how to execute the target, but all the volatile operations that are related to a single run of the target. So the Executor is, for instance, responsible for informing the program about the input that the fuzzer wants to use in the run, e.g., by writing to a memory location or by passing it as a parameter to the harness function. The Executor also maintains a set of Observers linked with each execution.

Feedback – The Feedback is an entity that classifies the outcome of an execution of the program under test as interesting or not. Typically, this information is used to decide whether the corresponding input is added to a corpus.

Most of the time, the notion of Feedback is deeply linked to the Observer, but the two are different concepts. In fact, the Feedback usually processes the information reported by one or more observers to decide if the execution is interesting. While the concept of "interesting" is abstract, it is typically related to a novelty search (i.e., interesting inputs are those that reach a previously unseen edge in the control flow graph). In another example [126], an Observer can be used to report all the sizes of memory allocations and a maximization Feedback can be used to maximize these values to spot pathological inputs in terms of memory consumption.

The process that identifies interesting inputs also has a second important

goal in fuzzing: finding the *solutions* that satisfy specific objectives, for example, an observable crash in the target program. This type of feedback, the Objectives, act as an oracle that describes the expected outcome of the fuzzing campaign, for instance, a set of crashing testcases with a unique stacktrace like in HonggFuzz or an input that triggers a crash along a specified path like in AFLGo [26].

Mutator – The Mutator is an entity that takes one or more Inputs and generates a new derived one. Mutators can be composed of other mutators and they are generally linked to a specific Input type. In a traditional fuzzer, mutators are composed of many bit-level mutations like bit flip or blocks swapping. A mutator can also be informed about the input format and mutate the internal representation of the Input, for instance, by swapping nodes in an Abstract Syntax Tree in case of a grammar fuzzer. Mutators are usually the part of a fuzzer that changes more often when creating a custom fuzzer.

Generator – A Generator is a component designed to generate a new Input from scratch. For instance, a random generator can be used to generate random inputs. While less popular in feedback-driven fuzzing, there are notable exceptions that adopt Generators. For instance, Nautilus [13] uses a grammar-based generator to create the initial corpus and a sub-tree generator as a mutation of its grammar mutator.

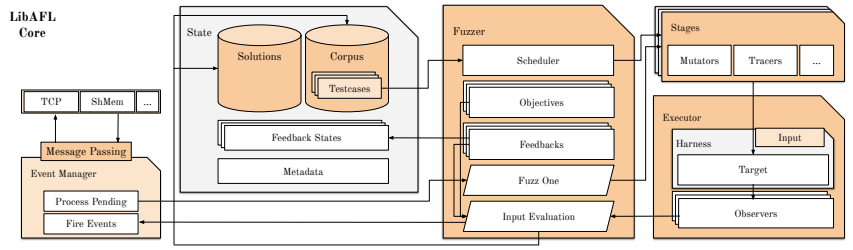


Figure 6.1: LibAFL Core architecture. Links are a representation of a non-comprehensive picture of the interactions.

6.3 Framework Architecture

The goal of LibAFL is to provide the basic blocks required to build a new generation of fuzzers through a modular design based on reusable components and reliable, fast, and scalable implementations of state-of-the-art

techniques. To achieve this objective, we decided to bound the framework's design to the actual programming language that we use, Rust, by exploiting its features from the design stage. In this section, we present and discuss the design of LibAFL as a system and its individual components.

6.3.1 Principles and High-level Design

The LibAFL framework is designed around three key principles:

- **Extensibility**, to allow the user to swap different implementations of the entities explained in Sec. 6.2 in or out, without touching other parts. This allows the seamless combinations of orthogonal techniques but also ease the design and development of new components;
- **Portability**, most of the existing fuzzers are OS-specific, running either under *nix or Microsoft Windows. To avoid this pitfall, we opted instead to design our core library in a system-independent way. Moreover, for maximum portability, we implemented a subset of LibAFL, including all core components, without any dependency on any standard library, thus allowing the users to write fuzzers for bare-metal targets like embedded systems and kernels;
- **Scalability**, no design choices must conflict with the ability to scale fuzzers over multiple cores and/or machines. Because of this, we design an event-based interface that enables and facilitates the communication between fuzzers;

As we already discussed, none of the existing fuzzing frameworks are completely extensible. Some are portable on different operating systems, like LibFuzzer [104] but none can compile on systems without a standard library. Last but not least, scalability is a known weakness of existing fuzzers. The design of AFL, and therefore of its many derivates, is based on disk IO communication and expensive syscalls such as `fork(2)` [177]. This causes a terrible performance when the fuzzer is scaled across multiple cores [59]. Other more scalable solutions, like HonggFuzz, are still based on syscalls to control the target and maintain a shared state between all the parallel threads, leading to lock contentions. On the other hand, LibFuzzer achieves greater scalability as different nodes cannot communicate while fuzzing, the corpus is merged after a defined time span, and the fuzzers are restarted.

To create a fuzzing framework following the three aforementioned objectives, we designed our system around three core libraries:

```
use libafl_sugar::InMemoryBytesCoverageSugar;
use libafl_targets::libfuzzer_test_one_input;

InMemoryBytesCoverageSugar::builder()
    .input_dirs(input_dirs)
    .output_dir(output_dir)
    .cores(cores)
    // For multi-node synchronization
    .broker_port(broker_port)
    .harness(|buf| {
        libfuzzer_test_one_input(buf);
    })
    .build()
    .run();
```

Listing 6.1: An example frontend with LibAFL Sugar.

- **LibAFL Core** is the main library and contains the fuzzing components and their implementations. A large part of this library depends only on Rust `core+alloc` and, thus, can run without any standard library;
- **LibAFL Targets** contains the code that lives within the target program, like the runtime library for coverage tracking;
- **LibAFL CC** provides the functionalities to write compiler wrappers for LibAFL, by providing to the user a set of compiler extensions useful for instrumentation;

In addition to these three core libraries, LibAFL contains several **Instrumentation Backends** that offer APIs to bridge LibAFL to different execution engines, such as QEMU usermode and Frida.

In our naming convention, all these libraries are part of a toolkit that is used to create fuzzers called **Fuzzer Frontends**. Some ready-to-use frontends are already available in an additional library in the framework, LibAFL Sugar, that provides a high-level glue API to quickly set up a frontend in just a few lines of code. We also provide Python bindings to the Sugar crate for quick prototyping without recompilation.

For instance, Listing 6.1 shows a simple fuzzer that bridges a LibFuzzer-style harness to a fuzzer that uses generic bit-level mutations and executes the target in-process written using the high-level APIs of LibAFL Sugar.

At the time of writing, the entire LibAFL framework, tests included, consists of 53k lines of Rust and 15.4k lines of C/C++.

6.3.2 The Core Library

Figure 6.1 shows the architecture of the core library in terms of links between components. Most components are a one-to-one mapping with the entities we discussed in Section 6.2, with the addition of three additional macro-components:
State, **Fuzzer** and **Events Manager**.

Each component is mapped to a Rust generic trait, allowing it to work in combination with any other orthogonal component. This configuration of the architecture is the standard architecture for a frontend proposed in LibAFL, but custom architectures can be defined, too. An alternative architecture, already implemented in LibAFL, consists of a pipeline without any executor, in which there is no traditional fuzzer loop, but the fuzzer is a service from which inputs can be requested. This way, LibAFL can be, for example, embedded in an emulation loop, to use in hooks of emulators such as Panda [52].

Zero-cost Abstractions

Extensibility comes with the price of introducing abstractions, which usually has a cost in terms of performance. As speed is an important metric in fuzzing, we devised a design that allows flexible abstractions without paying a noticeable cost at runtime.

Since the beginning, driven by micro-benchmarks during the early stage of development, we avoided traditional object-oriented patterns in favor of generic traits. This way, we leverage the design of the Rust programming language to allow the compiler to perform powerful optimizations. In LibAFL, each generic trait takes other related components as generic parameters. Sub-components are then defined via composition. In this way, we pay the cost of combinations of linked-but-independent entities at compile time, such as Executors and different kinds of Inputs. As a second design pattern, inspired by Haskell, we employ compile-time lists similar to hlist [121] to specify multiple objects, such as the set of observers of the set of mutations in a composable mutator. These lists have matching capabilities to retrieve single objects stored in the data structure. For instance, a feedback can access the observers that are useful to determine the interestingness of an execution by either name or type. By exploiting the powerful

compile-time facilities offered by Rust our code is compiler optimization friendly.

The State

The State is where all the non-volatile data resides. Everything that is part of the evolutionary algorithm's data must be included in the State, as well as the number of executions, the pseudo-random number generator state and the corpora (both the main and the solutions corpus).

As some types of feedbacks also need to maintain a state, for instance, the coverage observed so far in coverage-guided fuzzing, we introduce the **FeedbackState** component that is linked to both the state and the feedbacks. The instances of the feedback states are contained in State and are initially generated at the start of the fuzzing process.

The main purpose of having a place for the data of the fuzzer is to exploit the serialization facilities of Rust. Serializing and de-serializing a state allows any LibAFL-based fuzzer to stop and later restart from the exact same internal state. For in-process fuzzing, this novel approach allows LibAFL to recover instances from crashes by re-loading a serialized state in the crash handler, without the need to re-execute the entire corpus as previous solutions do.

The Fuzzer

The Fuzzer is a recipient for the operations that define what the fuzzer can do. It contains the Feedbacks, the Objectives, and the Scheduler, all independent operations that may alter the fuzzer state. These stages are separated from the fuzzer to respect the borrowing rules of Rust [1], as they may invoke some operations that alter Fuzzer and State at the same time.

The Fuzzer, in addition, provides the definition of how a single testcase should be processed and how to evaluate a new input. By default, the standard implementation of it, which consists of feedback-driven fuzzing, defines FuzzOne, the operation responsible to process a single testcase, as the invocation of the scheduler to get the testcase to fuzz and the invocation of every stage on the testcase. InputEvaluation, the operation that evaluates if an input must be added to the corpus, is by default the execution of the target program and the decision if it is interesting or a solution using the feedbacks.

[1] https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Custom architectures that implement their own Fuzzer and State entities, can be used to recreate concepts like Vuzzer [138], which decouples input generation from the immediate evaluation, in LibAFL.

The Events Manager

The Events Manager is an interface for generating and processing events, which can be used to implement multi-node synchronization in a parallel fuzzer or simply for the purpose of logging. The Events Manager is designed to maximize scalability. In fact, if we assume a communication channel that scales linearly (such as shared memory message passing [152]) the Manager does not introduce any further bottleneck as each fuzzer works on completely separated data and the process of pending events is deferred to specific reconciliation points in the fuzzer loop, triggered before the fuzzer requests a new testcase from the scheduler.

Our framework includes a rich set of events. For instance, a component can be notified when one fuzzer adds a new testcase to its corpus, receiving an event containing a serialized version of the input and the set of observers that were considered interesting by the feedback.

The Metadata System

Fuzzing algorithms often need to reason about the information associated with a given testcase or the overall state of the fuzzer. Therefore, LibAFL must provide a way to extend the data in the testcases of the corpus and the data maintained in the state. A naive but effective solution would be to redefine new types by composition, but in this case, the developer would need to be aware of each piece of metadata required by all the employed algorithms. Thus, in order to provide this capability while maintaining simplicity and performance, we designed a dedicated Metadata System for the State and Testcase components. In particular, in LibAFL any struct that implements the `SerdeAny` trait, a trait that we created to allow the serialization of trait objects [2] without the requirement of a standard library, can be used as metadata. This trait requires serialization capabilities and static lifetime [3], as the instances must be able to be converted to trait objects.

LibAFL provides then serializable maps that can store any instance that can be cast to a `SerdeAny` trait object. Both the Testcase and State holds a map of this type as an extensible container for metadata. In this way,

[2] https://doc.rust-lang.org/book/ch17-02-trait-objects.html
[3] https://doc.rust-lang.org/rust-by-example/scope/lifetime/static_lifetime.html

different but related components can cooperate by operating on the same metadata, while completely ignoring what the other components do with their own metadata. However, this is the only pattern in LibAFL that introduces a small runtime overhead, due to the map lookup (currently implemented as a hashmap).

