To Infinity, and Beyond (Coverage)

THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the registrar's office.

Doctoral Thesis
by

Ahmad Hazimeh

presented on 30th November, 2023
at EPFL, Lausanne

Approved by the examining committee:

Prof. Haitham Al Hassanieh, *Jury President*
Prof. Mathias Payer, *Thesis Director*
Prof. Clément Pit-Claudel, *Internal Examiner*
Dr. Eleonora Losiouk, *External Examiner*
Dr. Marcel Böhme, *External Examiner*

> You're my namesake, Dr. Ahmad...
>
> *Jeddo, 2011*

To my late grandfather (Ahmad H. Hazimeh).
To my past and future self.
To my family.

Acknowledgments

This work would not have been possible without the support and guidance of my advisor, Mathias Payer, to whom I express my thanks and gratitude for giving me the chance to pursue and finish my PhD. I extend my gratitude to the jury members, Prof. Haitham Al Hassanieh, Prof. Clément Pit-Claudel, Dr. Eleonora Losiouk, and Dr. Marcel Böhme for generously offering their time to serve on my committee and for providing their invaluable insights and feedback.

I dedicate a special thanks to Natascha Fontana for seamlessly orchestrating a lab as sizable as HexHive, for organizing lab lunches and retreats, and for providing administrative assistance on every occasion. My appreciation goes out as well to the members and collaborators of the Hive, past and present, with whom I shared countless academic discussions and friendly conversations over the past five years.

I am sincerely grateful to all those who have been instrumental in the completion of this thesis. The magnitude of your contributions cannot be quantified, and your continual guidance has been a cornerstone of this work.

Throughout this journey, I met a lot of people in Lausanne whom I'm fortunate to have as friends, and to whom I'm thankful for the happy moments and lasting memories we made together.

I'd like to thank the Lebanese gang, who took me in since day one in Switzerland: Hussein, Hala, Mahdi, Zeinab, Rania, Taha, Mira, Youssof, Wajeb, and Maaz. Thank you for all the hikes, the game nights, and the road trips. Special thanks go out to Hadi, Hasan, and Mohammad as well, for all the stress-free hangouts and the thrilling ski trips.

I'm also thankful to Las Chicas, a small group of friends who hold a big place in my heart, for putting a smile on my face in the good times and the bad. Thank you Karen, my EPIC buddy, Elsa, my matchmaker, and Virginia, my endless source of boomer humor. I owe a huge debt of gratitude to them for helping me pull through the hardships of life and work abroad, for being there whenever I needed them, and for taking care of the little ones 🐱.

I am most indebted to my partner and my best friend, Sarah, for her boundless love, patience, and support. I would like to extend my appreciation to her family, Grossmami Elsbeth, Barbara, Pascal, Fiona, and Valerie, for welcoming me into their lives and being my home away from home.

Finally, I want to express my endless gratitude to my family, namely my parents, Ali and Hala, and my sisters, Rim and Rana, for their unconditional love and support. Without them, I would not be where I am, who I am, today. I dedicate this thesis to them.

Lausanne, 30th November, 2023 Ahmad Hazimeh

Abstract

The pursuit of software security and reliability hinges on the identification and elimination of software vulnerabilities, a challenge compounded by the vast and evolving complexity of modern systems. Fuzzing has emerged as an indispensable technique for bug discovery, owing to its ability to automate the rapid generation and execution of test cases. However, its effectiveness is constrained by the quality of metrics employed for evaluation and optimization. This dissertation posits that for effective bug discovery in increasingly complex systems, fuzzing techniques must employ tailored metrics that capture application-specific features such as state and semantics.

The dissertation first addresses a glaring gap in the fuzzing landscape—the lack of standardized benchmarks for fuzzer evaluation—through MAGMA, a fuzzer benchmark with ground-truth metrics. MAGMA enables objective assessment of fuzzer performance across diverse software targets by leveraging real-world bugs, instrumented to provide bug-centric metrics. Through rigorous experiments, set up over 200,000 CPU-hours and involving state-of-the-art mutation-based fuzzers, MAGMA highlights the limitations of using crash counts as the de facto evaluation metric, and provides a unified platform for an accurate evaluation and comparison of fuzzers.

The second project, IGOR, addresses the inefficacy of crash de-duplication techniques, which typically suffer from bug-count inflation and conflation. Igor builds on the insight that a bug cannot be triggered without executing its code. Through a process of test case minimization and execution trace matching, we introduce a metric for crash de-duplication that goes beyond code coverage and call stacks. By employing control-flow graph similarity comparisons over minimized execution traces, IGOR demonstrates its capability to accurately group crashes, reducing "unique" bug counts by an order of magnitude compared to existing techniques.

The penultimate project, TANGO, addresses the inadequacy of traditional code coverage metrics in exploring the state spaces of complex systems like language parsers and video games. By incorporating "state" as a first-class citizen, TANGO enhances the fuzzer's ability to navigate complex systems. State inference led to the discovery of previously undetected bugs, and it also highlighted a novel observation: code coverage is insufficient for describing state. Through our evaluation, TANGO reveals that fuzzers which rely solely on code coverage could potentially spend upwards of 80% of their time duplicating their efforts in the face of stateful targets.

Finally, SENSEI aims to help fuzzers at incrementally exploring targets by fuzzing parsers in isolation and using those results to bootstrap the fuzzing of more complex targets. By leveraging the rich and specific domain knowledge encoded in Wireshark dissectors, SENSEI incorporates parser-specific metrics to guide fuzzing in the direction of high-quality and diverse inputs, to aid the exploration of network protocol implementations.

Collectively, these projects introduce novel metrics and methodologies for fuzzer development and evaluation. They provide empirical evidence supporting the thesis that tailored metrics are key to effective and successful fuzzing.

Résumé

La recherche de sécurité et de fiabilité logicielle repose sur l'identification et l'élimination des vulnérabilités logicielles, un défi exacerbé par la vaste et évolutive complexité des systèmes modernes. Le fuzzing est apparu comme une technique indispensable pour la découverte de bugs, grâce à sa capacité d'automatiser la génération et l'exécution rapide de cas de test. Cependant, son efficacité est limitée par la qualité des métriques utilisées pour l'évaluation et l'optimisation. Cette thèse postule que, pour une découverte efficace de bugs dans des systèmes de plus en plus complexes, les techniques de fuzzing doivent utiliser des métriques sur mesure qui capturent des caractéristiques spécifiques aux applications telles que l'état et la sémantique.

La dissertation commence par mettre en lumière un manque flagrant dans le paysage du fuzzing—l'absence de benchmarks standardisés pour l'évaluation des fuzzers—à travers MAGMA, un benchmark de fuzzer avec des métriques de vérité terrain. MAGMA permet une évaluation objective de la performance des fuzzers sur diverses cibles logicielles en exploitant des bugs réels, instrumentés pour fournir des métriques centrées sur les bugs. À travers des expériences rigoureuses, mises en place sur 200,000 heures-CPU et impliquant des fuzzers à base de mutations de pointe, MAGMA met en évidence les limites de l'utilisation des comptages de crashs comme métrique d'évaluation par défaut, et fournit une plateforme unifiée pour une évaluation et une comparaison précises des fuzzers.

Le deuxième projet, IGOR, aborde l'inefficacité des techniques de déduplication des crashs, qui souffrent généralement d'inflation et de conflation des comptes de bugs. Igor s'appuie sur l'aperçu qu'un bug ne peut pas être déclenché sans exécuter son code. À travers un processus de minimisation de cas de test et d'appariement de traces d'exécution, nous introduisons une métrique pour la déduplication de crash qui va au-delà de la couverture de code et des piles d'appels. En employant des comparaisons de similarité de graphes de flux de contrôle sur des traces d'exécution minimisées, IGOR démontre sa capacité à regrouper avec précision les crashs, réduisant les comptes de bugs «uniques» d'un ordre de grandeur par rapport aux techniques existantes.

L'avant-dernier projet, TANGO, aborde l'insuffisance des métriques traditionnelles de couverture de code pour explorer les espaces d'états de systèmes complexes comme les analyseurs de langage et les jeux vidéo. En incorporant «l'état» en tant que citoyen de première classe, TANGO améliore la capacité du fuzzer à naviguer dans des systèmes complexes. L'inférence d'état a conduit à la découverte de bugs auparavant non détectés, et elle a également mis en évidence une observation nouvelle : la couverture de code est insuffisante pour décrire l'état. À travers notre évaluation, TANGO révèle que les fuzzers qui s'appuient uniquement sur la couverture de code pourraient potentiellement passer plus de 80% de leur temps à dupliquer leurs efforts face à des cibles étatiques.

Enfin, SENSEI vise à aider les fuzzers à explorer de manière incrémentielle les cibles en fuzzant les analyseurs isolément et en utilisant ces résultats pour amorcer le fuzzing de cibles plus complexes. En tirant parti de la riche et spécifique connaissance du domaine encodée dans les dissecteurs

Wireshark, SENSEI intègre des métriques spécifiques aux analyseurs pour guider le fuzzing dans la direction d'entrées de haute qualité et diversifiées, pour aider à l'exploration des implémentations de protocoles réseau.

Collectivement, ces projets introduisent de nouvelles métriques et méthodologies pour le développement et l'évaluation des fuzzers. Ils fournissent des preuves empiriques soutenant la thèse que des métriques sur mesure sont la clé d'un fuzzing efficace et réussi.

Contents

Acknowledgments **v**

Abstract (English/Français) **vii**

1 Introduction **1**

1.1 Fuzzing Roadblocks . . . 2
1.2 The Fuzzing Frontier . . . 3
1.2.1 Performance Evaluation . . . 3
1.2.2 Bug Identification . . . 4
1.2.3 Stateful Exploration . . . 6
1.3 Going Beyond Coverage . . . 6
1.3.1 Improving Performance Evaluation through Ground-truth Benchmarks . . . 6
1.3.2 Simplifying Bug Identification through Crash De-duplication . . . 7
1.3.3 Guiding Stateful Exploration with State-aware Metrics . . . 8
1.3.4 Going Further . . . 8

2 MAGMA: A Ground-Truth Fuzzing Benchmark **11**

2.1 Background . . . 13
2.1.1 Fuzzing . . . 13
2.1.2 The Current State of Fuzzer Evaluation . . . 14
2.2 Desired Benchmark Properties . . . 17
2.2.1 Diversity (P1) . . . 17
2.2.2 Verifiability (P2) . . . 18
2.2.3 Usability (P3) . . . 19
2.3 MAGMA: Approach . . . 20
2.3.1 Target Selection . . . 21
2.3.2 Bug Selection and Insertion . . . 22
2.3.3 Performance Metrics . . . 24
2.3.4 Runtime Monitoring . . . 25
2.4 Design and Implementation Decisions . . . 25
2.4.1 Forward-Porting . . . 25
2.4.2 Weird States . . . 26
2.4.3 A Static Benchmark . . . 27
2.4.4 Leaky Oracles . . . 28
2.4.5 Proofs of Vulnerability . . . 29
2.4.6 Unknown Bugs . . . 30
2.4.7 Fuzzer Compatibility . . . 30

2.5 Evaluation . . . 30
  2.5.1 Methodology . . . 30
  2.5.2 Time to Bug . . . 31
  2.5.3 Experimental Results . . . 31
  2.5.4 Discussion . . . 37
2.6 MAGMA Summary . . . 38

3 IGOR: Crash De-duplication Through Root-Cause Clustering 41
3.1 Background . . . 44
  3.1.1 Behavioral Metrics . . . 44
  3.1.2 Test Case Reduction . . . 47
  3.1.3 Our Approach: IGOR . . . 48
3.2 IGOR Design . . . 49
  3.2.1 Data Preprocessing . . . 49
  3.2.2 IGORFUZZ: Minimum-Coverage Fuzzing . . . 50
  3.2.3 Test Case Similarity Measurement . . . 54
  3.2.4 Bug Clustering . . . 56
3.3 Implementation . . . 59
3.4 Experimental Evaluation . . . 59
  3.4.1 Benchmarks . . . 60
  3.4.2 Summary of Results . . . 61
  3.4.3 Test Case Reduction . . . 62
  3.4.4 Graph Similarity After Minimization . . . 63
  3.4.5 Verifying Minimization Results . . . 65
3.5 Case Studies . . . 66
  3.5.1 Assessing CVE Accuracy . . . 67
  3.5.2 High-value Needles in Test Case Haystacks . . . 67
  3.5.3 Detecting Semantically Equivalent Bugs . . . 68
3.6 Limitations . . . 68
3.7 Related Work . . . 69
  3.7.1 Test Case Reduction . . . 69
  3.7.2 Crash Grouping and De-duplication . . . 69
  3.7.3 Fault Localization . . . 70
3.8 IGOR Summary . . . 70