Composable Feedbacks

A fuzzer may require combining multiple feedbacks to evaluate how "interesting" was a given input or to support different objectives. In LibAFL, to avoid the need to create new aggregated feedbacks from scratch, feedbacks can be composed by using logical operators. For instance, a fuzzer may not want to save every crashing input but instead perform some sort of crash de-duplication. In LibAFL, this can be achieved for example by using a feedback that considers crashing inputs as interesting and one that considers an input interesting when it triggers a new stack trace never observed before. In this case, a crash de-duplication objective can be achieved by combining the two aforementioned feedbacks with a logical *AND*.

The Monitor

The last component in a LibAFL-based fuzzer is the **Monitor**. It is the component that maintains the statistics collected from the triggered events and displays them to the user. While this component is not required for a working fuzzer, the lack of human introspection reduces the effectiveness of a fuzzing campaign. Monitors allow the developers to report and display custom stats and to implement various reporting interfaces, such as printing a status screen in the terminal, or forwarding the data to a Grafana web interface with statsd [4].

6.3.3 Instrumentation Backends

LibAFL can be easily plugged into any instrumentation backend, like a binary translator or a simple compiler instrumentation pass. By default, we provide additional libraries that tie LibAFL with some popular instrumentation backends: LLVM [95], SanitizerCoverage [101], QEMU usermode [21] and Frida [2].

The runtime in LibAFL *Targets* can be linked to any SanitizerCoverage target, adding coverage and comparisons tracking to the fuzzer, using a compiler flag. The SanCov support allows users to create frontends that are

[4] https://github.com/statsd/statsd

compatible with non-C/C++ SanCov-enabled targets, such as Atheris [90] for Python and cargo-fuzz [18] for Rust.

LibAFL CC provides a set of LLVM passes to extend Clang and other LLVM-based compilers to track edge coverage, context-sensitive and K-context-sensitive [17] edge coverage, N-gram coverage [162], coverage accounting [166], comparisons with CmpLog [65, 16] and dictionary tokens with autotokens, and an enhanced version of AFL++'s `dict2file` pass that extracts the tokens from interesting functions such as `strcmp`.

LibAFL QEMU bridges QEMU usermode, and full system in the near future, to Rust with a novel emulator API with hooking capabilities to have programmatic control over the target's execution. Around this interface to QEMU, the library exposes structures like executors and helpers to install default hooks for common fuzzing tasks, such as edge coverage tracking, guest snapshot-restore, and binary-only ASan [64].

LibAFL Frida offers similar capabilities to the QEMU bridge, but with the features of a DBI, without a clear host-guest separation. It includes a binary-only ASan and, unlike QEMU usermode, it can work on various operating systems other than Linux such as Windows, macOS, and Android.

Instrumentation capabilities in LibAFL are also offered for concolic execution, through LibAFL Concolic and its Rust bridges to SymCC [132] and SymQEMU [133]. Our API allows a user to write custom constraint collection filtering in Rust. At target runtime, the constraints are then reported back to a LibAFL-based fuzzer in an easy-to-manipulate format. These constraints can then be used in a mutator, for instance, to generate inputs invoking a solver, or for fuzzing, similar to what Borzacchiello et al. proposed [29].

In addition to these stable backends, LibAFL already has partial support for TinyInst [68] to instrument binaries on Windows and macOS, and for Nyx [145] for hypervisor-level snapshot fuzzing.

6.4 Applications and Experiments

In this section, we discuss some of the techniques implemented in LibAFL and their relation with the entities presented in the previous sections. While LibAFL already has many implemented techniques, in the first part of this section, we focus on four popular problems in fuzzing that are the focus of many published works in the literature: roadblocks bypassing (e.g., [16, 138, 35, 123]), structure-aware fuzzing (e.g., [13, 130, 22, 63]), corpus scheduling (e.g., [164, 36, 53, 166]) and energy assignment (e.g., [25, 27, 26, 98]).

A non comprehensive list of the techniques integrated in LibAFL at

the time of writing is listed in Table 6.1, alongside the information if the technique requires additional implementations of components in the fuzzer side or additional instrumentation code in the target side.

Table 6.1: List of techniques integrated in LibAFL, the first part only contains the techniques evaluated in Section 6.4.

Technique	New components	Additional instrumentation
RedQueen [16]	✓	✓
Auto-tokens [65]		✓
Value-profile [105, 126]		✓
Block coverage accounting [166]	✓	✓
Function coverage accounting [166]	✓	✓
Loops coverage accounting [166]	✓	✓
Corpus culling scheduler [180]	✓	
Weigthed scheduler [65]	✓	
FAST power schedule [27, 65]	✓	
COE power schedule [27, 65]	✓	
EXPLORE power schedule [27, 65]	✓	
MOpt [107]	✓	
Nautilus [13]	✓	
Grimoire [22]	✓	
Gramatron [155]	✓	
Token-level [144]	✓	
NeoDiff [110]	✓	
LIN power schedule [27]	✓	
QUAD power schedule [27]	✓	
EXPLOIT power schedule [27]	✓	
SymCC [132]	✓	✓
SymQEMU [133]	✓	✓
Hitcounts [180]	✓	✓
Ngram coverage [162]		✓
Context-sensitive coverage [35]		✓
QASan [64]		✓
Atheris (compatibility) [90]	✓	
cargo-fuzz (compatibility) [18]	✓	

After discussing how these techniques are implemented, we evaluate them in three different sets of experiments:

1. We measure the performance in terms of code coverage and bug detec-

tion of several approaches implemented and ready-to-use in LibAFL;

2. We show how we can combine orthogonal approaches implemented in our framework to build new and never evaluated before fuzzers and measure their performance;

3. Lastly, to show the efficiency of our framework in a traditional context, we compare and evaluate a new generic bit-level fuzzer based on LibAFL using the previously presented techniques against other state-of-the-art fuzzers like AFL++ and HonggFuzz on FuzzBench [116];

We ran the first two sets of experiments on a x86_64 machine equipped with an Intel® Xeon® Platinum 8260 CPU with a clock of 2.40 GHz. The dataset is a subset of the FuzzBench suite, selected to include programs with diverse features. We run each session for 24 hours and repeated each experiment five times to mitigate the effect of randomness in fuzzing.

For the comparison with other fuzzers, we run the experiments on the full FuzzBench suite on the service offered by Google. Each run was 23 hours long and repeated 20 times. The benchmarks suite provides the initial corpus for every benchmark and we used the default setting. The overall ranking that we discuss is based on the average normalized score computed by FuzzBench, which represents the percentage of the highest reached median code or bugs coverage on a given benchmark.

Finally, in the last part of the section, we present a case study about how easy, we can implement with LibAFL a fuzzer that is different from the traditional setups like the ones shown in the first part, a differential fuzzer that can spot logic bugs in Ethereum VMs with a different kind of feedback based on the state of the VM instead of code coverage.

In the end, we discuss other implemented approaches and their relations without providing an evaluation for the sake of time and brevity.

6.4.1 Bypassing Roadblocks

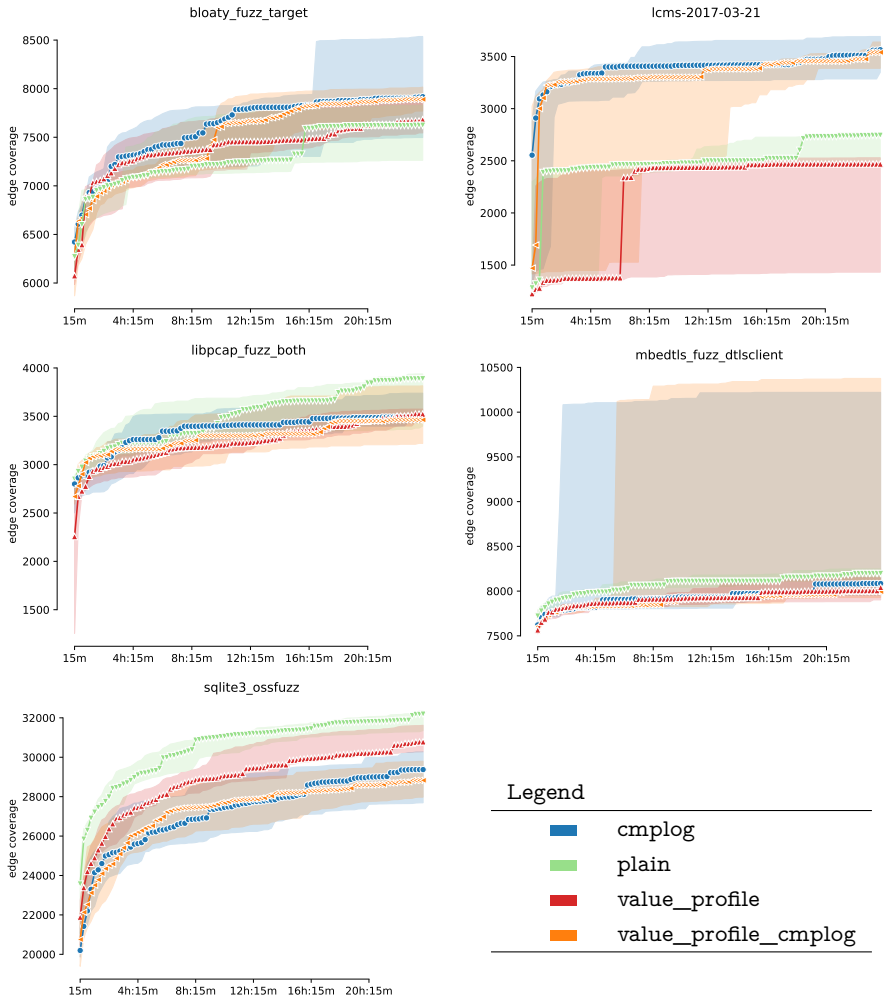


Figure 6.2: Uncovered code coverage over time (24h) of the roadblock bypassing experiment.

An important field of research in fuzz testing is the development of new techniques to increase code coverage by bypassing hard-to-solve constraints. For instance, multi-byte comparisons pose a severe problem for fuzzers that employ a generic bit-level mutator, since random generic mutations cannot bypass these comparisons as the solution space is huge and blind guessing impractical. LibAFL provides several ready-to-use techniques to implement

fuzzers that can overcome these roadblocks.

The first, in chronological order of appearance in the literature, is the *value-profile* [103] proposed by LibFuzzer in 2016. This technique tries to solve comparison instructions by maximizing the number of matching bits between the two operands of the instruction. In LibAFL, this is implemented with a map observer and a feedback that maximizes the entries of a map and considers an input as interesting when at least one new max is discovered. This type of feedback, `MaxMapFeedback`, is builtin into our framework and can be easily combined in OR with the basic edge coverage.

The second technique provided by LibAFL is *cmplog* from AFL++. This solution is based on the approach adopted by RedQueen [16] and Weizz [63], and can bypass comparisons by finding and replacing input-to-state values. It works by instrumenting the comparison instructions and any function with two pointers as arguments, and logging the related values in a map at runtime. This logging operation is done only once for each testcase in the corpus and in our framework this is implemented with a second executor with one observer that handles the cmplog map. The executor is invoked in a tracing stage at the beginning of the pipeline and the logged values are stored as metadata. Later on, a custom mutator matches the pattern in the input and replaces them with the other operand of the comparison in a specific mutator.

The third technique, *autotokens*, was also inspired by AFL++, and can only be used by instrumenting the target with an LTO pass. In LibAFL CC, this instrumentation is available for regular compilation, as well. This pass extracts the tokens from the comparison instructions and the functions with immediate values, encoding them in a section of the binary. A LibAFL-based fuzzer can then retrieve the tokens, add them to the dictionary in the State's metadata, and use the tokens in the mutator without any overhead. For this experiment, we consider autotokens as the baseline. In fact, since it does not introduce any overhead, there is no reason to not use it in a fuzzer.

Cmplog and value-profile, on the other hand, require additional instrumentation and value-profile can bloat the corpus with more input as it increases the sensitivity [162] of the fuzzer. Thus, we now evaluate four different options. These include `plain` (the baseline with autotokens), `value_profile`, and `cmplog`, as well as a new `value_profile_cmplog` which combines both of the aforementioned techniques. This combination was never evaluated in previous studies and shows how the composability of our solution allows experimenting with different combinations of components and simplifies the development of complex fuzzers.

Table 6.2 reports the coverage growth graphs over 5 benchmarks from FuzzBench. Overall, `cmplog` is the best performer (95.94), closely followed by `value_profile_cmplog` (95.03), and `plain` (94.65). Instead, `value_profile` performed considerably worse (90.13). This is interesting, as it suggests that autotokens alone is able to solve many roadblocks without additional overhead, allowing `plain` to shine on libpcap which is a benchmark with many input-to-state comparisons.

This confirms that most roadblocks are input-to-state and the solving capabilities of `value_profile` are not an adequate reward for the additional sensitivity it introduces (due to the possible internal wastage of the corpus by too many similar testcases). We think that the combination of the two techniques, however, can have interesting target-specific applications that can be investigated in the future.

6.4.2 Structure-aware Fuzzing

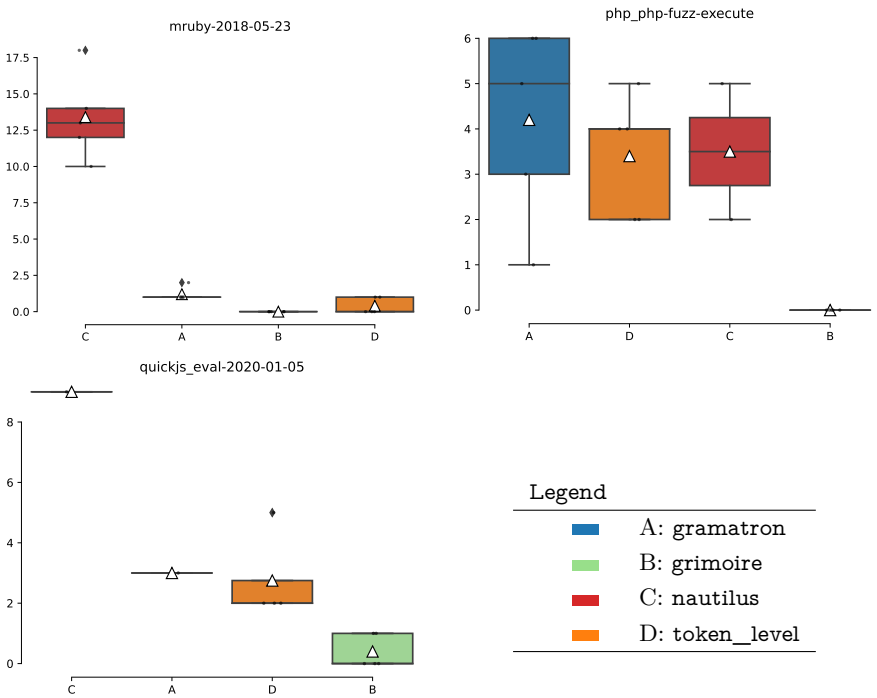


Figure 6.3: Uncovered bugs after 24h of the structure-aware fuzzing experiment.

While generic mutators can effectively stress parsers by generating invalid inputs, it is also important to fuzz deeper paths beyond the parsing routines to spot bugs in these code regions. A common solution to this problem is to make the fuzzer aware of the input format. While generating testcases from a specification is one of the oldest embodiments of fuzz testing, recent works in the literature have explored the combination of modern feedback-based fuzzing with a mutator that is structure-aware [13, 73, 130]. Beyond this, other approaches [22, 63, 144] proposed to approximate structure-aware fuzzing with learning heuristics without requiring any user-provided specification, thus working on targets without a known input format and reducing the amount of human work.

LibAFL provides several techniques to deal with structured inputs, taking advantage of the flexibility of all the other components that are agnostic to the input type. Nautilus [13] is a grammar-based coverage-guided fuzzer that evolves a corpus of syntax trees with mutations like subtree generation and replacement from another input in the corpus. In our framework, we implemented an input type for Nautilus, a generator that can create testcases from scratch, and a set of mutators that make use of the generator to create subtrees. All the other entities remain untouched and immediately compatible with the grammar fuzzer. For instance, the `ScheduledMutator` (the trait for mutators that can schedule other mutators as mutations), is employed seamlessly with a maximum of 8 stacked mutations. Another technique available in our framework is a re-implementation of Gramatron [155], a grammar-based fuzzer that employs a grammar-to-automata conversion to implement fast mutators. In LibAFL the grammar preprocessing utility is provided as a tool and the associated structures are specular to the ones used by Nautilus, with a different underlying implementation.

As an example of approaches that perform grammar learning, LibAFL implements Grimoire [22], a fuzzer that uses the portion of inputs that induced the novelty in coverage as tokens to build generalized "tree-like" inputs and perform grammar-like mutations. It also employs token-level fuzzing [144], an approach based on token extraction with a lexer. While the original solution was specific to JavaScript, our implementation is generic and can be applied to any programming language. Classic bit level mutations are then applied to encoded inputs and that is decoded before the target execution.

To evaluate these three approaches we decided to use the number of uncovered bugs instead of code coverage. In fact, the effectiveness of this type of fuzzer is less dependent on code coverage, especially when compared to variants that generate invalid inputs and inflate the coverage by exploring

error paths.

In this experiment, as both GRIMOIRE and token-level require some initial seeds, we used NAUTILUS to generate 4096 initial inputs for these two fuzzers to avoid a bias due to the quality of the seed corpora [83] provided by FUZZBENCH. Table 6.3 shows the number of bugs found by each variant over 3 popular compilers. Grammar-aware approaches are still superior, with a huge boost for NAUTILUS on 2 benchmarks over 3, but the performance of the token-level approach is surprisingly similar to NAUTILUS on PHP, given that they start from similar seed inputs.

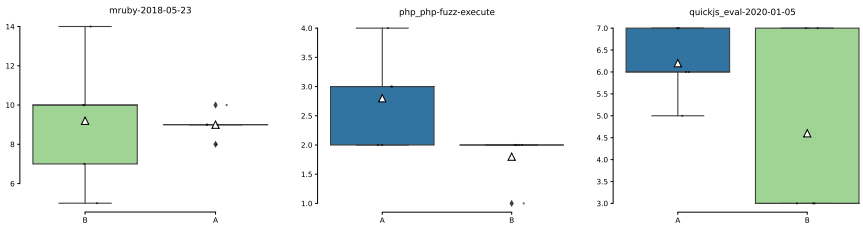


Figure 6.4: Uncovered bugs after 24h of the NAUTILUS +MOPT fuzzing experiment.

Given the promising results of this experiment, we decided to investigate whether we can further improve the best performer, NAUTILUS, by combining it with other orthogonal techniques provided by LIBAFL. For instance, we combine it with the MOPT mutation scheduler [107], to create a new and never evaluated variant. MOPT has been used to date only to schedule bit-level mutations by assigning probabilities to mutations based on the observed effectiveness during a learning phase by using a Particle Swarm Optimization algorithm. We repeated the experiments and compared this new variant against the 3 grammar benchmarks, running the fuzzer 5 times for 24 hours. The results are reported in Table 6.4 that shows how NAUTILUS (▬) perform well when coupled with MOPT (▬) on `mruby`, but worst on `php`. Overall, this confirm the highly target-dependent nature of MOPT that was observed [65] for bit-level fuzzing.

6.4.3 Corpus Scheduling

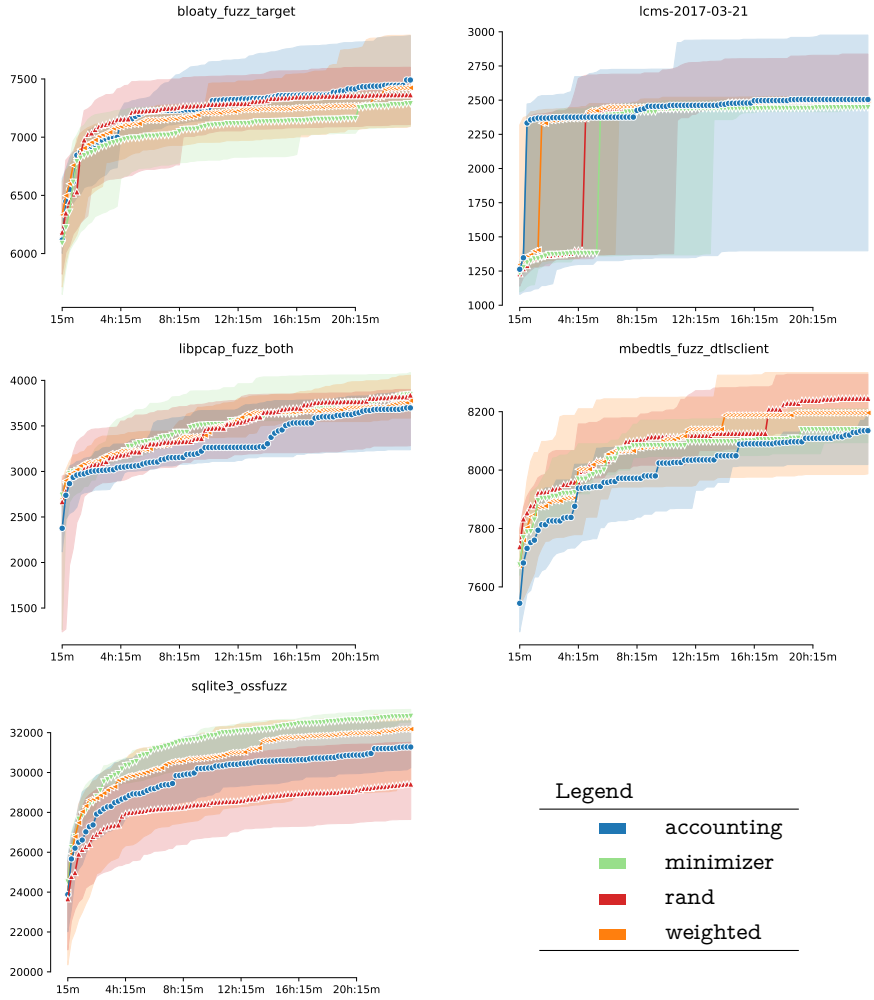


Figure 6.5: Uncovered code coverage over time (24h) of the corpus scheduling experiment.

The choice of the next testcase to use from the corpus is the focus of many different studies. The simplest solutions rely either on random selection or on a FIFO queue. LibAFL provides both, along with additional schedulers inspired by other state-of-the-art fuzzers.

The first is taken from AFL, which in every queue cycle selects a subset of "favored" seeds from the corpus. The seeds are chosen based on

execution speed and input length while preserving the maximum coverage. In LibAFL, we implemented a generic version of this approach called `MinimizerScheduler`, which computes the minset based on the entries of a given map feedback but with customizable weights. For simplicity, in the following experiment, we use the traditional weighting policy used by AFL.

The second scheduler uses a recent improvement proposed in AFL++ based on probabilistic sampling. The idea is to map each testcase in the corpus to a probability and a more "promising" neighboring testcase in terms of a computed score. The score is computed by using various metrics, including execution time and coverage map size, and it is used to calculate the probability too. For the selection process, the scheduler chooses a random testcase from the corpus, and a random number between 0 and 1. If this number is less than the probability associated with the testcase, this testcase is selected, otherwise, the more promising neighbor is picked.