4 TANGO: Extracting Higher-Order Feedback through State Inference 73
4.1 Background . . . 76
  4.1.1 Snapshot Management . . . 78
  4.1.2 State Modeling . . . 79
  4.1.3 State-aware Operation . . . 81
4.2 State Inference . . . 81

4.2.1 Cross-Pollination . . . 82
4.2.2 Subsumption and Elimination . . . 85
4.2.3 Colorful Collapse . . . 86
4.2.4 Successive Rounds of Inference . . . 88
4.3 Overhead, Optimizations, and Trade-offs . . . 89
4.3.1 State Broadcast (O_B) . . . 90
4.3.2 Response Fingerprinting (O_C) . . . 90
4.3.3 Test Extrapolation (O_D) . . . 91
4.4 TANGO: The Framework . . . 92
4.4.1 Workflow . . . 92
4.4.2 Implementation . . . 93
4.5 Evaluation . . . 93
4.5.1 Experimental Setup . . . 94
4.5.2 Empirical Overhead . . . 94
4.5.3 Snapshots in Biased Queues . . . 96
4.5.4 Reduction Ratio α . . . 98
4.5.5 Case Study on Cross-Inference . . . 99
4.6 Scope . . . 100
4.7 Related Work . . . 101
4.8 TANGO Summary . . . 101

5 SENSEI: Input Structure-Aware Fuzzing through Selective Feedback 103
5.1 Background . . . 105
5.2 Dissector-guided Fuzzing . . . 106
5.3 Preliminary Evaluation and Limitations . . . 108
5.4 Future Work . . . 108
5.4.1 Unified Scheduling . . . 109
5.4.2 Type-Aware Mutations . . . 109
5.5 Related Work . . . 110
5.6 Upcoming Challenges . . . 111

6 Summary 113

7 Closing Thoughts 115

Appendices 117

Bibliography 141

Curriculum Vitae 155

Chapter 1

Introduction

> Testing can only prove the presence of bugs, not their absence.
>
> *Edsger W. Dijkstra*

Software is written—for the most part—by humans, who are notorious for making mistakes. Mistakes in software give rise to bugs: alterations in the source code that may manifest as unexpected behavior, ranging from benign rounding errors all the way to complete system compromise. Software testing is the practice of assessing, analyzing, and probing a piece of software in search of bugs.

Fuzz testing—hereinafter referred to as fuzzing—is a search-based software testing technique, which has stood out in its bug-finding ability, owing to its speed, soundness, scalability, and accessibility. What sets fuzzing apart from other software testing techniques is that fuzz tests are concretely executed. Since any detected fault is proven through the input that triggered it, the findings of fuzzers are *sound* by construction, as long as they are reproducible, i.e., the system and its environment are deterministic. Concrete execution also implies speed, seeing as the target under test is evaluated at the *speed* it can process its inputs, while mostly running on bare-metal hardware. Fuzzing is yet another embarrassingly parallel problem, whereby multiple fuzzer instances can be dispatched to *scale up* the testing throughput. And over the past decade, fuzzing research has produced countless artifacts, of which AFL++ is a noteworthy mention due to its active development and its straightforward interface, providing *ease of access* to fuzzing and bringing it one step closer to large-scale adoption.

Grey-box fuzzing, a variant which incorporates runtime feedback for guiding exploration, is the most prominent form of fuzzing. The availability of feedback, often with little impact on performance, enables fuzzers to efficiently explore intricate systems, the most famous example being its ability to craft valid JPEGs "out of thin air" [172]. While several types of feedback have been explored, code coverage has proven to be the most effective and remains the metric of choice in most state-of-the-art fuzzers. The motivating intuition for code coverage is that "a bug cannot be triggered without executing the buggy code". Maximizing code coverage thus serves as a proxy objective for finding bugs: the more code is covered, the more likely that buggy code is executed, and that bugs are triggered. Further, code coverage became, and continues to be, the primary performance metric for comparing fuzzers. However, within this reasoning lies one flawed assumption that hinders the ability of fuzzers to explore complex systems, namely that bugs exist only in code.

1.1 Fuzzing Roadblocks

To prepare a system for fuzzing, it must consume input and yield observable output. A fuzzer would then sample the input space and execute the system with the chosen inputs, incorporating the output as feedback for its progress. The first and foremost challenge then resides in how to sample the–practically infinite–input space in a finite amount of time while maximizing the fuzzer's performance. To reduce the time spent executing a program with invalid inputs, a fuzzer that targets such software would have to craft inputs that diversify the explored behavior of the target, with "invalid input" being only one such behavior. This entails generating inputs with some knowledge of what the target expects as input.

In the context of software systems, namely those that accept byte sequences as inputs, the primary roadblocks to overcome take the form of (a) magic bytes, (b) structured formats, and (c) data transformations, among others.

For instance, a PNG image parser expects the first 8 bytes to be a fixed sequence. A fuzzer that is uniformly sampling bytes has a 1 in 2^{64} chance of selecting this sequence at random. Considering that the placement of the sequence within the input is also subject to randomness, the chances of it landing at the start are almost infinitely slim. To tackle this problem, RedQueen [12] collects feedback on the value comparisons made during execution, and through a process of matching and elimination, finds locations in the input where one operand of the comparison can be patched to match the other. This approach also allows RedQueen to patch in dynamic values such as calculated checksums. Alternatively, LAF [71] relies on compile-time program transformations to split multi-byte comparisons down to their constituents. This enables the fuzzer to incrementally solve the magic byte comparison by sampling from a space of 256 bytes at a time. Leveraging code coverage, the fuzzer can then observe when a single-byte comparison succeeds, and builds upon it to find the next bytes.

The PNG format specifies the structure of the image file and the chunks within it. For example, it defines the layout of each chunk and the order in which chunks must appear in a valid image. Stemming from the same issue of an infinite input space, the challenge of randomly generating valid chunks or sequences of them presents a harder obstacle to fuzzers. FormatFuzzer [43] incorporates machine-readable specification, in the form of binary dissectors, into the input generation process. By transforming the dissectors into generators, it enables the fuzzer to generate mostly-valid inputs without manually writing the specification. Other approaches have also been explored to tackle different formats. Mimid [60] infers context-free grammars implemented in parsers by tracking accesses to input bytes and reconstructing the parse trees. Google also explored experimental venues on LLM-aided fuzzing [55], whereby artificial neural networks, trained on vast amounts of code and text mined from the Internet, are used to create input generators and invocation wrappers for a given code base. Despite not being completely principled or understood, LLM prompting techniques have proven effective and may continue to find their way into fuzzing and other domains in systems security.

A PNG image also encodes pixels as a DEFLATE-compressed byte stream, which would then require the fuzzer to generate valid byte blocks that can be correctly decompressed. Without encoding that knowledge in the fuzzer, it would be next to impossible for it to pass this hurdle. Although practical solutions resort to disabling compression in the PNG parser, such solutions are not comprehensive, and may render some functionality in the parser unreachable, and hence, not testable. Fuzztruction [17] circumvents this requirement by relying on other programs that already encode this domain knowledge: generators. By fuzzing a PNG generator, such as a photo editor, it introduces an intermediate layer that transforms entropy into mostly valid inputs: instead of sampling the input bytes directly, it samples mutations to be applied to the generator program itself and executes it to yield mutated inputs.

1.2 The Fuzzing Frontier

Leveraging the knowledge that has been accrued through the latest research, state-of-the-art fuzzers can pass the first set of hurdles with relative ease. However, while there is no shortage of bugs found by fuzzers, they remain unable to effectively test complex systems, as they face a new set of challenges: (a) performance evaluation; (b) bug identification; (c) and stateful exploration.

1.2.1 Performance Evaluation

As a software testing technique, fuzzing's main objective is to discover bugs. An ideal objective for fuzzers would then be to maximize the number of triggered bugs, allowing for "partial triggers" as the fuzzer approaches a bug. However, formulating the objective in this manner would require knowledge of the bugs present in the system under test, defeating the purpose of testing. To that end, code coverage is the most prevalent metric of choice for measuring the fuzzer's progress and incorporating it as feedback.

Research is an incremental process, building upon what succeeds and learning from what fails. Evaluating the performance of a fuzzer serves as a measure of the fuzzer's success and determines its role and its impact. As such, to foster healthy research in fuzzing, it is imperative that assessments are performed in a sound and accurate manner. Code coverage has been adopted as the metric of choice for evaluating and comparing the performance of fuzzers. While fuzzers are free to choose which proxy metric they optimize for exploring the target and finding bugs, relying on one such metric to rank fuzzers does not yield sound results. A fuzzer that maximizes code coverage is naturally more likely to outrank another that optimizes for a different metric, such as data-flow coverage. Instead, we should use an objective metric that accurately measures fuzzer performance in alignment with their initial goal: finding bugs. The paradoxical nature of this metric can be avoided by assuming ground-truth knowledge of the bugs, in the context of a fuzzing benchmark. By introducing known bugs into a target, and means to measure when they are triggered, we can develop a benchmark where fuzzers are scored based on the bugs they find and how fast they found them. In Chapter 2, we introduce the challenges in designing a fair benchmark and provide a suite of real targets with real bugs and performance metrics based in ground truth.

1.2.2 Bug Identification

In short, a bug is a deviation from expected behavior. Although comprehensive, this definition is too loose to form a basis for a bug detection tool. The passive voice in "expected behavior" implies that there is an implicit counterpart: who is the expecting agent? For the most part, that refers to the developer who maintains the system in question. It is also possible that expectations are set by the model of the system (e.g., finite state machines or context-free grammars) or the framework within which the system is built, such as the instruction set architecture (ISA), the operating system (OS), or the semantics of the programming language.

Implicitly-validated behaviors

Example 1.1

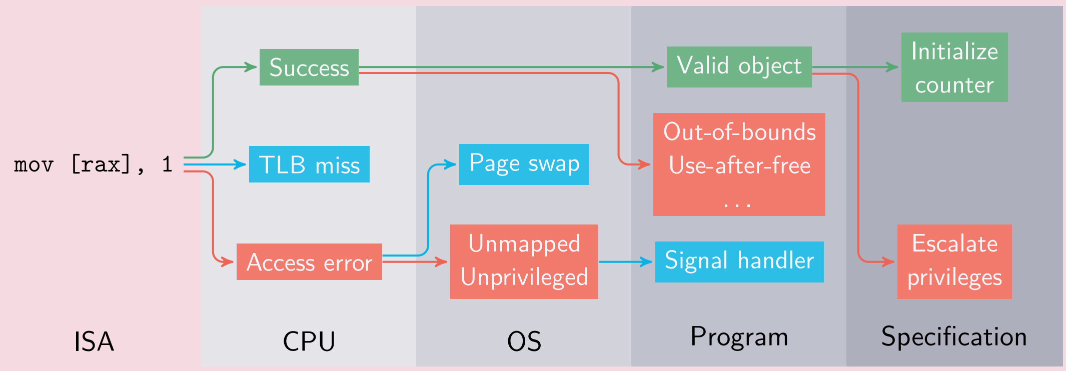


Figure 1.1 – Different views of a bug across the execution stack. The validity of a behavior depends on the expectations set by the observer.

The output of a fuzzer is typically a proof-of-concept file (PoC) which encodes the input and conditions needed to reproduce a bug detected in testing. It is then crucial that fuzzers are able to detect when a fault has occurred, i.e., when a bug has been triggered; otherwise, its output is void and its progress is lost. However, the lack of a clear definition for a bug makes it difficult to detect it reliably.

Further reading: Sanitizers

Earlier fuzzers, which mainly targeted user-space processes in Linux and were written in memory-unsafe languages like C/C++, resorted to *crashes* as indicators of bugs. Such crashes typically occur due to unmapped or unprivileged memory accesses, or due to the execution of invalid instructions, all indicating that something has gone awry. Although the C/C++ standards define the notion of objects and their liveness (e.g., function-local vs global variables), they do not enforce checks for their memory safety policies. As such, not all memory safety bugs can be detected through crashes, and many bugs could be lost in fuzzing.

To remedy that, later advancements in compilers introduced *sanitizers*: runtime mechanisms for enforcing language semantics, backed by compile-time instrumentation. AddressSanitizer (ASan) is one such mechanism, where object bounds and lifetimes are tracked, and memory accesses are augmented with validity checks, to reduce the likelihood that a violation goes undetected. Yet, these sanitizers can only enforce language-specific semantics. A bug in the logic of the program cannot be detected through such means.