The third scheduler is taken from TortoiseFuzz [166] and prioritizes inputs by using three security impact metrics: memory operations with both block and function granularity, and loop back edges counting. The tracking of these metrics requires a custom instrumentation, implemented in LibAFL CC with LLVM and a new associated observer. The scheduler prioritizes inputs with higher scores and, breaking ties based on input length and execution time.

Table 6.5 shows the results of four fuzzers based on the aforementioned schedulers: `accouting` is the fuzzer using the TortoiseFuzz instrumentation and a scheduler with function-level granularity, `minimizer` is using the AFL's algorithm, `rand` the random selection baseline, and `weighted` is using the probabilistic scheduler from AFL++.

In terms of average normalized score of the uncovered coverage, `weighted` achieves the best results (with a score of 98.91), closely followed by `minimizer` (98.71), `accounting` (98.03), and `rand` in last position (97.50). The small difference among all solutions is interesting and shows that, despite the huge attention given to this problem in the literature, a simple random approach still achieves decent results and it is perfectly suitable for real-world fuzzing campaigns on fast targets. We also expected `accouting` to outperform `minimizer`, which instead was not the case. However, unlike in the original evaluation carried out in the TortoiseFuzz paper by using AFL, LibAFL achieves much higher throughput by executing the target in-process, thus reducing the difference and impact of these scheduling techniques. While on fast targets the difference is minimal, we believe that the scheduling problem is crucial on slow targets, where deciding which testcase to fuzz beforehand can have a large impact on the fuzzing campaign.

6.4.4 Energy Assignment

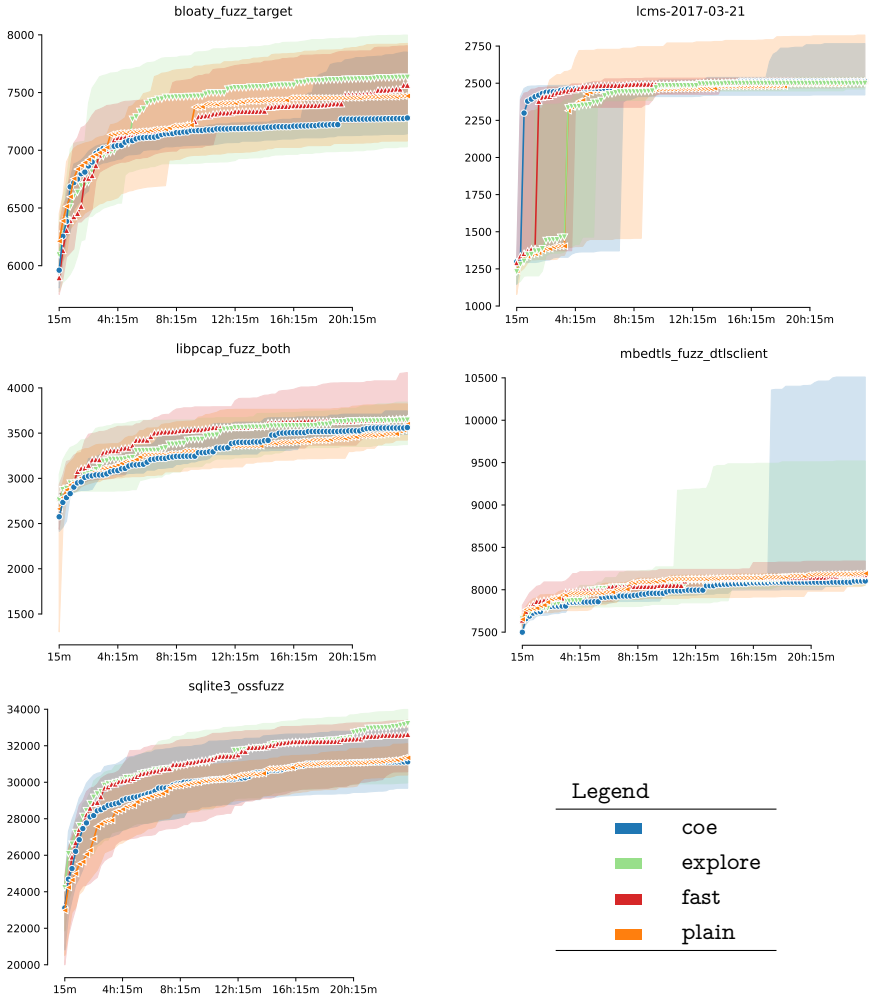


Figure 6.6: Uncovered code coverage over time (24h) of the energy assignment experiment.

Energy assignment tries to answer the question of how many times a single input in the corpus needs to be mutated to create new testcases. The general problem, also known as *power scheduling* problem, was introduced in the literature by Böhme et al. [27] in 2016.

The most naive solution is to use a constant value, while the most commonly used simple approach [105, 158] assigns to each seed a random value

in a given interval. LibAFL also provides this simple algorithm, named `plain`, with a range between 1 and 128 when using the default mutational stage.

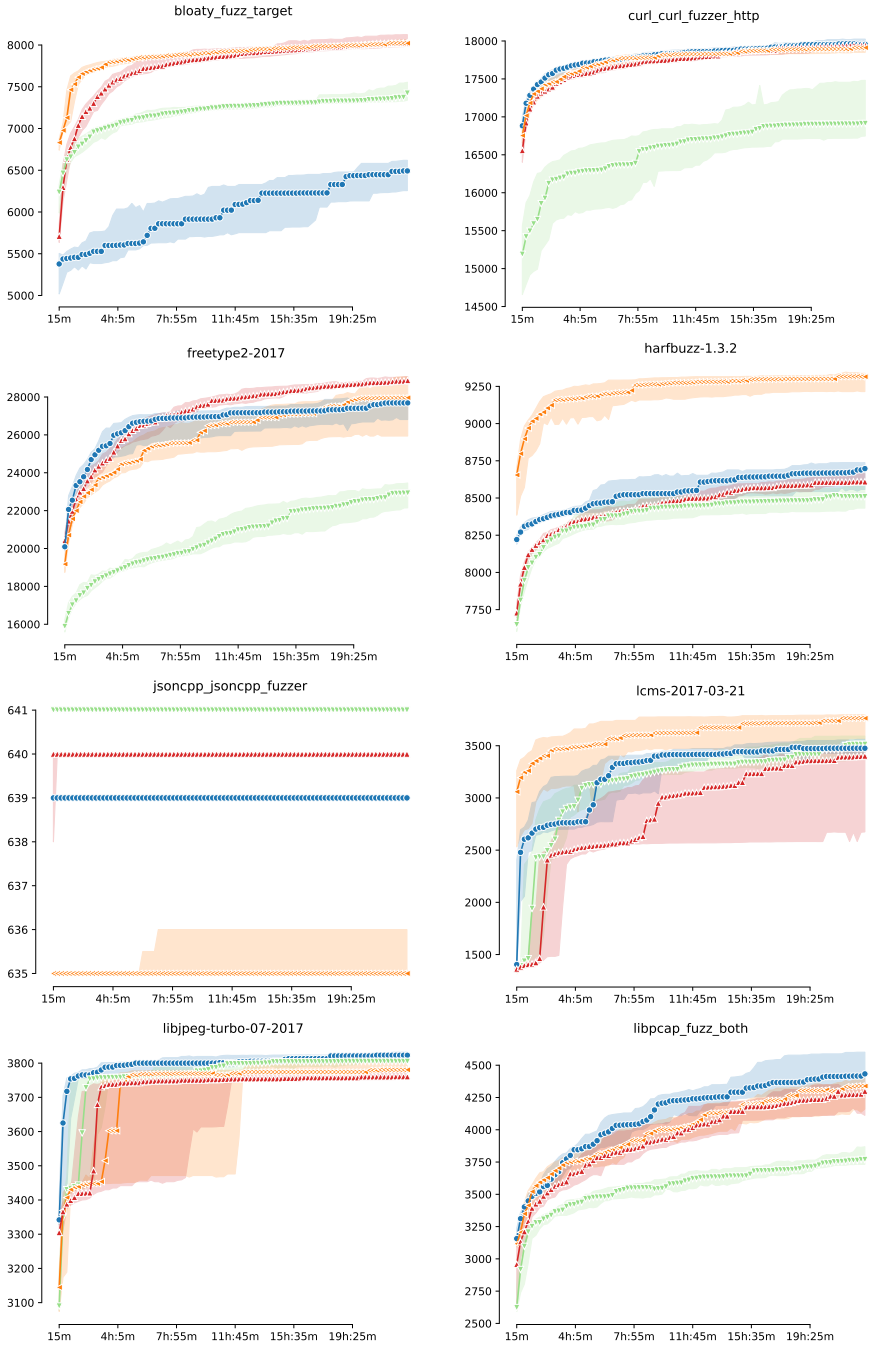
While many solutions tune this specific parameter, often for domain-specific fuzzers [130, 26], the seminal work of AFLFast remains the most complete coverage of this problem for generic fuzzing. AFLFast proposed six different algorithms: *exploit*, *explore*, *coe*,*fast*, *lin* and *quad*. In detail, *exploit* assigns energy proportional to some metrics like execution time of the input, coverage density, and creation time of the testcase. *Explore* assigns low energy, dividing the energy of exploit by a constant. *Coe* is an exponential scheme that assigns 0 as energy to inputs that trigger high-frequency edges until they become low-frequency. *Fast* is an extension of *coe*, in which instead of assigning 0 the scheme assigns energy inversely proportional to the amount of visited high-frequency edges. *Lin* assigns energy linearly w.r.t the times the testcase has been chosen to be fuzzed, while *quad* is quadratic.

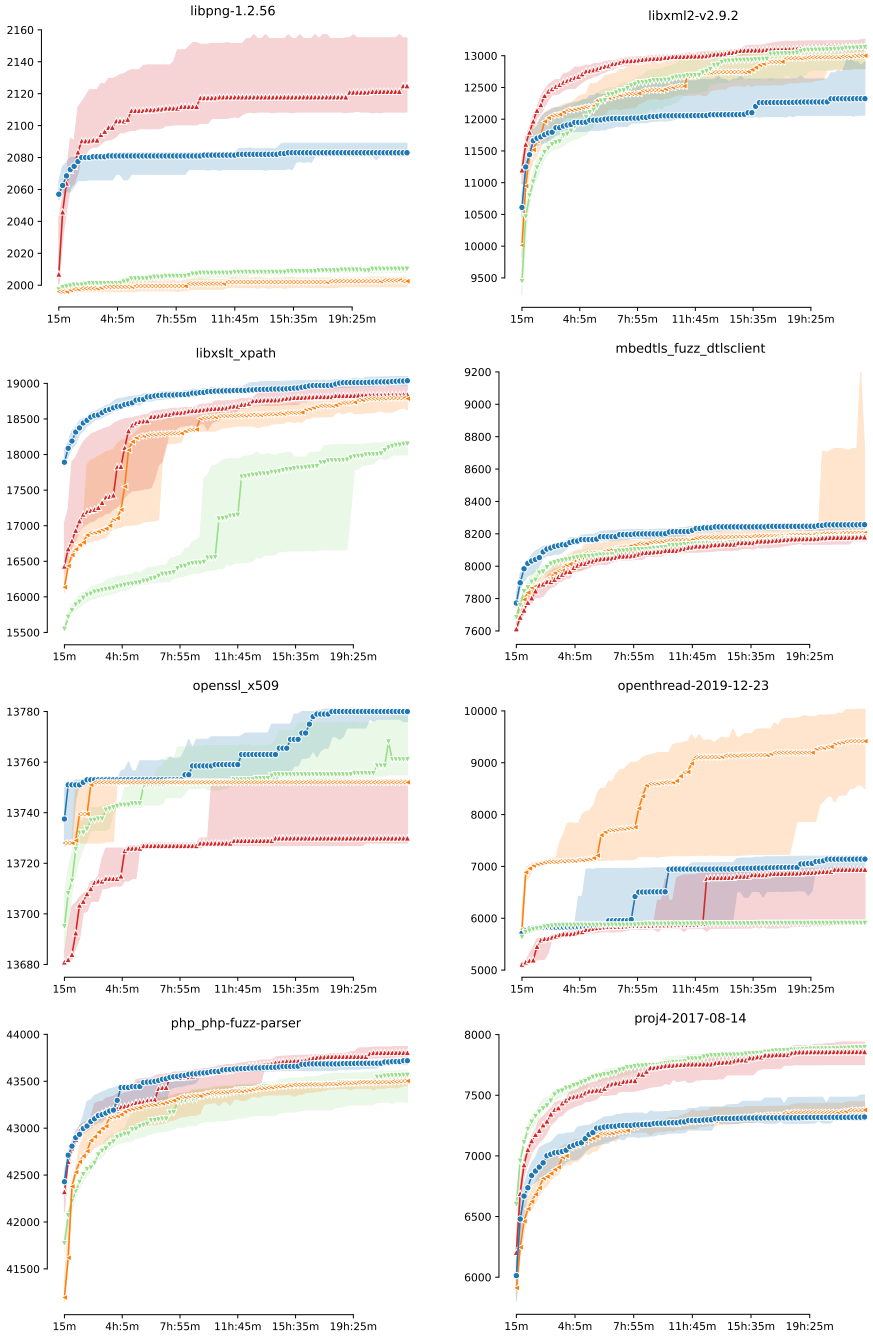
In LibAFL, we designed an interface for power scheduling based on metadata, on top of this, we implemented the six aforementioned algorithms. Our implementation is, however, based on an optimized version of the algorithms integrated into AFL++, which was developed years after the AFLFast paper and that was never evaluated in the literature. Among the six, the most effective[5] are *explore*, the default in AFL, and *coe* and *fast*, the default in AFL++. Therefore, we will focus our tests on these three algorithms and compare them with a baseline (never evaluated before) based on the `plain` algorithm. Table 6.6 shows the results in terms of code coverage.

Overall, considering the average normalized score across all benchmarks, `explore` is the best performer (99.72), with `fast` following (99.43) and then `plain` (98.12) and `coe` (97.08). This results confirm the trend observed in the AFL++ version of the power schedules, with *fast* and *explore* as top performer, with *fast* as the now default schedule in AFL++.

The LibAFL implementations emphasize once again the same observations we did for the corpus scheduling problem. Fast fuzzers (on fast targets) reduce the benefit that can be gained by using more complex scheduling as less useful executions have a limited effect on the throughput. The score of `coe`, which performs worse than the random baseline, can be considered target-specific. In fact, while the algorithm suffered on some targets (particularly on the `bloaty` benchmark), it was the best performer on others (e.g., on a few runs of `mbedtls`).

[5] https://github.com/google/fuzzbench/issues/249#issuecomment-700470906





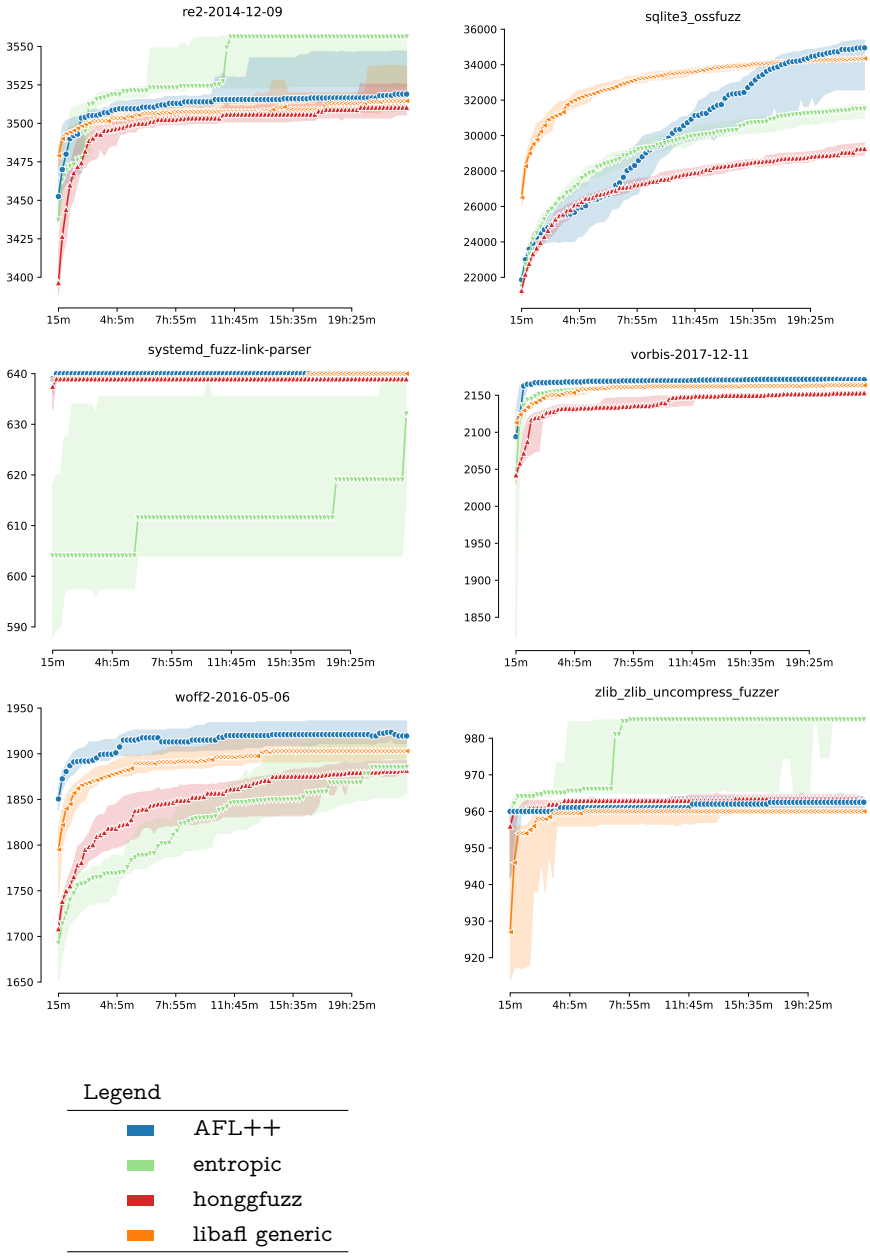


Figure 6.6: Uncovered code coverage over time (23h) of the generic bit-level fuzzer experiment.

6.4.5 A Generic Bit-level Fuzzer

While the main goal of LibAFL is to be a Swiss-army knife to build custom fuzzers, we also strive to provide good default implementations to use for out-of-the-box generic bit-level fuzzing.

In this section, we present a frontend for LibAFL to fuzz LibFuzzer harnesses with a generic mutator. We evaluate this fuzzer against the state-of-the-art fuzzers used in Google OSS-Fuzz to fuzz thousands of open source projects every day, AFL++, HonggFuzz, and LibFuzzer, with the Entropic [25] option enabled for better performance.

Our fuzzer, like LibFuzzer and any other fuzzer used in the previous experiments, uses an in-process executor to run the target harness, while AFL++ and HonggFuzz control the target with forms of IPC (like pipes). Our fuzzer also employs some of the improvements we covered in the previous experiments: cmplog, the weighted corpus scheduler, the *explore* energy assignment scheme, and the MOpt mutator.

For this particular experiment, we submitted a request to the FuzzBench service [6], to execute the four fuzzers on 22 different benchmarks and evaluate the reached code coverage. Each fuzzer was scheduled for 23h and every single experiment was repeated 20 times to mitigate the randomness. The results for each benchmark are reported in Table 6.6. The overall result, based on the average normalized score of the coverage uncovered across all the targets, is that LibAFL clearly outperforms all the other fuzzers with a score of 98.61, followed by HonggFuzz (96.65), AFL++ (96.32) and Entropic (94.22). This result is even more relevant since the other fuzzers were gradually improved over time [116] based on the outcome of several FuzzBench runs, while LibAFL has been developed independently from this benchmarking suite.

By inspecting the results in isolation, we can see that LibAFL shines on 3 benchmarks, `harbfuzz`, `openthread` and `sqlite3`. Notably, it is the only fuzzer that can reach the coverage breakthrough by unlocking the fuzzer from saturation in a few runs on `mbedtls` and consistently on `openthread`.

On the other hand, our fuzzer clearly underperforms AFL++ and HonggFuzz on `libpng`, resulting in an almost equal performance with Entropic. The missing coverage of these two fuzzers can be explained with the execution model that they use, in-process, versus the out-of-process executor that the other two uses. The latter is slower but more reliable in handling timeouts. The strength of our approach is that limitations like this one can

[6] The complete details of the experiment are available at https://www.fuzzbench.com/reports/experimental/2022-04-11-libafl/index.html

be easily overcome by changing a few lines of code in the frontend, in this particular case to move from `InProcessExecutor` to `ForkserverExecutor`.

Overall, these results show that LibAFL is a mature framework capable to be the backbone of a modern generic fuzzer that can compete with state-of-the-art solutions. We foresee the development of a new version of highly customizable but with good defaults generic implementations, fuzzers (like AFL++) based on LibAFL.

6.4.6 Differential Fuzzing

In the previous sections, we presented variants based on coverage-guided fuzzing. However, LibAFL is not limited to code coverage and to its derivates (like context and ngrams [162]), but can also work with other types of feedback. As an example, in this section, we discuss the development of a frontend, inspired by NeoDiff [110], for differential fuzzing of two Ethereum virtual machines.

In short, NeoDiff, originally written in Python, compares the outcome of the executions of two VMs provided with the same input. To do so, it uses a *state hash*, a hash of the registers values, memory, and a probabilistic sampling of the stack at each instruction site of the executed trace. As feedback to evolve its corpus, it uses instead a *type hash*, the hash of the opcodes and the types of the first two items on the stack for each instruction. Any input that generates a trace with a new type hash is added to the corpus.

In LibAFL, NeoDiff can be implemented by taking advantage of the differential executor component, a structure that acts as a proxy to two underlying executors. The type of the inner executors is `CommandExecutor`, a simple kind of executor that spawns a new process given a command line, and its observers are linked to the stdout of such commands, as the Ethereum VMs print their traces in JSON after executing the input bytecode. These observers are processed by a custom feedback, the `TypeHashFeedback` that decodes the trace, computes the type hash, and compares it to the other hashes observed so far in the linked feedback state. A differential feedback, the feedback responsible to compare two observers, is used as objective. The differential feedback uses the state hash to compare the observers linked to the two executors and, if different, considers the input as a solution only if having a novel typehash for de-duplication.

Overall, re-implementing NeoDiff from scratch in LibAFL took 2 working days person and consists of 900 lines of Rust code. We also decided to compare it against the original implementation of NeoDiff, by using the same metric adopted in the original paper, i.e., the number of diffing inputs

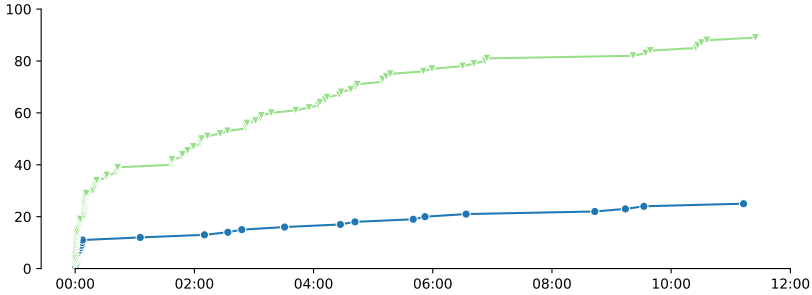


Figure 6.7: Uncovered diffing inputs (unique type hashes) for the original NeoDiff (▬) and the LibAFL version (▬) over 12h.

with unique typehash. We run both fuzzers for 12 hours testing the same `go-ethereum` and `openethereum` versions tested in the original paper. In Figure 6.7 we report the findings over time of both fuzzers, showing that our implementation clearly outperforms the original in this metric. We believe that the bit-level mutators built-in in LibAFL play a major contribution in this experiment.