Sanitizers widened the scope of bugs that could be detected by fuzzers for a relatively low cost, but they had an unintended side-effect: overwhelming developers with bug reports. Despite those sanitizers providing context-rich summaries to help developers in triaging crashes, it remains a fact that they enforce language-level policies rather than the programmer's expectations (or the reference specification). A single bug in the implementation can manifest as many, seemingly-different violations detected by the sanitizer. As a result, developers are left to triage hundreds or thousands of bug reports, many of which are duplicates resulting from the same root cause. *De-duplicating* crashes is the process of grouping PoCs that result from the same bug. Given that the bug is not known *a priori*, proxy metrics are again used to approximate the root cause. A study by Klees et al. [83] reports that de-duplicated bug counts can still inflate true bug counts by anywhere between 66× to 6339× when using stack hashes or coverage profiles, respectively. Fuzzers' inability to discern true bugs represents a challenge to their usability and impacts their effectiveness in securing real and complex systems. We tackle this issue in Chapter 3, where we propose and evaluate a technique for grouping and de-duplicating crashes based on execution trace clustering. The key insight is that if two PoCs share the same root cause, they must execute the same buggy code.

1.2.3 Stateful Exploration

Fuzzing stateful systems requires special consideration: observed behavior is only reproducible in the context of the current state. State adds a new dimension to the operations of a fuzzer. Further, any interaction with the system may alter its state. This introduces a new challenge to fuzzers: state explosion. To ensure reproducibility of results, a fuzzer must keep track of the current state and the sequence of interactions made within it. This entails recognizing when a state transition took place, as well as maintaining the set of explored states and dedicating sufficient resources for exploring them.

There is often no explicit definition of a state in the implementation of a protocol or a specification. State is rather the sum and summary of the system's runtime behavior and how it processes inputs. Despite not having direct access to state-aware feedback, it is possible for a fuzzer to observe the behavior of a system and make conclusions based on how it handles inputs under different conditions. State inference is the subject of Chapter 4, where we develop a state-aware metric through active invocation and learning of the system's behavior.

1.3 Going Beyond Coverage

Thesis Statement

Fuzzing is a search for bugs guided by feedback metrics. For simple targets, executing buggy code is often sufficient to trigger faulty behavior, and code coverage excels at guiding the fuzzer towards such bugs. However, to assess and improve the efficacy of fuzzers on more complex systems, specialized metrics that capture application-specific information, such as system state and bug semantics, are instrumental for targeting bugs embedded deeply in the code.

In this dissertation, we address the problems encountered when using code coverage as the objective metric in fuzzing. By tackling the challenges arising from incorporating other metrics, we present novel techniques for improving fuzzers through accurate performance evaluation, path-based crash de-duplication, and state-aware exploration.

1.3.1 Improving Performance Evaluation through Ground-truth Benchmarks

High scalability and low running costs have made fuzz testing the de facto standard for discovering software bugs. Fuzzing techniques are constantly being improved in a race to build the ultimate bug-finding tool. However, while fuzzing excels at finding bugs in the wild, evaluating and comparing fuzzer performance is challenging due to the lack of metrics and benchmarks. For example, crash count—perhaps the most commonly-used performance metric—is inaccurate due to imperfections in de-duplication techniques. Additionally, the lack of a unified set of targets results in ad hoc evaluations that hinder fair comparison.

In Chapter 2, we tackle these problems by developing MAGMA, a ground-truth fuzzing benchmark that enables uniform fuzzer evaluation and comparison. By introducing *real* bugs into *real* software, MAGMA allows for the realistic evaluation of fuzzers against a broad set of targets. By instrumenting these bugs, MAGMA also enables the collection of bug-centric performance metrics independent of the fuzzer. MAGMA is an open benchmark consisting of seven targets that perform a variety of input manipulations and complex computations, presenting a challenge to state-of-the-art fuzzers.

We evaluate seven widely-used mutation-based fuzzers (AFL, AFLFast, AFL++, FAIRFUZZ, MOPT-AFL, honggfuzz, and SYMCC-AFL) against MAGMA over 200,000 CPU-hours. Based on the number of bugs, reached, triggered, and detected, we draw conclusions about the fuzzers' exploration and detection capabilities. This provides insight into fuzzer performance evaluation, highlighting the importance of ground truth in performing more accurate and meaningful evaluations.

1.3.2 Simplifying Bug Identification through Crash De-duplication

The output of a fuzzer is a set of proof-of-concept (PoC) test cases for all observed "unique" crashes. It costs developers substantial efforts to analyze each crashing test case. This mostly manual process has led to the number of reported crashes out-pacing the number of bug fixes. Automatic crash de-duplication techniques, which mostly rely on coverage profiles and stack hashes, are supposed to alleviate these pressures. However, these techniques both inflate actual bug counts and falsely conflate unrelated bugs [83]. This hinders, rather than helps, developers, and calls for more accurate techniques.

The highly-stochastic nature of fuzzing means that PoCs commonly exercise many program behaviors that are orthogonal to the crash's underlying root cause. This diversity in program behaviors manifests as a diversity in crashes, contributing to bug-count inflation and conflation. Based on this insight, Chapter 3 details our proposal for IGOR, an automated dual-phase crash de-duplication technique. By minimizing each PoC's execution trace, we obtain pruned test cases that exercise the critical behavior necessary for triggering a bug. Then, we use a graph similarity comparison to cluster crashes based on the control-flow graph of the minimized execution traces, with each cluster mapping back to a single, unique root cause.

We evaluate IGOR against 39 bugs resulting from 254,000 PoCs, distributed over 10 programs. Our results show that IGOR accurately groups these crashes into 48 uniquely identifiable clusters, while other state-of-the-art methods yield bug counts at least one order of magnitude larger.

1.3.3 Guiding Stateful Exploration with State-aware Metrics

While coverage-guided fuzzing excels at code discovery, its effectiveness falters when applied to complex systems. One such class of complex systems is persistent targets whose behavior depends on the state of the system, where discovering new states is key to comprehensive testing. Code coverage alone is not enough to describe the state of a system. This makes it difficult for an evolutionary fuzzer to optimize for state discovery when the feedback is not correlated with the objective.

Chapter 4 introduces TANGO, an extensible framework for state-aware fuzzing. Our design incorporates "state" as a first-class citizen in all operations, enabling TANGO to fuzz complex targets that otherwise remain out-of-scope. Inspired by hypothesis testing and system identification, we present *state inference*, a cross-validation technique that distills portable coverage metrics to reveal hidden path dependencies in the target. State awareness in turn allows us to aggregate feedback from different paths while maintaining state-specific operation. We leverage TANGO to fuzz stateful targets ranging from language parsers to video games, demonstrating the flexibility of our framework in exploring complex systems. Using state inference, we employ an active learning process to shrink the scheduling queue of a fuzzer by over 80% through identifying functionally-equivalent paths. We extend current state-of-the-art fuzzers, AFL++ and Nyx-Net, with state feedback from TANGO. During our evaluation, fuzzers using our technique uncovered two new bugs in `yajl` and `dcmtk`.

1.3.4 Going Further

The bulk of this dissertation tackles the aforementioned three challenges at the fuzzing frontier. Finally, in Chapter 5, we outline a proposal for future work, where we present the challenges still encountered in the efficient exploration of network protocol servers, mainly addressing the obstacles impeding the generation of high-quality inputs. We propose a design for a fuzzing pipeline that incorporates domain knowledge encoded in Wireshark dissectors, by leveraging them as oracles for the quality of generated inputs. To incorporate feedback from both the dissector and the target, we apply multi-objective optimization, which allows the fuzzer to find inputs that simultaneously maximize target coverage while improving the quality and validity of the generated inputs. While this portion of the dissertation remains as future work, it underscores our conclusion, from previous chapters, that metrics are pivotal for the efficacy of fuzzers against complex systems.

This dissertation thus tackles the shortcomings of existing metrics in fuzzing, through the introduction of application-specific metrics tailored for the fuzzer's objective. We summarize our contributions as follows:

- With MAGMA [64], we explore ways to interpret and assess the output of fuzzers by introducing accurate ground-truth metrics to measure their performance as a function of the discovered bugs.
- Through IGOR [75], we present an approach for extending access to bug-specific metrics beyond the context of a benchmark and into the realm of software in-the-wild.
- We also develop TANGO[1] to tackle the challenges put forth by stateful targets through a state-aware fuzzing framework that accounts for the target's state as another dimension to the fuzzer's operations.
- Finally, to better guide a fuzzer's incremental problem-solving in the face of complex targets, we propose SENSEI[2], a dual-phase fuzzing architecture that leverages domain knowledge encoded in open-source parsers to generate high-quality inputs and guide the exploration of full-fledged network systems.

---

[1] TANGO is currently under submission.

[2] We present SENSEI as future work, outlining its design and preliminary evaluation.

Chapter 2

MAGMA: A Ground-Truth Fuzzing Benchmark

> Not everything that can be counted counts, and not everything that counts can be counted.
>
> *William Bruce Cameron*

Fuzzing has become a cornerstone technique for automated bug discovery in software systems. A crucial challenge lies in evaluating and comparing the performance of different fuzzers: fair metrics and representative workloads. This chapter aims to highlight the limitations of prevalent metrics like crash counts and code coverage, emphasizing the need for more robust evaluation methods. It also introduces the challenges of crafting more accurate and fair benchmarks for fuzzing, and how we tackle them in the development of our own benchmark, MAGMA.

Hypothesis 1

The goal of fuzzers is to find bugs. To properly evaluate fuzzers, the use of classical metrics like code coverage and crash counts is insufficient, and metrics that accurately depict bug-finding ability are necessary.

The contents of this chapter are adapted from Hazimeh et al. [64].

Fuzzing is a widely-used dynamic bug discovery technique. A fuzzer procedurally generates inputs and subjects the target program (the "target") to these inputs with the aim of triggering a fault (i.e., discovering a bug). Fuzzing is an inherently sound but incomplete bug-finding process (given finite resources). State-of-the-art fuzzers rely on *crashes* to mark faulty program behavior. The existence of a crash is generally symptomatic of a bug (soundness), but the lack of a crash does not necessarily mean that the program is bug-free (incompleteness). Fuzzing is wildly successful in finding bugs in open-source [4] and commercial off-the-shelf [9, 10, 132] software.

The success of fuzzing has resulted in an explosion of new techniques claiming to improve bug-finding performance [104]. In order to highlight improvements, these techniques are typically evaluated across a range of metrics, including: (i) crash counts; (ii) ground-truth bug counts; and/or (iii) code-coverage profiles. While these metrics provide some insight into a fuzzer's performance, we argue that they are insufficient for use in fuzzer comparisons. Furthermore, the set of targets that these metrics are evaluated on can vary wildly across papers, making cross-fuzzer comparisons impossible. Each of these metrics has particular deficiencies.

Crash counts The simplest fuzzer evaluation method is to count the number of crashes triggered by a fuzzer, and compare this crash count with that achieved by another fuzzer (on the same target). Unfortunately, crash counts often inflate the number of actual bugs in the target [83]. Moreover, deduplication techniques (e.g., coverage profiles, stack hashes) fail to accurately identify the root cause of these crashes [19, 83].

Bug counts Identifying a crash's *root cause* is preferable to simply reporting raw crashes, as it avoids the inflation problem inherent in crash counts. Unfortunately, obtaining an accurate *ground-truth* bug count typically requires extensive manual triage, which in turn requires someone with extensive domain expertise and experience [114].

Code-coverage profiles Code-coverage profiles are another performance metric commonly used to evaluate and compare fuzzing techniques. Intuitively, covering more code correlates with finding more bugs. However, previous work [83] has shown that there is a weak correlation between coverage-deduplicated crashes and ground-truth bugs, implying that higher coverage does not necessarily indicate better fuzzer effectiveness.

The deficiencies of existing performance metrics calls for a rethinking of fuzzer evaluation practices. In particular, the performance metrics used in these evaluations must accurately measure a fuzzer's ability to achieve its main objective: *finding bugs*. Similarly, the targets that are used to assess how well a fuzzer meets this objective must be realistic and exercise diverse behavior. This allows a practitioner to have confidence that a given fuzzing technique will yield improvements when deployed in real-world environments.