In terms of unique diffing instructions, the original paper reports that 6 instructions are the causes of the found differences. During our experiment, we reproduced the NeoDiff results finding only 5 instructions over 12h with NeoDiff and 15 instructions with our implementation.

The diffing opcodes in details are (in hex): 3, 31, 38, **3b**, 3c, **3f**, 44, **45**, **46**, 52, 5a, f1, f2, f4, fa. The opcodes found by both the original and our NeoDiff implementation are in bold, the others were discovered only by the LibAFL-based variant.

Our findings are so a superset of the 6 NeoDiff instructions, showing that LibAFL is outperforming the python implementation by a huge margin.

6.4.7 Third-party Applications

During its development, LibAFL had already been adopted to implement several fuzzing frontends by a number of new users, who had no previous experience with our framework. For instance, it has been used to create a symbolic-model-guided fuzzer, TLSPUFFIN [12] that employs a concrete semantic to execute TLS symbolic traces, thus proposing a new approach that mixes fuzzing and model testing. Thanks to this combination, TLSPUFFIN can reach critical protocol states that are impossible to find with classic coverage-guided fuzzing.

A second third-party application of LibAFL is a snapshot fuzzer based

on KVM, TARTIFLETTE [44], which provides a new executor to run a Linux ELF as a VM with system calls emulation and instrumentation facilities for coverage tracking and snapshotting.

Finally, another LIBAFL-based project worth mentioning is BANANAFZZ [87], a fuzzer to detect race conditions with a novel design based on loop-per-thread calls generations.

6.5 Limitations and Future Work

While extensible by design, the current implementation of LIBAFL still lacks some components that are required to implement some specific fuzzing applications.

For instance, at the time of writing LIBAFL CC does not include Link Time Optimization passes to reason about the whole program Control Flow Graph. This type of instrumentation is required to implement most of the directed fuzzing approaches [26, 33, 123] and thus, LIBAFL is not currently providing any directed fuzzing application. This limitation, however, is not intrinsic to our design and support for directed fuzzing will be integrated in the near future.

A powerful feature integrated into LIBAFL is the concolic tracing API, which can be used to extend SYMCC or SYMQEMU with custom constraints filtering and communicate the symbolic trace to a LIBAFL-based fuzzer. Currently, LIBAFL provides a solver stage based on Z3 that generates new testcases as traditional concolic fuzzers do. However, there are two main limitations in traditional concolic fuzzers that our architecture could help to overcome. First, solvers are hard to scale and are both time- and resource-consuming tasks. This could be mitigated by solving symbolic expressions [29] using fuzzing techniques [16, 63, 35]. The other limitation is that fuzzers and concolic engines poorly cooperate. Even when a solver outputs a testcase that solves a complex expression, it is very hard for a generic bit-level fuzzer to mutate and stress the program points related to this testcase without breaking the validity of the solved expressions. Approaches like PANGOLIN [89] go in this direction.

The possibility to build mutators using concolic expressions in LIBAFL allows developers to implement approaches to overcome the mentioned limitations and reproduce previous experiments (e.g., the PANGOLIN artifacts that have never been publicly released). However, none of these have been implemented in LIBAFL yet.

Finally, a core principle of LIBAFL is scalability. Therefore, an interesting future work would be to evaluate different fuzzing synchronization

approaches in terms of scalability. LibAFL already implements an event manager capable of, if the target permits, scale linearly over multiple cores and machines. It also provides an alternative AFL-like disk-based method to synchronize testcases over nodes. An interesting research question is to measure how different approaches like TCP connections or shared memory-based communication affect fuzzing and to pinpoint their trade-offs.

Chapter 7

A LibAFL-based AFL++ Prototype

The Fuzzing community is very active and prolific, with an always growing number of proposed ideas and prototypes [111]. In practice, however, for generic fuzzing there are three main engines that are widely used. These are AFL++ [65], which is gradually replacing AFL [180], LibFuzzer [105] and HonggFuzz [158].

As a spin-off of AFL++, over the last two years – as presented in Chapter 6 – we developed a new fuzzing framework to cope with the extensibility problem of this widely used, but monolithic fuzzer. This framework, LibAFL [66] is not a tool by itself but rather a collection of Rust libraries to write fuzzers. While power users appreciate the flexibility of writing custom fuzzers with LibAFL, most users still prefer AFL++ for its out-of-the-box experience that fits most use-cases well.

To bridge the gap between our two projects, and solve the problem of adding additional fuzzing algorithms to AFL++, we plan to rewrite AFL++ as a frontend of LibAFL. While this process will take many months, we can already provide a LibAFL-based fuzzer that mimics AFL++, just without the many command line options the main project provides to customize the fuzzing behavior.

To challenge this new fuzzer, we submitted it to the first fuzzing com- petition at the 16th International Workshop on Search-Based and Fuzz Testing (SBFT) [100] and it arrived first in the bug-finding category.

7.1 AFL++ on FuzzBench

Our goal is to faithfully replicate the configuration of AFL++ that is used in FuzzBench [116], as it is the best performer in a generic setup.

This configuration consists of the modified SanitizerCoverage `trace-pc-guard` module shipped in AFL++ as part of the LLVM-based instrumentation with `afl-clang-fast`. Here, the AFL++ edge coverage logging routine is inlined in the LLVM IR, while still using the guards generated with `trace-pc-guard`. This pass breaks the direct edges of the Control Flow Graph in each function, inserting an intermediate basic block and this allows to precisely count the edges. The coverage is non-colliding: AFL++ adapts the size of the shared coverage map to the number of instrumented edges, and thus it doesn't suffer from the well-known collision problem.

The other enabled option is the compilation of a secondary binary with the CmpLog instrumentation. The second binary logs the content of each comparison instruction and each routine with two pointers as arguments in a shared memory region, from which the fuzzer can read. AFL++ can use this runtime information to run an enhanced version of the RedQueen [16] mutator.

The last option, `dict2file`, extract the constants operators used in comparisons-related functions such as `strcpy` during compilation. These tokens are then used to build a dictionary for the fuzzer.

7.2 Implementing AFLrustrust

As described in the LibAFL paper [66], several components must be defined to build a fuzzer based on LibAFL.

Executor To emulate the AFL++ behaviour, the executor that must be used is a forkserver. The forkserver implementation in LibAFL supports the binaries compiled with `afl-clang-fast` from AFL++ and their advanced features such as shared memory input delivery and CmpLog instrumentation.

Feedback The feedback used is the maximization of each entry of a coverage map. The coverage map is the one created by the AFL++ target binary. It is exposed to LibAFL with a shared-memory-based map observer. As objective, we simply consider every crashing input.

Mutator The mutator is based on MOpt and schedule two sets of operations available in LibAFL, the bit-level havoc mutations and the token-based ones in order to use an user-supplied or autogenerated dictionary – with `dict2file` in this case.

Scheduler The next input to fuzz is chosen by reusing the same algorithms of AFL++, reimplemented in LibAFL. Corpus culling is done by selecting a minimal set of testcases covering every edge seen so far with a weighted prioritization based on the testcase length and the execution time. The selection from this pool of testcases is then performed with the AFL++ weighted scheduler using the *explore* energy assignment scheme [27].

Stages The stages that compose the fuzzer are four, starting with *calibration*, the stage used to measure stats about the current testcase such as stability and average execution time. Then, a tracing stage is used to run the target under a second forkserver executor with a CmpLog enabled binary to collect the cmp traces into the fuzzer metadata. After this one, and depending on it, there is the *input-to-state* stage that uses the CmpLog metadata to match tokens in the input with the various I2S mutators. In the end, a mutational stage with the classic havoc mutations is used and the energy is assigned using the power schedule from the metadata generated with the weighted scheduler.

7.3 SBST'23 Competition Results

The SBST'23 fuzzing competition [100] is composed of two experiments in which AFLrustrust was evaluated versus 11 fuzzers, AFL++ included.

The first experiment [1] is coverage-based in which the fuzzers are compared using the uncovered branch coverage over 23 hours on 38 different programs from OSS-Fuzz [3].

In terms of average rank, AFLrustrust placed 5th, and in terms of average normalized score[2] it took the 4th position. In both the ranking, it was really close to AFL++ which took a position in front of our tool in both the scores.

[1] https://storage.googleapis.com/www.fuzzbench.com/reports/experimental/SBFT23/Final-Coverage/index.html

[2] This score is based on the average of per-benchmark scores, where the score represents the percentage of the highest reached median code coverage on a given benchmark.

The second experiment [3] is bugs-based in which the fuzzers are evaluated in their ability to discover crashes that are linked to bugs in 15 vulnerable applications from OSS-Fuzz during 23 hours runs.

In terms of average rank, AFLrustrust wins the first position this time and the second place, with a tie in the score with the first fuzzer Pastis which is a hybrid fuzzer while our approach is purely based on coverage-guided fuzzing, in the average normalized score [4] ranking.

While the performance on the bugs-based dataset seems great, we believe that our tool was penalized in the coverage-based experiment due to the score 0 assigned to the `zlib_zlib_uncompress_fuzzer` benchmark. For an unknown reason, the fuzzer failed to start the experiment with this target program but we could not replicate the failure locally as AFLrustrust builds and runs fine this zlib fuzzer.

7.4 Discussion

The proposed fuzzer based on LibAFL is a first attempt at the upcoming rewriting of AFL++ as a frontend of LibAFL and thus may be incomplete or even with buggy components. In security research, often the best fuzzer is not the one that finds slightly more coverage than the others but the one that fits the user needs, and LibAFL aims at this. On the other hand, beginners and developers need an off-the-shelf environment that can fuzz a target with a minimal setup, as AFL does, and an AFL++ clone based on LibAFL can provide the best of both worlds.

The results of the fuzzing competition shows that AFLrustrust shines even if it is just a prototype. The lack of uncovered coverage compared to the original tool, AFL++, is due to the missing implementation of the input-to-state mutator in the same advanced way as AFL++ which includes advancements from RedQueen [16], Weizz [63] and others original algorithms to approximately solve branch constraints developed during the years. This is now a work in progress in LibAFL and an equivalent mutator are under development and in the roadmap of the rewrite of AFL++ on top of LibAFL.

[3] https://storage.googleapis.com/www.fuzzbench.com/reports/experimental/SBFT23/Final-Bug/index.html

[4] This ranking is based on the average of per-benchmark scores, where the score represents the percentage of the highest reached median bug coverage on a given benchmark.

Chapter 8

Conclusion

In this chapter, we're bringing together the findings from the different contributions of the thesis to highlight the latest trends and potential future steps in the fuzz testing field. During this journey, we investigated the benefits of using likely invariants violations and data dependency graph edges as feedbacks, dived deep into the details of the American Fuzzy Lop project, and introduced a new, flexible fuzz testing framework, LibAFL. Here, we outline the achievements of our research and the future directions to further investigate the challenges that this thesis proposed to address.

First off, using likely invariants as feedback for fuzzers brings novel ideas to better abstract, and in turn explore program states. We argue that some bugs may be readily discovered by taking into account program state conditions that control flow alone does not entail, accessing seldom-explored corner cases where vulnerabilities may lie undetected for a long time. We achieve this goal without incurring the state explosion problem and with a moderate performance overhead, amortized by an increased number of found bugs. We hope that our work can pave the way for more research on program state approximations to serve as feedback for Fuzz Testing, such as the online likely invariants mining module for fuzzing proposed in §3.4 but also other approaches not based on invariant violations.

Next up, the second chapter talks about blending information from the Data Dependency Graph with traditional edge coverage to improve fuzzing feedback. Our experiments show that, for applications that have a rich set of data dependencies, this approach leads to the discovery of more and diverse bugs. Moreover, we hope that our technique and prototype based on AFL++ and LLVM will be adopted by users to fuzz programs alongside the existing coverage metrics. We are working on the integration of this feedback into LibAFL, meanwhile another similar approach [84] tracking

data dependencies at a higher level – using dynamically allocated objects – has been already proposed.