To satisfy these criteria, we present MAGMA, a ground-truth fuzzer benchmark based on real programs with real bugs. MAGMA consists of seven widely-used open-source libraries and applications, totalling 2 MLOC. For each MAGMA workload, we manually analyze security-relevant bug reports and patches, reinserting defective code back into these seven programs (in total, 118 bugs were analyzed and reinserted). Additionally, each reinserted bug is accompanied by a lightweight *oracle* that detects and reports if the bug is *reached* or *triggered*. This distinction between reaching and triggering a bug—in addition to a fuzzer's ability to *detect* a triggered bug—presents a new opportunity to evaluate a fuzzer across multiple dimensions (again, focusing on ground-truth bugs).

The remainder of this chapter presents the motivation behind MAGMA, the methodology behind MAGMA's design and choice of performance metrics, implementation details, and a set of preliminary results that demonstrate MAGMA's utility. We make the following contributions:

- A set of bug-centric performance metrics for a fuzzer benchmark that allow for a fair and accurate evaluation and comparison of fuzzers.
- A quantitative comparison of existing fuzzer benchmarks.
- The design and implementation of MAGMA, a ground-truth fuzzing benchmark based on real programs with real bugs.
- An evaluation of MAGMA against seven widely-used fuzzers.

2.1 Background

This section introduces fuzzing as a software testing technique, and how new fuzzing techniques are currently evaluated and compared against existing ones. This aims to motivate the need for new fuzzer evaluation practices.

2.1.1 Fuzzing

A fuzzer is a dynamic testing tool that discovers software flaws by running a target program (the "target") with a large number of automatically-generated inputs. Importantly, these inputs are generated with the intention of triggering a crash in the target. This input generation process is dependent on the fuzzer's knowledge of the target's *input format* and *program structure*. For example, *grammar-based* fuzzers (e.g., Superion [160], Peach [44], and QuickFuzz [61]) leverage the target's input format (which must be specified *a priori*) to intelligently craft inputs (e.g., based on data width and type, and on the relationships between different input fields). In contrast, *mutational* fuzzers (e.g., AFL [171], Angora [29], and MemFuzz [33]) require no *a priori* knowledge of the input format. Instead, mutational fuzzers leverage preprogrammed mutation operations to iteratively modify the input.

Fuzzers are classified by their knowledge of the target's program structure. For example, *whitebox* fuzzers [52, 54, 125] leverage program analysis to infer knowledge about the program structure. In comparison, *blackbox* fuzzers [8, 165] blindly generate inputs in the hope of discovering a crash. Finally, *greybox* fuzzers [29, 98, 171] leverage program instrumentation (instead of program analysis) to collect runtime information. Program-structure knowledge guides input generation in a manner more likely to trigger a crash.

Importantly, fuzzing is a *highly stochastic* bug-finding process. This randomness is independent of whether the fuzzer synthesizes inputs from a grammar (grammar-based fuzzing), transforms an existing set of inputs to arrive at new inputs (mutational fuzzing), has no knowledge of that target's internals (blackbox fuzzing), or uses sophisticated program analyses to understand the target (whitebox fuzzing). The stochastic nature of fuzzing makes evaluating and comparing fuzzers difficult. This problem is exacerbated by existing fuzzer evaluation metrics and benchmarks.

2.1.2 The Current State of Fuzzer Evaluation

The rapid emergence of new and improved fuzzing techniques [104] means that fuzzers are constantly compared against one another, in order to empirically demonstrate that the latest fuzzer supersedes previous state-of-the-art fuzzers. To enable fair and accurate fuzzer evaluation, it is critical that fuzzing campaigns are conducted on a suitable benchmark that uses an appropriate set of metrics. Unfortunately, fuzzer evaluations have so far been ad hoc and haphazard. For example, Klees et al.'s study of 32 fuzzing papers found that *none* of the surveyed papers provided sufficient detail to support their claims of fuzzer improvement [83]. Notably, their study highlights a set of criteria that should be adopted across all fuzzer evaluations. These criteria include:

Performance metrics: How the fuzzers are evaluated and compared. This is typically one of the approaches previously discussed (crash count, bug count, or coverage profiling).

Targets: The software being fuzzed. This software should be both diverse and realistic so that a practitioner has confidence that the fuzzer will perform similarly in real-world environments.

Seed selection: The initial set of inputs that bootstrap the fuzzing process. This initial set of inputs should be consistent across repeated trials and the fuzzers under evaluation.

Trial duration (timeout): The length of a single fuzzing trial should also be consistent across repeated trials and the fuzzers under evaluation. We use the term *trial* to refer to an instance of the fuzzing process on a target program, while a *fuzzing campaign* is a set of N repeated trials on the same target.

Number of trials: The highly-stochastic nature of fuzzing necessitates a large number of repeated trials, allowing for a statistically sound comparison of results.

Table 2.1 – Summary of existing fuzzer benchmarks and our benchmark, Magma. We characterize benchmarks across two dimensions: the targets that make up the benchmark workloads and the bugs that exist across these workloads. For both dimensions we count the number of workloads/bugs (#) and classify them as **R**eal or **S**ynthetic. Bug density is the mean number of bugs per workload. Finally, ground truth may be available (✓), available but not easily accessible (◗), or unavailable (✗).

Benchmark	Workloads		Bugs		Bug Density	Ground truth
	#	Real/Synthetic	#	Real/Synthetic		
BugBench [99]	17	R	19	R	1.12	◗
CGC [26]	131	S	590	S	4.50	◗
Google FTS [58]	24	R	47	R	1.96	◗
Google FuzzBench [57]	21	R	—	—	—	—
LAVA-M [40]	4	R	2265	S	566.25	✓
UniFuzz [93]	20	R	?	R	?	✗
Open-source software	—	R	?	R	?	✗
Magma	7	R	118	R	16.86	✓

Klees et al.'s study demonstrates the need for a *ground-truth fuzzing benchmark*. Such a benchmark must use suitable performance metrics and present a unified set of targets.

Existing Fuzzer Benchmarks

Fuzzers are typically evaluated on a set of targets sourced from one of the following benchmarks. These benchmarks are summarized in table 2.1.

The LAVA-M [40] test suite (built on top of `coreutils-8.24`) aims to evaluate the effectiveness of a fuzzer's exploration capability by injecting bugs in different execution paths. However, the LAVA bug injection technique only injects a single, simple bug type: an out-of-bounds memory access triggered by a "magic value" comparison. This bug type does not accurately represent the statefulness and complexity of bugs encountered in real-world software. We quantify these observations in section 2.5.3.

In contrast, the Cyber Grand Challenge (CGC) [26] sample set provides a wider variety of bugs that are suitable for testing a fuzzer's fault detection capabilities. Unfortunately, the relatively small size and simplicity of the CGC's synthetic workloads does not enable thorough evaluation of the fuzzer's ability to explore complex programs.

BugBench [99] and the Google Fuzzer Test Suite (FTS) [58] both contain real programs with real bugs. However, each target only contains one or two bugs (on average). This sparsity of bugs, combined with the lack of automatic methods for triaging crashes, hinders adoption and makes both benchmarks unsuitable for fuzzer evaluation. In contrast, Google FuzzBench [57]—the successor to the Google FTS—is a fuzzer evaluation platform that relies solely on coverage profiles as a performance metric. As previously discussed, this metric has limited utility when evaluating fuzzers on their bug-finding capability. UniFuzz [93]—which was developed concurrently but independently from Magma—is similarly built on real programs containing real bugs. However, it lacks ground-truth knowledge and it is unclear how many bugs each target contains. Not knowing how many bugs exist in a benchmark makes fuzzer comparisons challenging.

Finally, popular open-source software (OSS) is often used to evaluate fuzzers [22, 83, 90, 102, 121, 159]. Although real-world software is used, the lack of ground-truth knowledge about the triggered crashes makes it difficult to provide an accurate, verifiable, quantitative evaluation. First, it is often unclear which software version is used, making fair cross-paper comparisons impossible. Second, multiple software versions introduce *version divergence*, a subtle evaluation flaw shared by both crash and bug count metrics. After running for an extended period, a fuzzer's ability to discover new bugs diminishes over time [20]. If a second fuzzer later fuzzes a new version of the same program—with the bugs found by the first fuzzer appropriately patched—then the first fuzzer will find fewer bugs in this newer version. Version divergence is also inherent in UniFuzz, which builds on top of older versions of OSS.

Crashes as a Performance Metric

Most, if not all, state-of-the-art fuzzers implement fault detection as a *crash listener*. A program crash can be caused by an *architectural violation* (e.g., division-by-zero, unmapped/unprivileged page access) or by a *sanitizer* (a dynamic bug-finding tool that generates a crash when a security policy violation—e.g., object out-of-bounds, type safety violation—occurs [148]).

The simplicity of crash detection has led to the widespread use of *crash count* as a performance metric for comparing fuzzers. However, crash counts have been shown to yield inflated results, even when combined with deduplication methods (e.g., coverage profiles and stack hashes) [19, 83]. Instead, the number of bugs found by each fuzzer should be compared: if fuzzer A finds more bugs than fuzzer B, then A is superior to B. Unfortunately, there is no single formal definition for a bug. Defining a bug in its proper context is best achieved by formally modeling program behavior. However, deriving formal program models is a difficult and time-consuming task. As such, bug detection techniques tend to create a blacklist of faulty behavior, mislabeling or overlooking some bug classes in the process. This often leads to incomplete detection of bugs and root-cause misidentification, resulting in a duplication of crashes and an inflated set of results.

2.2 Desired Benchmark Properties

Benchmarks are important drivers for computer science research and product development [18]. Several factors must be taken into account when designing a benchmark, including: relevance; reproducibility; fairness; verifiability; and usability [2, 82]. While building benchmarks around these properties is well studied [2, 18, 79, 82, 99, 129, 135, 150], the highly-stochastic nature of fuzzing introduces new challenges for benchmark designers.

For example, *reproducibility* is a key benchmark property that ensures a benchmark produces "*the same results consistently for a particular test environment*" [82]. However, individual fuzzing trials vary wildly in performance, requiring a large number of repeated trials for a particular test environment [83]. While performance variance exists in most benchmarks (e.g., the SPEC CPU benchmark [150] uses the median of three repeated trials to account for small variations across environments), this variance is more pronounced in fuzzing. Furthermore, a fuzzer may actively modify the test environment (e.g., T-Fuzz [121] and FuzzGen [72] transform the target, while Skyfire [159] generates new seed inputs for the target). This is very different to traditional performance benchmarks (e.g., SPEC CPU [150], DaCapo [18]), where the workloads and their inputs remain fixed across all systems-under-test. This leads us to define the following set of properties that we argue *must* exist in a fuzzing benchmark:

Diversity (P1): The benchmark contains a wide variety of bugs and programs that resemble real software testing scenarios.
Verifiability (P2): The benchmark yields verifiable metrics that accurately describe performance.
Usability (P3): The benchmark is accessible and has no significant barriers for adoption.

These three properties are explored in the remainder of this section, while section 2.3 describes how MAGMA satisfies these criteria.

2.2.1 Diversity (P1)

Fuzzers are actively used to find bugs in a variety of *real* programs [4, 9, 10, 132]. Therefore, a fuzzing benchmark must evaluate fuzzers against programs and bugs that resemble those encountered in the "real world". To this end, a benchmark must include a *diverse* set of bugs *and* programs.

Bugs should be diverse with respect to:

Class: Common Weakness Enumeration (CWE) [109] bug classes include memory-based errors, type errors, concurrency issues, and numeric errors.

Distribution: “Depth”, fan-in (i.e, the number of paths which execute the bug), and spread (i.e., the ratio of faulty-path counts to the total number of paths).

Complexity: Number of input bytes involved in triggering a bug, the range of input values which triggers the bug, and the transformations performed on the input.

Similarly, targets (i.e, the benchmark workloads) should be diverse with respect to:

Application domain: File and media processing, network protocols, document parsing, cryptography primitives, and data encoding.

Operations performed: Parsing, checksum calculation, indirection, transformation, state management, and data validation.

Input structure: Binary, text, formats/grammars, and data size.

Satisfying the diversity property requires bugs that resemble those encountered in real-world environments. Both LAVA-M and Google FuzzBench fail this requirement: the former contains only a single bug class (an out-of-bounds memory access), while FuzzBench does not consider bugs as an evaluation metric. BugBench primarily focuses on memory corruption vulnerabilities, but also contains uninitialized read, memory leak, data race, atomicity, and semantic bugs (totalling nine bug classes). Conversely, Google FTS and FuzzBench satisfy the target diversity requirement: both contain workloads from a wide variety of application domains (e.g., cryptography, image parsing, text processing, and compilers).

Ultimately, real programs are the only source of real bugs. Therefore, a benchmark designed to evaluate fuzzers must include *real programs with a variety of real bugs*, thus ensuring diversity and avoiding bias (e.g., towards a specific bug class). Whereas discovering and reporting real bugs is desirable (i.e, when OSS is used), performance metrics based on an unknown set of bugs (with an unknown distribution) make it impossible to compare fuzzers. Instead, fuzzers should be evaluated on workloads containing known bugs for which ground truth is available and *verifiable*.

2.2.2 Verifiability (P2)

Existing ground-truth fuzzing benchmarks lack a straightforward mechanism for determining a crash’s root cause. This makes it difficult to verify a fuzzer’s results. Crash count, a widely-used performance metric, suffers from high variability, double-counting, and inconsistent results across multiple trials (see section 2.1.2). Automated techniques for deduplicating crashes are not reliable, and hence should not be used to verify the bugs discovered by a fuzzer. Ultimately, a fuzzing benchmark should provide a set of known bugs for which ground truth can be used to verify a fuzzer’s findings.

While the CGC sample set provides crashing inputs—also known as a *proof of vulnerability* (PoV)—for all known bugs, it does not provide a mechanism for determining the root cause of a fuzzer-generated crash. Similarly, the Google FTS provides PoVs (for 87 % of bugs) and a script for triaging and deduplicating crashes. This script parses the crash report or looks for a specific line of code at which to terminate program execution. However, this approach is limited and does not allow for the detection of complex bugs (e.g., where simply executing a line of code is not sufficient to trigger the bug).

In contrast to the CGC and Google FTS benchmarks, for which ground truth is available but not easily accessible, LAVA-M clearly reports the bug triggered by a crashing input. However, LAVA-M does not provide a runtime interface for accessing this information. Unless a fuzzer is specialized to collect LAVA-M metrics, it cannot monitor progress in real-time. Thus, a post-processing step is required to collect metrics. Finally, Google FuzzBench relies solely on coverage profiles (rather than fault-based metrics) to evaluate and compare fuzzers. FuzzBench dismisses the need for ground truth, which we believe sacrifices the significance of the results: more coverage does not necessarily imply higher bug-finding effectiveness.

Ground-truth bug knowledge allows for a fuzzer's findings to be verified, enabling accurate performance evaluation and allowing meaningful comparisons between fuzzers. To this end, a fuzzing benchmark must provide *easy access to ground-truth metrics* describing the bugs a fuzzer can reach, trigger, and detect.

2.2.3 Usability (P3)

Fuzzers have evolved from simple blackbox random-input generation to complex control- and data-flow analysis tools. Each fuzzer may introduce its own instrumentation into a target (e.g., AFL [171]), run the target in a specific execution engine (e.g., QSYM [170], Driller [152]), or provide inputs through a specific channel (e.g., libFuzzer [98]). Fuzzers come in a variety of forms (described in section 2.1.1), so a fuzzing benchmark must not exclude a particular type of fuzzer. Additionally, using a benchmark must be manageable and straightforward: it should not require constant user intervention, and benchmarking should finish within a reasonable time frame. The inherent randomness of fuzzing complicates this, as multiple trials are required to achieve statistically-meaningful results.

Some existing benchmark workloads (e.g., those from CGC and Google FTS) contain multiple bugs, so it is not sufficient to only run the fuzzer until the first crash is encountered. However, the lack of easily-accessible ground truth makes it difficult to determine if/when all bugs are triggered. Moreover, inaccurate deduplication techniques mean that the user cannot simply equate the number of crashes with the number of bugs. Thus, additional time must be spent triaging crashes to obtain ground-truth bug counts, further complicating the benchmarking process.

In summary, a benchmark should be *usable* by fuzzer developers, without introducing insurmountable or impractical barriers to adoption. To satisfy this property, a benchmark must thus provide a *small set of targets with a large number of discoverable bugs*, and it must provide a *usable framework that measures and reports fuzzer progress and performance*.

2.3 MAGMA: Approach

We present MAGMA, a ground-truth fuzzing benchmark that satisfies the previously-discussed benchmark properties. MAGMA is a collection of seven targets with widespread use in real-world environments. These initial targets have been carefully selected for their *diversity* and the variety of security-critical bugs that have been reported throughout their lifetimes (satisfying **P1**).

Importantly, MAGMA's seven workloads contain 118 bugs for which ground truth is *easily accessible* and *verifiable* (satisfying **P2**). These bugs are sourced from older versions of the seven workloads, and then *forward-ported* to the latest version contained within MAGMA. Finally, MAGMA imposes minimal requirements on the user, allowing fuzzer developers to seamlessly integrate the benchmark into their development cycle (satisfying **P3**).

For each workload, we manually inspect bug and vulnerability reports to find bugs that are suitable for inclusion in MAGMA (e.g., ensuring that the bug affects the core codebase). For these bugs, we reintroduce ("inject") each bug into the latest version of the code through a process we call *forward-porting* (see section 2.3.2). In addition to the bug, we also insert minimal source-code instrumentation—a *canary*—to collect data about a fuzzer's ability to reach and trigger the bug (see section 2.3.3). A bug is *reached* when the faulty line of code is executed, and *triggered* when the fault condition is satisfied. Finally, MAGMA provides a *runtime monitor* that runs in parallel with the fuzzer to collect real-time statistics. These statistics are used to evaluate the fuzzer (see section 2.3.4).

Fuzzer evaluation is based on the number of bugs *reached*, *triggered*, and *detected*. MAGMA's instrumentation only yields usable information when the fuzzer exercises the instrumented code, allowing us to determine whether a bug is *reached*. The fuzzer-generated input *triggers* a bug when the input's dataflow satisfies the bug's trigger condition(s). Once triggered, the fuzzer should flag the bug as a fault or crash, enabling us to assess the fuzzer's bug *detection* capability. These metrics are described further in section 2.3.3.

Finally, MAGMA provides a *fatal canaries* mode. In fatal canaries mode, the program is terminated if a canary's condition is satisfied (similar to LAVA-M). The fuzzer then saves this crashing input for post-processing. Fatal canaries are a form of *ideal sanitization*, in which triggering a bug immediately results in a crash, regardless of the nature of the bug. Fatal canaries allow developers to evaluate their fuzzers under ideal sanitization assumptions without incurring additional

Table 2.2 – The targets, driver programs, bug counts, and evaluated features incorporated into Magma. The versions used are the latest at the time of writing.

Target	Drivers	Version	File type	Bugs	Magic values	Recursive parsing	Compression	Checksums	Global state
libpng	`read_fuzzer`, `readpng`	1.6.38	PNG	7	✓	✗	✓	✓	✗
libtiff	`read_rgba_fuzzer`, `tiffcp`	4.1.0	TIFF	14	✓	✗	✓	✗	✗
libxml2	`read_memory_fuzzer`, `xml_reader_for_file_f` `xmllint`	2.9.10	XML	18	✓	✓	✗	✗	✗
poppler	`pdf_fuzzer`, `pdfimages`, `pdftoppm`	0.88.0	PDF	22	✓	✓	✓	✓	✗
openssl	`asn1`, `asn1parse`, `bignum`, `bndiv`, `client`, `cms`, `conf`, `crl`, `ct`, `server`, `x509`	3.0.0	*Binary blobs*	21	✓	✗	✓	✓	✓
sqlite3	`sqlite3_fuzz`	3.32.0	SQL queries	20	✓	✓	✗	✗	✓
php	`exif`, `json`, `parser`, `unserialize`	8.0.0–dev	*Various*	16	✓	✓	✗	✗	✗

sanitization overhead. This mode increases the number of executions during an evaluation, reducing the cost of evaluating a fuzzer but sacrificing the ability to evaluate a fuzzer's detection capabilities.

2.3.1 Target Selection

Magma contains seven targets, which we summarize in table 2.2. In addition to these seven *targets* (i.e., the codebases into which bugs are injected), Magma also includes 25 *drivers* (i.e., executable programs that provide a command-line interface to the target) that exercise different functionality within the target. Inspired by Google OSS-Fuzz [4], these drivers are sourced from the original target codebases (as drivers are best developed by domain experts).

Magma's seven targets were selected for their diversity in functionality (summarized qualitatively in table 2.2). Inspired by benchmarks in other fields [18, 76, 127, 129], we apply *Principal Component Analysis* (PCA) to quantify this diversity. PCA is a statistical analysis technique that transforms an N-dimensional space into a lower-dimensional space while preserving variance as much as possible [117]. Reducing high-dimensional data into a set of *principal components* allows for the application of visualization and/or clustering techniques to compare and discriminate benchmark workloads.

We apply PCA as follows. First, we use an Intel Pin [100] tool to record instruction traces for $K = 284$ *subjects* (i.e., a library wrapped with a particular driver program [98, 105]): four from LAVA-M, 14 from the FTS, 25 from Magma, and 241 from the CGC [155]. Each trace is driven by seeds provided by the benchmark (exercising functionality—and hence code—that would be explored by a fuzzer) and contains instructions executed by both the subject and any linked libraries. Second, instructions are categorized according to Intel XED, a disassembler built into

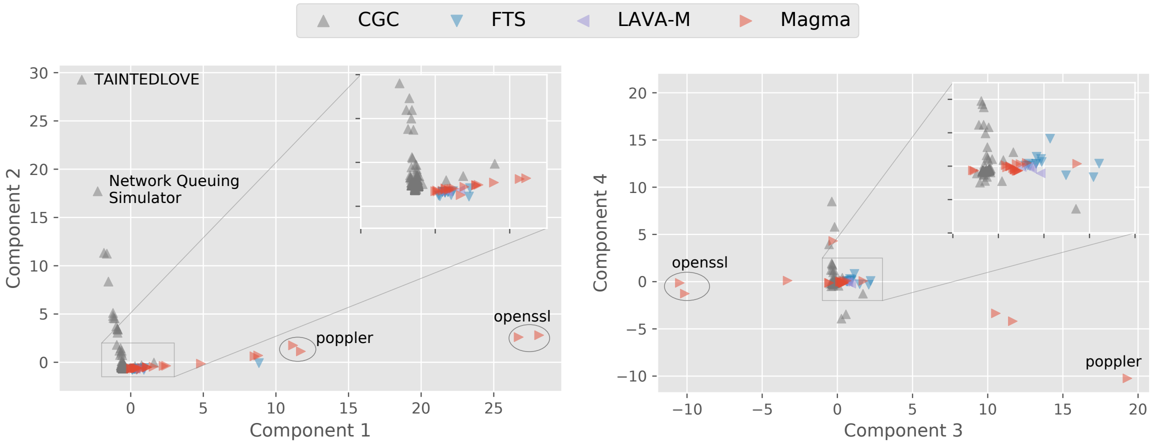


Figure 2.1 – Scatter plots of benchmark scores over the first four principal components (which account for ~60 % of the variance in the benchmark workloads). Each point corresponds to a particular subject in a benchmark.

Pin. A XED instruction category is "*a higher level semantic description of an instruction than its opcodes*" [70]. XED contains $N = 94$ instruction categories, spanning logical, floating point, syscall, and SIMD operations (amongst others). We use these categories as an approximation of the subject's functionality. Third, we create a matrix X, where $x_{ij} \in X$ $(i \in [1, N]$ and $j \in [1, K])$ is the mean number of instructions executed in a particular category for a given subject (over all seeds supplied with that subject). Finally, PCA is performed on a normalized version of X. The first four principal components, which in our case account for 60 % of the variance between benchmarks, are plotted in a two-dimensional space in fig. 2.1.

Figure 2.1 shows that the four LAVA-M workloads are tightly clustered over the first four principal components. This is unsurprising, given that the LAVA-M workloads are all sourced from coreutils and hence share the same codebase. In contrast, both the CGC and Magma provide a wide-variety of workloads. For example, *openssl*—which contains a large amount of cryptographic and networking code—appears distinct from the main clusters in fig. 2.1. The CGC's *TAINTEDLOVE* workload is similarly distinct, due to the relatively large number of floating point operations performed.