Moving on, the third contribution takes a closer look at the AFL project, breaking down its structure and how it impacts fuzz testing features. It's clear that the small details of AFL play a big role in how effective a fuzz testing campaign can be, both in good and bad ways. We confirm the positive effects of some aspects of AFL, such as the novelty search algorithm in §5.3.2, but also the negative impact of others, such as its testcase scoring, in §5.3.4. AFL's prior decisions affect evaluations of new research based on AFL. Researchers who clone and extend AFL need to be aware that AFL's implementation details will impact their research and the outcome of their experiments.

We hope that our study provides useful information for researchers and practitioners who, in the future, will have to work on the previously unevaluated aspects of this fuzzer.

The last part of the thesis presents a novel and completely extensible fuzzing framework, LibAFL. To show its versatility, and the comprehensiveness of its ready-to-use components to build state-of-the-art fuzzers, we present several frontends based on LibAFL and perform experiments with them covering different problems in the fuzzing literature. We highlight the customization allowed by the LibAFL design and the power of the combination of several orthogonal techniques, leading to a fuzzer that outperforms the best publicly available tools. Using LibAFL as a backbone, we can also build a clone of AFL++, AFLRustTrust, the winner of the SBFT'23 fuzzing competition in the bug-finding category. A work-in-progress replacement for LibFuzzer [43] based on LibAFL will enable any fuzzing infrastructure such as cargo-fuzz [18] to replace the LibFuzzer runtime with a new compatible runtime based on LibAFL with modern algorithms that outperform the now deprecated LibFuzzer.

To wrap it up, this thesis highlights the cool progress made in fuzz testing with our research in the past 3 years and calls for even more innovation in this area. It points towards focusing on better feedback systems, a deeper understanding of baselines like AFL, and creating flexible and powerful fuzz testing frameworks. This direction not only strengthens the basics of fuzz testing but also opens up new opportunities, aiming for a future with stronger and more reliable software security. We hope that bringing all these studies together will spark new ideas and developments, pushing even further what we can achieve in fuzz testing.

References

[1] Critical Edges Elimination Pass. https://llvm.org/doxygen/BreakCriticalEdges_8cpp_source.html. [Online; accessed 08 Feb. 2022].

[2] Frida - a world-class dynamic instrumentation framework. https://www.frida.re/. [Online; accessed January 17, 2024].

[3] Google OSS-Fuzz: continuous fuzzing of open source software. https://github.com/google/oss-fuzz. [Online; accessed January 17, 2024].

[4] Kvasir C/C++ front end. https://plse.cs.washington.edu/daikon/download/doc/daikon.html#Kvasir. [Online; accessed January 17, 2024].

[5] Circumventing Fuzzing Roadblocks with Compiler Transformations. https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/, 2016. [Online; accessed January 17, 2024].

[6] Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, 2016. [Online; accessed January 17, 2024].

[7] afl-cov. https://github.com/mrash/afl-cov, Accessed January 17, 2024.

[8] Datadependencegraph class reference in the llvm framework. https://llvm.org/doxygen/classllvm_1_1DataDependenceGraph.html, Accessed January 17, 2024.

[9] Definition of ddg in the llvm framework. https://llvm.org/docs/DependenceGraphs/index.html, Accessed January 17, 2024.

[10] Fuzzbench configuration. https://google.github.io/fuzzbench/, Accessed January 17, 2024.

[11] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques and tools*. 2020.

[12] Maximilian Ammann. Symbolic-Model-Guided Fuzzing of Cryptographic Protocols. Master's thesis, University of Augsburg, Institute for Software & Systems Engineering, 2021. *See https://github.com/tlspuffin/tlspuffin.*

[13] Cornelius Aschermann, Tommaso Frassetto, T. Holz, Patrick Jauernig, A. Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, 2019.

[14] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612. IEEE, 2020.

[15] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. IJON: Exploring deep state spaces via fuzzing. In *IEEE Symposium on Security and Privacy (Oakland)*, 2020.

[16] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: fuzzing with input-to-state correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS*, 2019.

[17] Giorgio Ausiello, Camil Demetrescu, Irene Finocchi, and Donatella Firmani. K-calling context profiling. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 867–878, New York, NY, USA, 2012. Association for Computing Machinery.

[18] Rust Fuzzing Authority. cargo-fuzz: Command line helpers for fuzzing. https://github.com/rust-fuzz/cargo-fuzz. [Online; accessed January 17, 2024].

[19] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 51(3):50:1–50:39, 2018.

[20] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA, 2005.

[21] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[22] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1985–2002, Santa Clara, CA, aug 2019. USENIX Association.

[23] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. AURORA: Statistical crash analysis for automated root cause explanation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 235–252. USENIX Association, aug 2020.

[24] Rastislav Bodik and Rajiv Gupta. Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 159–170, 1997.

[25] Marcel Böhme, Valentin Manès, and Sang Kil Cha. Boosting fuzzer efficiency: An information theoretic perspective. In *Proceedings of the 14th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 1–11, 2020.

[26] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2329–2344, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as Markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. Association for Computing Machinery.

[28] Pietro Borrello, Andrea Fioraldi, Daniele Cono D'Elia, Davide Balzarotti, Leonardo Querzoni, and Cristiano Giuffrida. Predictive Context-sensitive Fuzzing. In *Network and Distributed System Security (NDSS) Symposium*, NDSS 24, February 2024.

[29] Luca Borzacchiello, Emilio Coppa, and Camil Demetrescu. Fuzzing Symbolic Expressions. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, 2021.

[30] Marcel Böhme and Soumya Paul. A probabilistic analysis of the efficiency of automated software testing. *IEEE Transactions on Software Engineering*, 42(4):345–360, 2016.

[31] Qiong Cai, Lin Gao, and Jingling Xue. Region-based partial dead code elimination on predicated code. In *International Conference on Compiler Construction*, pages 150–166. Springer, 2004.

[32] S. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, Los Alamitos, CA, USA, may 2012. IEEE Computer Society.

[33] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2095–2108, New York, NY, USA, 2018. Association for Computing Machinery.

[34] Jian-Liang Chen, Feng-Jian Wang, and Yung-Lin Chen. An object-oriented dependency graph for program slicing. In *Proceedings. Technology of Object-Oriented Languages. TOOLS 24 (Cat. No. 97TB100240)*, pages 121–130. IEEE, 1997.

[35] P. Chen and H. Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[36] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. MEUZZ: Smart seed scheduling for hybrid fuzzing. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 77–92, 2020.

[37] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441. IEEE, 2018.

[38] Animesh Basak Chowdhury, Raveendra Kumar Medicherla, and R Venkatesh. Verifuzz: Program aware fuzzing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 244–249. Springer, Cham, 2019.

[39] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[40] Code Intelligence. Jazzer - Coverage-guided, in-process fuzzing for the JVM. https://github.com/CodeIntelligenceTesting/jazzer, 2021. [Online; accessed January 17, 2024].

[41] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, auf 1992.

[42] Marco Cova, Davide Balzarotti, Viktoria Felmetsger, and Giovanni Vigna. Swaddler: An approach for the anomaly-based detection of state violations in web applications. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 63–86, Queensland, Australia, September 5–7, 2007.

[43] Addison Crump, Andrea Fioraldi, Dominik Maier, and Dongjia Zhang. LibAFL_libfuzzer: Libfuzzer on Top of LibAFL. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 70–72, 2023.

[44] Tanguy Dubroca César Belley. Tartiflette: Snapshot fuzzing with KVM and libAFL. https://www.lse.epita.fr/lse-winter-days-2021/slides/lse_winter_days_tartiflette.pdf, 2021. [Online; accessed January 17, 2024].

[45] DARPA. Cyber Grand Challenge (CGC) (Archived). https://www.darpa.mil/program/cyber-grand-challenge. [Online; accessed January 17, 2024].

[46] Thomas Defard, Aleksandr Setkov, Angelique Loesch, and Romaric Audigier. Padim: a patch distribution modeling framework for anomaly detection and localization, 2020.

[47] Daniele Cono D'Elia, Emilio Coppa, Simone Nicchi, Federico Palmaro, and Lorenzo Cavallaro. SoK: Using dynamic binary instrumentation for security (and how you may get caught red handed). In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 15–27, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. Mining hot calling contexts in small space. *Software: Practice and Experience*, 46(8):1131–1152, 2016.

[49] Jared D. DeMott and R. Enbody. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. Black Hat USA, 2007.

[50] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.Ng: A unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, page 131–141, New York, NY, USA, 2017. Association for Computing Machinery.

[51] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121. IEEE, 2016.

[52] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 839–850. ACM, 2013.

[53] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 60–71. IEEE Press, 2019.

[54] Gregory J Duck, Xiang Gao, and Abhik Roychoudhury. Binary rewriting without control flow recovery. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 151–163, 2020.

[55] M. Eddington. Peach fuzzing platform. https://web.archive.org/web/20180621074520/http://community.peachfuzzer.com/WhatIsPeach.html. [Online; accessed January 17, 2024].

[56] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[57] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, dec 2007.

[58] Michael Dean Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, USA, 2000. AAI9983472.

[59] Brandon Falk. aflbench. https://github.com/gamozolabs/aflbench, 2020. [Online; accessed January 17, 2024].

[60] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[61] G. Fink and K. Levitt. Property-based testing of privileged programs. In *Tenth Annual Computer Security Applications Conference*, pages 154–163, 1994.

[62] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, aug 2021.

[63] Andrea Fioraldi, Daniele Cono D'Elia, and Emilio Coppa. WEIZZ: Automatic grey-box fuzzing for structured binary formats. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Soft-*

ware Testing and Analysis, ISSTA 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[64] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. Fuzzing binaries for memory safety errors with QASan. In *2020 IEEE Secure Development Conference (SecDev)*, 2020.

[65] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, aug 2020.

[66] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A framework to build modular and reusable fuzzers. 2022.

[67] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. Dissecting american fuzzy lop: A fuzzbench evaluation. *ACM Trans. Softw. Eng. Methodol.*, 32(2), mar 2023.

[68] Ivan Fratric. TinyInst. https://github.com/googleprojectzero/TinyInst. [Online; accessed January 17, 2024].

[69] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594. USENIX Association, aug 2020.

[70] C. Giuffrida, L. Cavallaro, and A.S. Tanenbaum. Practical automated vulnerability monitoring using program state invariants. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12. IEEE CS, 2013.

[71] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'08, 2008.

[72] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 50–59, Piscataway, NJ, USA, 2017. IEEE Press.

[73] Google Inc. libprotobuf-mutator. https://github.com/google/libprotobuf-mutator. [Online; accessed January 17, 2024].

[74] Alex Groce and John Regehr. The Saturation Effect in Fuzzing. https://blog.regehr.org/archives/1796. [Online; accessed January 17, 2024].

[75] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. FUZZILLI: fuzzing for javascript JIT compiler vulnerabilities. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[76] Christian Hammer. *Information flow control for Java: a comprehensive approach based on path conditions in dependence graphs*. 2009.

[77] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-aware code generation to find vulnerabilities in javascript engines. In *ndss*, 2019.

[78] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 291–301, New York, NY, USA, 2002. Association for Computing Machinery.

[79] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, 3(03):243–250, may 1977.

[80] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), nov 2020.

[81] Mark Heffernan and Kent Wilken. Data-dependency graph transformations for instruction scheduling. *Journal of Scheduling*, 8(5):427–451, 2005.

[82] Mark Heffernan, Kent Wilken, and Ghassan Shobaki. Data-dependency graph transformations for superblock scheduling. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 77–88. IEEE, 2006.

[83] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. Seed selection for successful fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, page 230–243, New York, NY, USA, 2021. Association for Computing Machinery.

[84] Adrian Herrera, Mathias Payer, and Antony L. Hosking. Dataflow: Toward a data-flow-guided fuzzer. *ACM Trans. Softw. Eng. Methodol.*, mar 2023. Just Accepted.