2.3.2 Bug Selection and Insertion

Magma contains 118 bugs, spanning 11 CWEs (summarized in fig. 2.2; the complete list of bugs is given in table A.1). Compared to existing benchmarks, Magma has both the second-largest variety of bugs (by CWE) and second-largest "bug density" (the ratio of the number of bugs to the number of targets) after the CGC and LAVA-M, respectively. While the CGC has a wider variety of

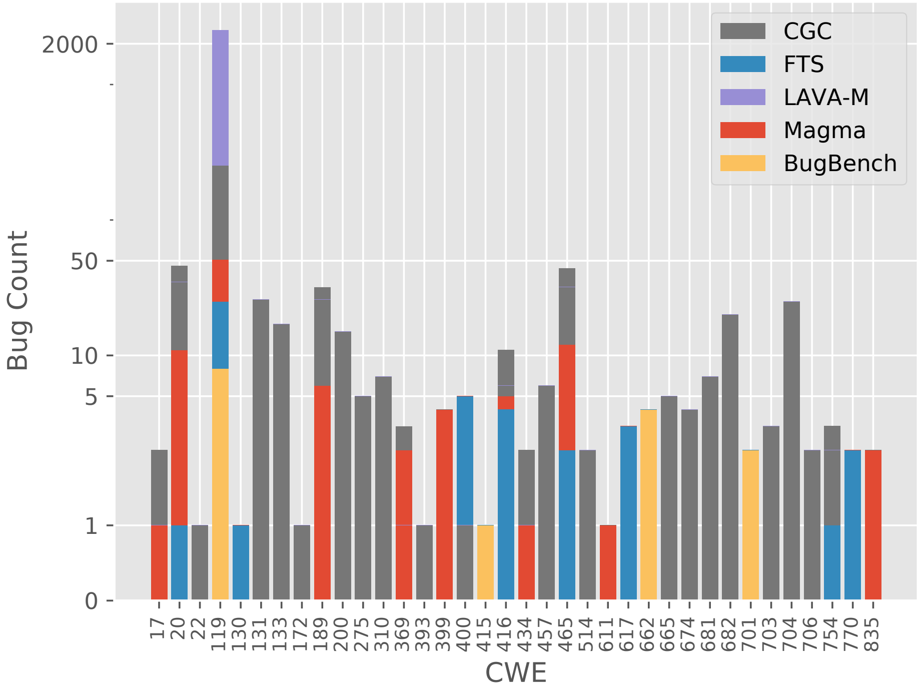


Figure 2.2 – Comparison of benchmark bug classes. The y-axis uses a log scale. A complete list of MAGMA bugs is presented in table A.1.

bugs, its workloads are not indicative of real-world software (in terms of both size and complexity). Similarly, while LAVA-M's bug density (566.25 bugs per target) is an order-of-magnitude larger than MAGMA's (16.86 bugs per target), LAVA-M is restricted to a single, synthetic bug type.

Importantly, MAGMA contains *real* bugs sourced from bug reports and *forward-ported* to the most recent version of the target codebase. This is in contrast to existing fuzzing benchmarks (e.g., BugBench, Google FTS) that rely on old, unpatched versions of the target codebase. Unfortunately, using older codebases limits the number of bugs available in each target (as evident by the low bug densities in table 2.1). In comparison, forward-porting—which is synonymous to *back-porting* fixes from newer codebases to older, buggy releases—does not suffer from this issue, making MAGMA's targets *easily extensible*.

Forward-porting begins with the identification—from the reported bug fix—of the code changes that must be reverted to reintroduce the bug. Bug-fix commits can contain multiple fixes to one or more bugs, so disambiguation is necessary to prevent the introduction of unintended bugs. Alternatively, bug fixes may be spread over multiple commits (e.g., if the original fix did not cover all edge cases). Following the identification of code changes, we identify what program state is involved in evaluating the trigger condition. If necessary, we introduce additional program variables to access that state. From this state, we determine a boolean expression that serves as a light-weight oracle for identifying a triggered bug. Finally, we identify a point in the program where we inject a canary before the bug can manifest faulty behavior. This canary helps measure our fuzzer performance metrics, discussed in the following section.

2.3.3 Performance Metrics

Fuzzer evaluation has traditionally relied on crash counts, bug counts, and/or code-coverage profiles for measuring and comparing fuzzer performance. While the problems with crash counts and code-coverage profiles are well known (see section 2.1.2), in our view, simply counting the number of bugs discovered is too coarse-grained. Instead, we argue that it is important to distinguish between *reaching*, *triggering*, and *detecting* a bug. Consequently, Magma uses these three bug-centric performance metrics to evaluate fuzzers.

Reached bugs

Definition 2.1 A *reached* bug refers to a bug whose oracle was called, implying that the executed path reaches the context of the bug, without necessarily triggering a fault. This is where coverage profiles fall short: simply covering the faulty code does not mean that the program is in the correct state to trigger the bug.

Triggered bugs

Definition 2.2 A *triggered* bug refers to a bug that was reached, and *whose triggering condition was satisfied*, indicating that a fault occurred. Whereas triggering a bug implies that the program has transitioned into a faulty state, the symptoms of the fault may not be directly observable at the oracle injection site. When a bug is triggered, the oracle only indicates that the conditions for a fault have been satisfied, but this does not imply that the fault was encountered or detected by the fuzzer.

Source-code instrumentation (i.e., the canary) provides ground-truth knowledge and runtime feedback of reached and triggered bugs. Each bug is approximated by (a) the lines of code patched in response to a bug report, and (b) a boolean expression representing the bug's trigger condition. The canary reports: (i) when the line of code is reached; and (ii) when the input satisfies the conditions for faulty behavior (i.e., triggers the bug). Section 2.4.4 discusses how we prevent canaries from leaking information to the system-under-test.

Finally, we also draw a distinction between *triggering* and *detecting* a bug. Whereas most security-critical bugs manifest as a low-level security policy violation for which state-of-the-art sanitizers are well-suited (e.g., memory corruption, data races, invalid arithmetic), other bug classes are not as easily observed. For example, resource exhaustion bugs are often detected long after the fault has manifested, either through a timeout or an out-of-memory error. Even more obscure are semantic bugs, whose malfunctions cannot be observed without a specification or reference. Consequently, various fuzzing techniques have been developed to target these bug classes (e.g., SlowFuzz [123] and NEZHA [122]). Such advancements in fuzzer techniques may benefit from an evaluation which includes the bug *detection* rate as another dimension for comparison.

2.3.4 Runtime Monitoring

MAGMA provides a runtime monitor that collects real-time statistics from the instrumented target. This provides a mechanism for visualizing the fuzzer's progress and its evolution over time, without complicating the instrumentation.

The runtime monitor collects data about reached and triggered bugs (section 2.3.3). Because this data primarily relates to the fuzzer's program exploration capabilities, we post-process the monitor's output to study the fuzzer's fault detection capabilities. This is achieved by replaying the crashing inputs (produced by the fuzzer) against the benchmark canaries to determine which bugs were triggered and hence detected. Importantly, it is possible that the fuzzer produces crashing inputs that do not correspond to any injected bug. If this occurs, the new bug is triaged and added to the benchmark for other fuzzers to discover.

2.4 Design and Implementation Decisions

MAGMA's unapologetic focus on fuzzing (as opposed to being a general bug-detection benchmark) necessitates a number of key design and implementation choices. We discuss these choices here.

2.4.1 Forward-Porting

Forward-Porting vs. Back-Porting

In contrast to back-porting bugs to previous versions, forward-porting ensures that all *known* bugs are fixed, and that the reintroduced bugs will have ground-truth oracles. While it is possible that the new fixes and features in newer codebases may (re)introduce unknown bugs, forward-porting allows MAGMA to evolve with each published bug fix. Additionally, future code changes may render a forward-ported bug obsolete, or make its trigger conditions unsatisfiable. Without verification, forward-porting may inject bugs which cannot be triggered. We use fuzzing to reduce this possibility, reducing the cost of manually verifying injected bugs. A fuzzer-generated PoV demonstrates that the bug is triggerable. Bugs that are discovered this way are added to the list of verified bugs, helping the evaluation of other fuzzers. While this approach may skew MAGMA towards fuzzer-discoverable bugs, we argue that this is a nonissue: any newly-discovered PoV will update the benchmark, thus ensuring a fair and balanced bug distribution.

Manual Forward-Porting

All MAGMA bugs are manually introduced. This process involves: (i) searching for bug reports; (ii) identifying bugs that affect the core codebase; (iii) finding the relevant fix commits; (iv) recognizing the bug conditions from the fix commits; (v) collecting these conditions as a set of path constraints; (vi) modeling these path constraints as a boolean expression (the bug canary); and (vii) injecting these canaries to flag bugs at runtime. The complexity of this process led us to reject a wholly-automated approach; automating bug injection would likely result in an incomplete and error-prone technique, ultimately yielding fewer bugs of lower quality. Moreover, an automated approach still requires manual verification of the results. Dedicating human resources to the forward-porting process maximizes the correctness of MAGMA's bugs.

To justify a manual approach, we enumerate the *scopes* (i.e., code blocks, functions, modules) spanned by each bug fix and use these scopes as a measure of bug-porting complexity (scope measures for all bugs are given in table A.1). While a simple bug-porting technique works well for fixes with a scope of one, the bug-porting technique must become more advanced as the number of scopes increases (e.g., it must handle *interprocedural* constraints). Of the 118 MAGMA bugs, 34 % had a scope measure greater than one.

Finally, our manual porting process was heavily reliant on prose; in particular, by the comments and discussions contained within bug reports. These discussions provide valuable insight into (a) developers' intent, and (b) the construction of precise trigger conditions. Additionally, function names (particularly those from the standard library) provide key insight into the code's objective, without requiring in-depth analysis into what each function does. An automated technique would require either: (i) an in-depth analysis of such functions, likely resulting in path explosion; or (ii) inference of bug conditions and function utilities via natural language processing (NLP). Both of these approaches are too complex to be included in the scope of MAGMA's development and would likely require several years of research to be effective.

2.4.2 Weird States

When a fuzzer generates an input that triggers an undetected bug, and execution continues past this bug, the program transitions into an undefined state: a *weird state* [42]. Any information collected after transitioning to a weird state is unreliable. To address this issue, we allow the fuzzer to continue the execution trace, but only collect bug oracle data *before and until* the first bug is triggered (i.e., transition to a weird state). Oracles do not signify that a bug has been executed; they only indicate whether the conditions required to execute a bug are satisfied.

Example 2.1 highlights the interplay between weird states. It shows two bugs: an out-of-bounds write (bug 1) and a division-by-zero (bug 2). When `tmp.len == 0`, the condition for bug 1 (line 5) remains unsatisfied, logging and triggering bug 2 instead (lines 6 and 7, respectively).

When `tmp.len > 16`, bug 1 is logged and triggered on lines 4 and 5, and `tmp.len` is overwritten by a non-zero value, leaving bug 2 untriggered.

However, bug 1 is triggered when `tmp.len == 16`, overwriting `tmp.len` with the `NULL` terminator and setting its value to 0 (on a Little-Endian system). This also triggers bug 2, despite the input not explicitly specifying a zero-length `str`.

Escaping semantics

Example 2.1

```
1 void libfoo_baz(char *str) {
2   struct { char buf[16]; size_t len; } tmp;
3   tmp.len = strlen(str);
4   magma_log(1, tmp.len >= sizeof(tmp.buf)); // Bug 1: possible OOB write in strcpy()
5   strcpy(tmp.buf, str);
6   magma_log(2, tmp.len == 0); // Bug 2: possible div-by-zero
7   int repeat = 64 / tmp.len;
8 }
```

Listing 2.1 – Weird states can result in execution traces which do not exist in the context of normal program behavior.

2.4.3 A Static Benchmark

Much like other widely-used performance benchmarks—e.g., SPEC CPU [150] and DaCapo [18]—Magma is a *static* benchmark that contains realistic workloads. These benchmarks assume that if the system-under-test performs well on the benchmark's workloads, then it will perform similarly on real workloads. While realistic, static benchmarks are susceptible to *overfitting*. Overfitting can occur if developers tweak the system-under-test to perform better on a benchmark, rather than focusing on real workloads.

Overfitting could be overcome by *dynamically synthesizing* a benchmark (and ensuring that the system-under-test is unaware of the synthesis parameters). However, this approach risks generating workloads different from real-world scenarios, rendering the evaluation biased and/or incomplete.

While program synthesis is a well-studied topic [15, 62, 72], it remains difficult to generate large programs that remain faithful to real development patterns and styles.

To prevent overfitting, Magma's forward-porting process allows targets to be updated as they evolve in the real-world. Each forward-ported bug requires minimal code changes: the addition of Magma's instrumentation and the faulty code itself. This makes it relatively straightforward to update targets, including introducing new bugs and new features. For example, two undergraduate students without software security experience added over 60 bugs in three new targets over a single semester. These measures ensure that Magma remains representative of real, complex targets and suitable for fuzzer evaluation.

2.4.4 Leaky Oracles

Introducing oracles into the benchmark may leak information that interferes with a fuzzer's exploration capability, potentially leading to overfitting (as discussed in section 2.4.3). For example, if oracles were implemented as `if` statements, fuzzers that maximize branch coverage could detect the oracle's branch and hence generate an input that satisifies the branch condition.

One possible solution to this *leaky oracle* problem is to produce both instrumented and uninstrumented target binaries (with respect to Magma's instrumentation, not any instrumentation that the fuzzer injects). The fuzzer's input would be fed into both binaries, but the fuzzer would only collect the data it needs (e.g., coverage feedback) from the uninstrumented binary. The instrumented binary would collect canary data and report it to the runtime monitor. This approach, however, introduces other challenges associated with duplicating the execution trace between two binaries (e.g., replicating the environment, maintaining synchronization between executions), greatly complicating Magma's implementation and introducing runtime overheads.

Instead, we use *always-evaluate memory writes*, whereby an injected bug oracle evaluates a boolean expression representing the bug's trigger condition. This typically involves a binary comparison operator, which most compilers (e.g., gcc, clang) translate into a pair of `cmp` and `set` instructions embedded into the execution path. The results of this evaluation are then shared with the runtime monitor (section 2.3.4). This process is demonstrated in listings 2.2 and 2.3.

```c
1 void magma_log(int id, bool cond) {
2   extern struct magma_bug *bugs;
3   extern bool faulty; // init := false
4   bool mask = faulty ^ 1;
5   bugs[id].reached   += 1 & mask;
6   bugs[id].triggered += cond & mask;
7   faulty = faulty | cond;
8 }
```

Listing 2.2 – MAGMA instrumentation.

```c
1 void libfoo_bar() {
2   // uint32_t a, b, c;
3   magma_log(BUG_ID,
4     (a == 0) | (b == 0));
5   // possible divide-by-zero
6   uint32_t x = \
7     c / (a * b);
8 }
```

Listing 2.3 – Instrumented example.

Listing 2.2 shows MAGMA's canary implementation. The always-evaluated memory accesses are shown on lines 5 and 6. The `faulty` flag addresses the problem of weird states (section 2.4.2), and disables future canaries after the first bug is encountered.

Listing 2.3 shows an example program instrumented with a canary. A call to `magma_log` is inserted (line 3) prior to the execution of the faulty code (line 7). Compound trigger conditions—i.e., those including the logical `and` and `or` operators—often generate implicit branches at compile-time (due to short-circuit compiler behavior). To avoid leaking information through coverage, we provide custom x86-64 assembly blocks to evaluate these logical operators in a single basic block (without short-circuit behavior). We revert to C's bitwise operators (`&` and `|`)—which are more brittle and susceptible to safety-agnostic compiler passes [149]—when the compilation target is not x86-64.

Although this approach may introduce memory access patterns that are detectable by taint tracking and other data-flow analysis techniques, statistical tests can be used to infer whether the fuzzer overfits to these access patterns. By repeating the fuzzing campaign with the uninstrumented binary, we can verify if the results vary significantly.

2.4.5 Proofs of Vulnerability

In order to increase confidence in the injected bugs, a proof of vulnerability (PoV) input must be supplied for every bug, verifying that the bug can be triggered. The process of manually crafting PoVs, however, is arduous and requires domain-specific knowledge, both about the input format and the target program, potentially bringing the bug-injection process to a grinding halt.

When available, we extract PoVs from public bug reports. When no PoV is available, we launch multiple fuzzing campaigns against these targets in an attempt to trigger each injected bug. Inputs that trigger a bug are saved as a PoV. Bugs which are not triggered, even after multiple campaigns, are manually inspected to verify path reachability and satisfiability of trigger conditions.

2.4.6 Unknown Bugs

Because MAGMA uses real-world programs, it is possible that bugs exist for which no ground-truth is available (i.e., an oracle does not exist). A fuzzer might inadvertantly trigger these bugs and (correctly) detect a fault. Due to the imperfections in automated deduplication techniques, these crashes are not included in MAGMA's metrics. Instead, such crashes are used to improve MAGMA itself. The bug's root cause can be determined by manually studying the execution trace, after which the bug can be added to the benchmark.

2.4.7 Fuzzer Compatibility

Fuzzers are not limited to a specific execution engine under which they analyze and explore a program. For example, some fuzzers (e.g., Driller [152], T-Fuzz [121]) leverage symbolic execution (using an engine such as angr [146]) to explore the target. This can introduce (a) incompatibilities with MAGMA's instrumentation, and (b) inconsistencies in the runtime environment (depending on how the symbolic execution engine models the environment).

However, the defining trait of most fuzzers, in contrast to other types of bug-finding tools, is that they concretely execute the target on the host system. Unlike benchmarks such as the CGC and BugBench—which aim to evaluate *all* bug-finding tools—MAGMA is unapologetically a *fuzzing* benchmark. This includes whitebox fuzzers that use symbolic execution to guide input generation, provided that the target is executed on the host system (SYMCC [128] is one such fuzzer that we include in our evaluation).

We therefore impose the following restriction on the fuzzers evaluated by MAGMA: the fuzzer must execute the target in the context of an OS process, with unrestricted access to OS facilities (e.g., system calls, libraries, file system). This allows MAGMA's runtime monitor to extract canary statistics using the operating system's services at relatively low overhead/complexity.

2.5 Evaluation

2.5.1 Methodology

We evaluated several fuzzers in order to establish the versatility of our metrics and benchmark suite. We chose a set of seven *mutational fuzzers* whose source code was available at the time of writing: AFL [171], AFLFast [22], AFL++ [49], FAIRFUZZ [90], MOPT-AFL [102], honggfuzz [153], and SYMCC-AFL [128]. These seven fuzzers were evaluated over ten identical 24 h and 7 d fuzzing campaigns for each fuzzer/target combination. This amounts to 200,000 CPU-hours of fuzzing.

To ensure fairness, benchmark parameters were identical across all fuzzing campaigns. Each fuzzer was bootstrapped with the same set of seed files (sourced from the original target codebase) and configured with the same timeout and memory limits. MAGMA's monitoring utility was configured to poll canary information every five seconds, and *fatal canaries* mode (section 2.3) was used to evaluate a fuzzer's ability to *reach* and *trigger* bugs. All experiments were run on one of three machines, each with an Intel® Xeon® Gold 5218 CPU and 64 GB of RAM, running Ubuntu 18.04 LTS 64-bit. The targets were compiled for x86-64.

AddressSanitizer (ASan) [143] was used to evaluate *detected* bugs. Crashing inputs (generated by fatal canaries) were validated by replaying them through the ASan-instrumented target. Although this evaluation method measures ASan's fault-detection capabilities, it still highlights the bugs that fuzzers can realistically detect when fuzzing without ground truth.

2.5.2 Time to Bug

We use the time required to find a bug as a measure of fuzzer performance. As discussed in section 2.3.3, MAGMA records the time taken to both reach and trigger a bug, allowing us to compare fuzzer performance across multiple dimensions. Fuzzing campaigns are typically limited to a finite duration (we limit our campaigns to 24 h and 7 d, repeated ten times), so it is important that the time-to-bug discovery is low.

The highly-stochastic nature of fuzzing means that the time-to-bug can vary wildly between identical trials. To account for this variation, we repeat each trial ten times. Despite this repetition, a fuzzer may still fail to find a bug within the alloted time, leading to missing measurements. We therefore apply *survival analysis* to account for this missing data and high variation in bug discovery times. Specifically, we adopt Wagner's approach [157] and use the Kaplan-Meier estimator [77] to model a bug's *survival function*. This survival function describes the probability that a bug remains undiscovered (i.e., "survives") within a given time (here, 24 h and 7 d trials). A smaller survival time indicates better fuzzer performance.

2.5.3 Experimental Results

Figures 2.3 and 2.4 and tables A.2 and A.3 present the results of our fuzzing campaigns.

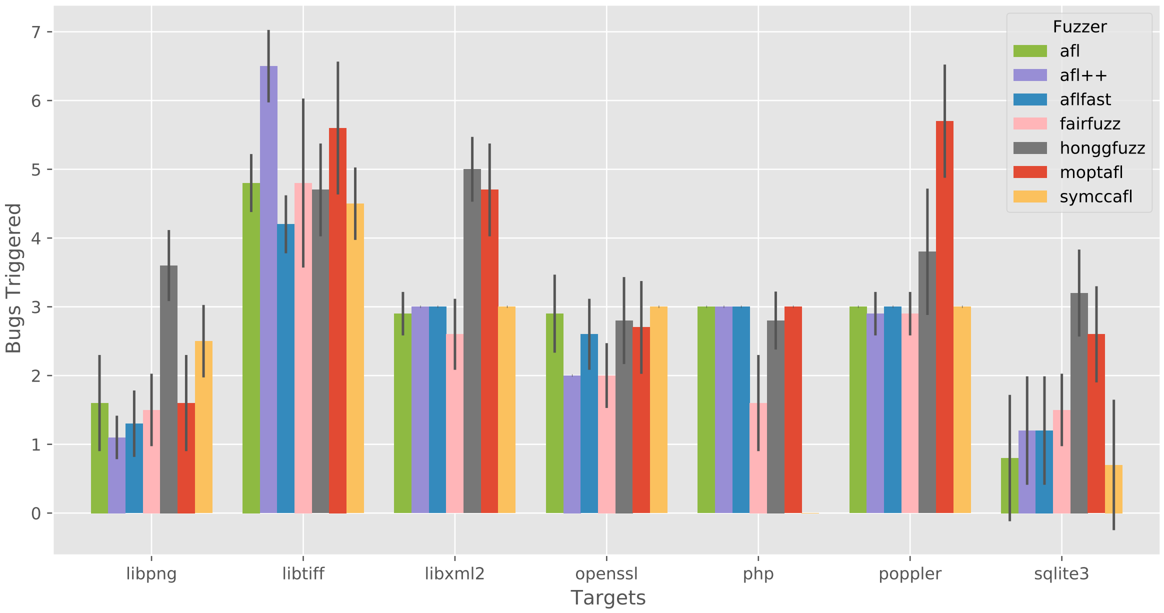


Figure 2.3 – The mean number of bugs (and standard deviation) found by each fuzzer across ten 24 h campaigns.

Bug Count and Statistical Significance

Figure 2.3 shows the mean number of bugs found per fuzzer (across ten 24 h campaigns). These values are susceptible to outliers, limiting the conclusions that we can draw about fuzzer performance. We therefore conducted a statistical significance analysis of the collected sample-set pairs to calculate p-values using the Mann-Whitney U-test. P-values provide a measure of how different a pair of sample sets are, and how significant these differences are. Because our results are collected from independent populations (i.e., different fuzzers), we make no assumptions about their distributions. Hence, we apply the Mann-Whitney U-test to measure statistical significance. Figure 2.4 shows the results of this analysis.

The Mann-Whitney U-test shows that AFL, AFLFast, AFL++, and SYMCC-AFL performed similarly against most targets (signified by the large number of red and white cells in fig. 2.4), despite some minor differences in mean bug counts (shown in fig. 2.3). Figure 2.4 shows that, in most cases, the small fluctuations in mean bug counts are not significant, and the results are thus not sufficiently conclusive. One oddity is the performance of AFL++ against *libtiff*. Figure 2.3 reveals that AFL++ scored the highest mean bug count compared to all other fuzzers, and fig. 2.4 shows that this difference is statistically significant.