[85] Jesse Hertz and Tim Newsham. Project triforce: Run afl on everything! [Online; accessed January 17, 2024].

[86] Marc Heuse. afl-clang-lto - collision free instrumentation at link time. https://github.com/AFLplusplus/AFLplusplus/blob/stable/instrumentation/README.lto.md, 2020. [Online; accessed January 17, 2024].

[87] Peter Hlavaty. bananafzz. https://github.com/rezerodai/bananafzz, 2022. [Online; accessed January 17, 2024].

[88] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA, aug 2012. USENIX Association.

[89] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. Pangolin: Incremental hybrid fuzzing with polyhedral path abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1613–1627, Los Alamitos, CA, USA, may 2020. IEEE Computer Society.

[90] Google Inc. Atheris: A Coverage-Guided, Native Python Fuzzer. https://github.com/google/atheris. [Online; accessed January 17, 2024].

[91] David A Kinloch and Malcolm Munro. Understanding c programs using the combined c graph representation. In *ICSM*, pages 172–180, 1994.

[92] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[93] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Partial dead code elimination. *ACM SIGPLAN Notices*, 29(6):147–158, 1994.

[94] David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[95] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[96] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[97] Chingren Lee, Jenq Kuen Lee, and TingTing Hwang. Compiler optimization on instruction scheduling for low power. In *Proceedings 13th International Symposium on System Synthesis*, pages 55–60. IEEE, 2000.

[98] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 475–485, New York, NY, USA, 2018. Association for Computing Machinery.

[99] Yu Liang, Song Liu, and Hong Hu. Detecting logical bugs of DBMS with coverage-based guidance. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4309–4326, Boston, MA, August 2022. USENIX Association.

[100] D. Liu, J. Metzman, M. Bohme, O. Chang, and A. Arya. Sbft tool competition 2023 - fuzzing track. In *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, pages 51–54, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[101] LLVM. SanitizerCoverage. https://clang.llvm.org/docs/SanitizerCoverage.html. [Online; accessed January 17, 2024].

[102] LLVM. SanitizerCoverage - Edge coverage. https://clang.llvm.org/docs/SanitizerCoverage.html#edge-coverage. [Online; accessed January 17, 2024].

[103] LLVM Project. LibFuzzer - Value Profile. https://llvm.org/docs/LibFuzzer.html#value-profile. [Online; accessed January 17, 2024].

[104] LLVM Project. [libFuzzer] Port to Windows. https://reviews.llvm.org/D51022. [Online; accessed January 17, 2024].

[105] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, sep 2018. [Online; accessed January 17, 2024].

[106] Shan Lu, Pin Zhou, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. Pathexpander: Architectural support for increasing the path coverage of dynamic bug detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 38–52. IEEE, 2006.

[107] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, aug 2019. USENIX Association.

[108] David R. MacIver, Zac Hatfield-Dodds, and Many Other Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4(43):1891, 2019.

[109] Dominik Maier, Otto Bittner, Marc Munier, and Julian Beier. Fitm: Binary-only coverage-guided fuzzing for stateful network protocols. In *Workshop on Binary Analysis Research (BAR), 2022*, 2022.

[110] Dominik Maier, Fabian Fäßler, and Jean-Pierre Seifert. Uncovering smart contract vm bugs via differential fuzzing. In *Reversing and Offensive-Oriented Trends Symposium*, ROOTS'21, pages 11–22, New York, NY, USA, 2021. Association for Computing Machinery.

[111] V. Manes, H. Han, C. Han, S.K. Cha, M. Egele, E. J. Schwartz, and M. Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, (01), oct 5555.

[112] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. Ankou: Guiding grey-box fuzzing towards combinatorial difference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pages 1024–1036, New York, NY, USA, 2020. Association for Computing Machinery.

[113] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. *Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing*, page 701–712. Association for Computing Machinery, New York, NY, USA, 2020.

[114] Masahiro Matsubara, Kohei Sakurai, Fumio Narisawa, Masushi Enshoiwa, Yoshio Yamane, and Hisamitsu Yamanaka. Model checking with program slicing based on variable dependence graphs. *arXiv preprint arXiv:1301.0041*, 2013.

[115] Raveendra Kumar Medicherla, Raghavan Komondoor, and Abhik Roychoudhury. *Fitness Guided Vulnerability Detection with Greybox Fuzzing*, page 513–520. Association for Computing Machinery, New York, NY, USA, 2020.

[116] Jonathan Metzman, László Szekeres, Laurent Maurice Romain Simon, Read Trevelin Sprabery, and Abhishek Arya. Fuzzbench: An open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2021.

[117] B. Miller, M. Zhang, and E. Heymann. The relevance of classic fuzz testing: Have we solved this one? *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[118] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.

[119] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, San Diego, United States, 02 2018.

[120] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. Binary-level directed fuzzing for use-after-free vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 47–62, San Sebastian, oct 2020. USENIX Association.

[121] Keean Schupke Oleg Kiselyov, Ralf Laemmel. HList: Heterogeneous lists. https://hackage.haskell.org/package/HList, 2004. [Online; accessed January 17, 2024].

[122] Tavis Ormandy. This shouldn't have happened: A vulnerability postmortem. https://googleprojectzero.blogspot.com/2021/12/this-shouldnt-have-happened.html. [Online; accessed January 17, 2024].

[123] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *USENIX Security*, aug 2020.

[124] Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. *ACM Sigplan Notices*, 19(5):177–184, 1984.

[125] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 329–340, New York, NY, USA, 2019. Association for Computing Machinery.

[126] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. FuzzFactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[127] Karthik Pattabiraman, Giancinto Paolo Saggese, Daniel Chen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing*, 8(5):640–655, 2010.

[128] Mathias Payer. The fuzzing hype-train: How random testing triggers thousands of crashes. *IEEE Security and Privacy*, 17(1):78–82, 2019.

[129] H. Peng, Y. Shoshitaishvili, and M. Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, May 2018.

[130] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 2019.

[131] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet: A greybox fuzzer for network protocols. In *Proceedings of the 13rd IEEE International Conference on Software Testing, Verification and Validation : Testing Tools Track*, 2020.

[132] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with symcc: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198. USENIX Association, aug 2020.

[133] Sebastian Poeplau and Aurélien Francillon. SymQEMU: Compilation-based symbolic execution for binaries. In *Network and Distributed System Security Symposium*. Network & Distributed System Security Symposium, February 2021.

[134] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers Presented at the December 1-3, 1959, Eastern Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '59 (Eastern), pages 133–138, New York, NY, USA, 1959. Association for Computing Machinery.

[135] Chenxiong Qian, Xiapu Luo, Yu Le, and Guofei Gu. Vulhunter: toward discovering vulnerabilities in android applications. *IEEE Micro*, 35(1):44–53, 2015.

[136] Fernando Magno Quintao Pereira, Raphael Ernani Rodrigues, and Victor Hugo Sperle Campos. A fast and low-overhead technique to secure programs against integer overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, USA, 2013. IEEE Computer Society.

[137] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012*

7th International Workshop on Automation of Software Test (AST), pages 36–42. IEEE, 2012.

[138] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *24th Annual Network and Distributed System Security Symposium, NDSS*, 2017.

[139] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[140] Bob Joyce Reed Hastings. Purify: Fast detection of memory leaks and access errors. In *In proc. of the winter 1992 usenix conference*. Citeseer, 1991.

[141] Marc Roper. Software Testing, 1994.

[142] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. Association for Computing Machinery.

[143] Swarup Kumar Sahoo, John Criswell, Chase Geigle, and Vikram Adve. Using likely invariants for automated software fault localization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 139–152, New York, NY, USA, 2013. Association for Computing Machinery.

[144] Christopher Salls, Chani Jindal, Jake Corina, Christopher Kruegel, and Giovanni Vigna. Token-Level fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2795–2809. USENIX Association, aug 2021.

[145] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, Vancouver, B.C., aug 2021. USENIX Association.

[146] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. KAFL: Hardware-assisted feedback

fuzzing for OS kernels. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 167–182, USA, 2017. USENIX Association.

[147] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: Network fuzzing with incremental snapshots. *arXiv preprint arXiv:2111.03013*, 2021.

[148] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIXATC 12)*, pages 309–318, 2012.

[149] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, page 28. USENIX Association, 2012.

[150] Kostya Serebryany. OSS-Fuzz - google's continuous fuzzing service for open source software. Vancouver, BC, August 2017. USENIX Association.

[151] Chayanika Sharma, Sangeeta Sabharwal, and Ritu Sibal. A survey on software testing techniques using genetic algorithm. *arXiv preprint arXiv:1411.1154*, 2014.

[152] Anand Sivasubramaniam, Aman Singla, Umakishore Ramachandran, and H. Venkateswaran. An approach to scalability study of shared memory parallel systems. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, pages 171–180, New York, NY, USA, 1994. Association for Computing Machinery.

[153] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. *NDSS*, 2016.

[154] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz. SoK: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295, 2019.

[155] Prashast Srivastava and Mathias Payer. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA

2021, page 244–256, New York, NY, USA, 2021. Association for Computing Machinery.

[156] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[157] Harmen-Hinrich Sthamer. *The automatic generation of software test data using genetic algorithms*. PhD thesis, University of Glamorgan, 1995.

[158] Robert Swiecki. Honggfuzz. https://github.com/google/honggfuzz. [Online; accessed January 17, 2024].

[159] Fabian Toepfer and Dominik Maier. Bsod: Binary-only scalable fuzzing of device drivers. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 48–61, 2021.

[160] Guido Vranken. Cryptofuzz - differential cryptography fuzzing. https://github.com/guidovranken/cryptofuzz, 2019. [Online; accessed January 17, 2024].

[161] Dmitry Vyukov. syzkaller - kernel fuzzer. [Online; accessed January 17, 2024].

[162] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, Chaoyang District, Beijing, sep 2019. USENIX Association.

[163] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *NDSS*, 2021.

[164] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[165] Lei Wang, Qiang Zhang, and PengChao Zhao. Automated detection of code vulnerabilities based on program analysis and model checking. In

2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, pages 165–173. IEEE, 2008.

[166] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *NDSS*, 2020.

[167] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[168] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2313–2328, New York, NY, USA, 2017. Association for Computing Machinery.

[169] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604. IEEE, 2014.

[170] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812. IEEE, 2015.

[171] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[172] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang. SLF: Fuzzing without valid seed inputs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 712–723, 2019.

[173] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324. USENIX Association, aug 2020.

[174] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 745–761, USA, 2018. USENIX Association.

[175] Michał Zalewski. American Fuzzy Lop. https://lcamtuf.coredump.cx/afl/. [Online; accessed January 17, 2024].

[176] Michał Zalewski. Binary fuzzing strategies: what works, what doesn't. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html, 2014. [Online; accessed January 17, 2024].

[177] Michał Zalewski. Fuzzing random programs without execve(). https://lcamtuf.blogspot.com/2014/10/fuzzing-binaries-without-execve.html, 2014. [Online; accessed January 17, 2024].

[178] Michał Zalewski. Pulling JPEGs out of thin air. https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html, 2014. [Online; accessed January 17, 2024].

[179] Michał Zalewski. afl-fuzz: making up grammar with a dictionary in hand. https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html, 2015. [Online; accessed January 17, 2024].

[180] Michał Zalewski. American Fuzzy Lop - Whitepaper. https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016. [Online; accessed January 17, 2024].

[181] Michał Zalewski. Bunny the Fuzzer. https://code.google.com/archive/p/bunny-the-fuzzer/, 2016. [Online; accessed January 17, 2024].

[182] Michał Zalewski. "FidgetyAFL" implemented in 2.31b. https://groups.google.com/g/afl-users/c/1PmKJC-EKZo/m/zck6Iu77DgAJ, 2016. [Online; accessed January 17, 2024].

[183] Zeineb Zhioua, Stuart Short, and Yves Roudier. Static code analysis for software security verification: Problems and approaches. In *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, pages 102–109. IEEE, 2014.