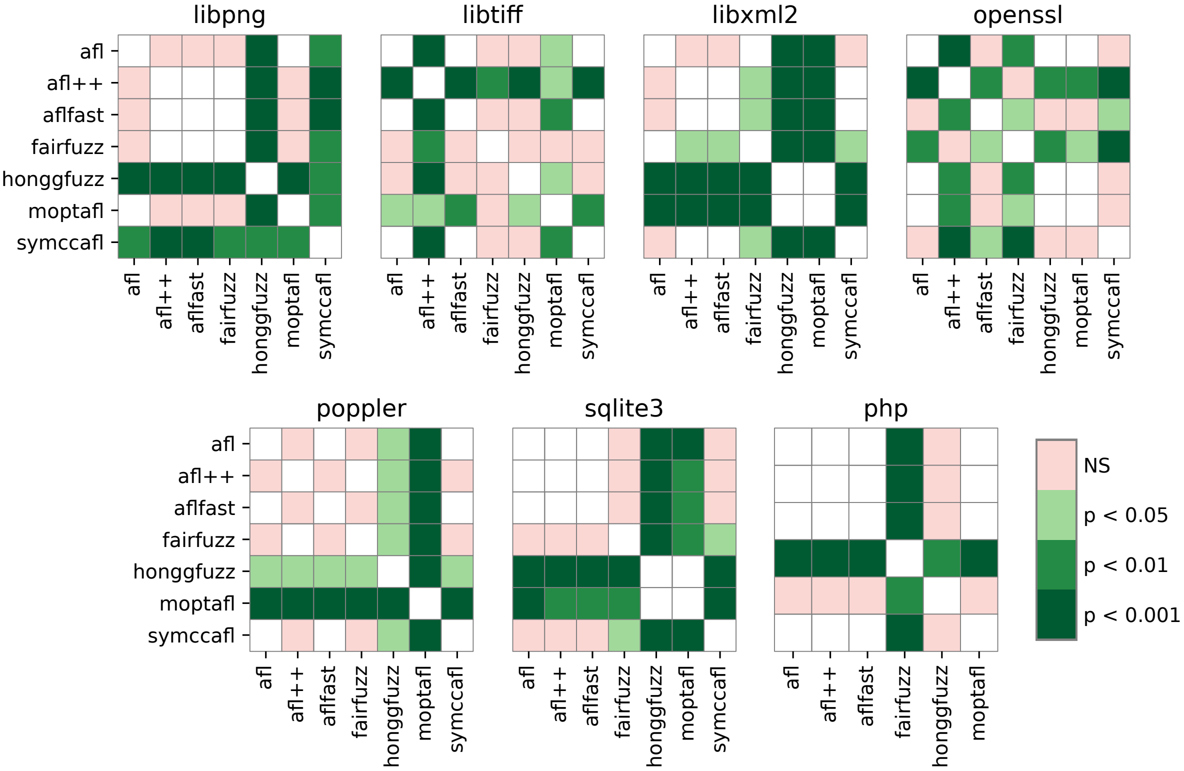


Figure 2.4 – Significance of evaluations of fuzzer pairs using p-values from the Mann-Whitney U-Test. We use $p < 0.05$ as a threshold for significance. Values greater than 0.05 are shaded red. Darker shading indicates a lower p-value, or higher statistical significance. White cells indicate that the pair of sample sets are identical.

On the other hand, FAIRFUZZ [90] displayed significant performance regression against *libxml2*, *openssl*, and *php*. While the original evaluation of FAIRFUZZ claims that it achieved the highest coverage against `xmllint`, that improvement was not reflected in our results.

Finally, honggfuzz and MOPT-AFL performed significantly better than all other fuzzers in three out of seven targets. Additionally, honggfuzz was the best fuzzer for *libpng* as well. We attribute honggfuzz's performance to its wrapping of memory-comparison functions, which provides comparison progress information to the fuzzer (similar to Steelix [92]).

Time to Bug

In total, during the 24 h campaigns, 74 of the 118 MAGMA bugs (62 %) were reached. Additionally, 43 of the 54 *verified* bugs (79 %)—i.e., those with PoVs—were triggered. Notably, no single fuzzer triggered more than 37 bugs (68 % of the verified bugs). These results are presented in table A.2. Here, bugs are sorted by the mean trigger time, which we use to approximate "difficulty".

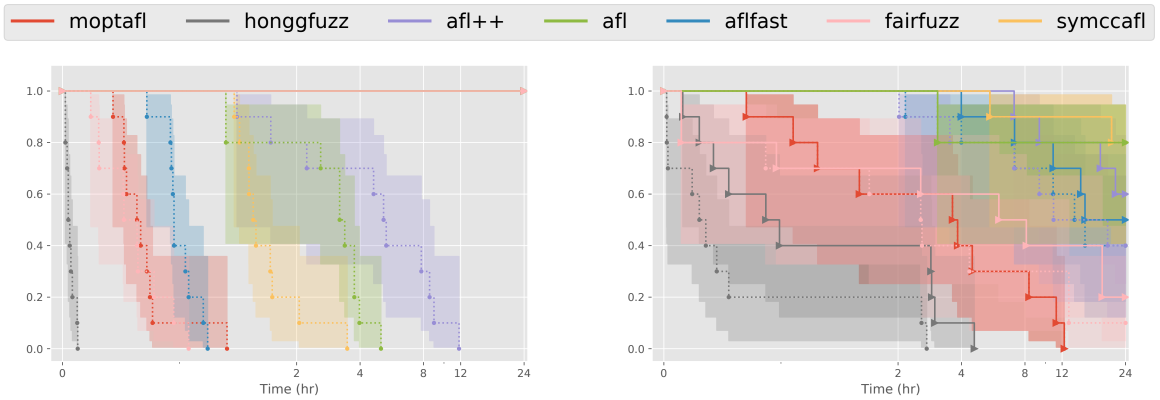
The long bug discovery times (19 of the 43 triggered bugs—44 %—took on average more than 20 h to trigger) suggests that the evaluated fuzzers still have a long way to go in improving program exploration. However, while many of the MAGMA bugs are difficult to discover, table A.2 highlights a set of 17 "simple" bugs that all fuzzers find consistently within 24 h. These bugs provide a baseline for detecting performance regression: if a new fuzzer fails to discover these bugs, then its program exploration strategy should be revisited.

Most of the bugs in table A.2 were reached by all fuzzers. SYMCC-AFL was the worst performing fuzzer in this regard, failing to reach nine bugs (the highest amongst the seven evaluated fuzzers). Interestingly, most bugs show a large difference between reach and trigger times. For example, only the first three bugs listed in table A.2 were triggered when first reached. In contrast, bugs such as MAE115 (from *openssl*) take 10 s to reach (by all fuzzers), but up to 20 h (on average) to trigger. This difference between time-to-reach and time-to-trigger a bug provides another feature for determining bug "difficulty": while control flow may be trivially satisfied (as evidence by the time to reach a bug), bugs such as MAE115 may require complex, stateful data-flow constraints.

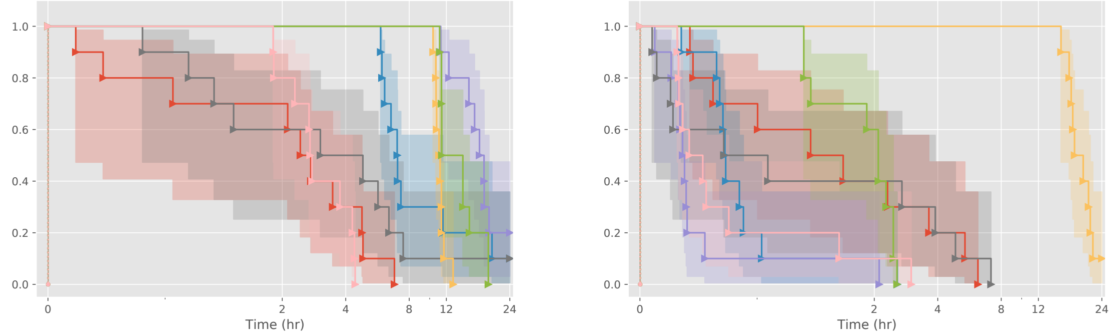
The longer, 7 d campaigns in table A.3 reveal a peculiar result: while honggfuzz was faster to trigger bugs during the 24 h campaigns, MOPT-AFL was faster to trigger 11 additional bugs after 24 h, making it the most successful fuzzer over the 7 d campaigns. Notably, honggfuzz failed to trigger any of these 11 bugs. This highlights the importance of long fuzzing campaigns and the utility of MAGMA's survival time analysis for comparing fuzzer performance.

Figure 2.5 plots four survival functions for three MAGMA bugs (AAH018, JCH232, and AAH020). These plots illustrate the probability of a bug surviving a 24 h fuzzing trial, and are generated by applying the Kaplan-Meier estimator to the results of ten repeated fuzzing trials. Dotted lines represent survival functions for *reached* bugs, while solid lines represent survival functions for *triggered* bugs. Confidence intervals are shown as shaded regions. Figure 2.5a shows the time to reach bug AAH018 (*libtiff*). Notably, this bug was not triggered by any of the seven evaluated fuzzers. Thus, the probability of bug AAH018 "surviving" 24 h (i.e., not being triggered) remains at one. In comparison, fig. 2.5b shows the differences in the time taken to reach and trigger bug JCH232 (*sqlite3*). Here, honggfuzz is the best performer, because the bug's probability of survival approaches zero the fastest. Notably, the variance is much higher compared to bug AAH018 (as evident by the larger confidence intervals). Finally, figs. 2.5d and 2.5c compare the probability of survival for bug AAH020 (*libtiff*) across two driver programs: `tiffcp` and `read_rgba_fuzzer`. The former is a general-purpose application, while the latter is a driver specifically designed as a fuzzer harness. While the bug is reached relatively quickly by both drivers, the fuzzer harness is clearly superior at *triggering* the bug, as it is faster across *all* fuzzers. This result supports our claim in section 2.3.1 that *domain experts are most suitable for selecting and developing fuzzing drivers.*

Again, it is clear that honggfuzz outperforms all other fuzzers (in both reaching and triggering bugs), finding 11 additional bugs not triggered by other fuzzers. In addition to its finer-grained



(a) Bug AAH018 (*libtiff* with `read_rgba_fuzzer`). (b) Bug JCH232 (*sqlite3* with `sqlite3_fuzz`).



(c) Bug AAH020 (*libtiff* with `tiffcp`). (d) Bug AAH020 (*libtiff* with `read_rgba_fuzzer`).

Figure 2.5 – Survival functions for a subset of MAGMA bugs. The y-axis is the *survival probability* for the given bug. Dotted lines represent survival functions for *reached* bugs, while solid lines represent survival functions for *triggered* bugs. Confidence intervals are shown as shaded regions.

instrumentation, honggfuzz natively supports persistent fuzzing. Our experiments show that honggfuzz's execution rate was at least three times higher than that of AFL-based fuzzers using persistent drivers. This undoubtedly contributes to honggfuzz's strong performance.

Achilles' Heel of Mutational Fuzzing

AAH001 (CVE-2018-13785, shown in listing 2.4), is a divide-by-zero bug in *libpng*. It is triggered when the input is a non-interlaced 8-bit RGB image with a width of `0x55555555`. This "magic value" is not encoded anywhere in the target, and is easily calculated by solving the constraints for `row_factor == 0`. However, mutational fuzzers struggle to discover this bug type. This is because mutational fuzzers sample from an extremely large input space, making them unlikely to pick the exact byte sequence required to trigger the bug (here, `0x55555555`). Notably, only honggfuzz, AFL, and SymCC-AFL were able to trigger this bug. SymCC-AFL was the fastest to do so, likely due to its constraint-solving capabilities.

Magic Value Identification

AAH007 is a dangling pointer bug in *libpng*, and illustrates how some fuzzer features improve bug-finding ability. To trigger this bug, it is sufficient for a fuzzer to provide a valid input with an `eXIF` chunk (which remains unmarked for release upon object destruction, leading to a dangling pointer). Unlike the AFL-based fuzzers, honggfuzz is able to consistently trigger this bug relatively early in each campaign. We posit that this is due to honggfuzz replacing the `strcmp` function with an instrumented wrapper that incrementally satisfies string magic-value checks. SymCC-AFL also consistently triggers this bug, demonstrating how whitebox fuzzers can trivially solve constraints based on magic values.

Semantic Bug Detection

AAH003 (CVE-2015-8472) is a data inconsistency in `libpng`'s API, where two references to the same piece of information (color-map size) can yield different values. Such a semantic bug does not produce observable behavior that violates a known security policy, and it cannot be detected by state-of-the-art sanitizers without a specification of expected behavior.

Semantic bugs are not always benign. Privilege escalation and command injection are two of the most security-critical logic bugs that are still found in modern systems, but they remain difficult to detect with standard sanitization techniques. This observation highlights the shortcomings of current fault detection mechanisms and the need for more fault-oriented bug-finding techniques (e.g., NEZHA [122]).

```
void png_check_chunk_length(png_ptr, length) {
  size_t row_factor = png_ptr->width // uint32_t
    * png_ptr->channels // uint32_t
    * (png_ptr->bit_depth > 8? 2: 1)
    + 1
    + (png_ptr->interlaced? 6: 0);

  if (png_ptr->height > UINT_32_MAX/row_factor) {
    idat_limit = UINT_31_MAX;
  }
}
```

Listing 2.4 – Divide-by-zero bug in *libpng*. Input undergoes non-trivial transformations to trigger the bug.

Table 2.3 – Overheads introduced by LAVA-M compared to `coreutils-8.24`. These overheads denote increases in LLVM IR instruction counts, object file sizes, and average runtimes when processing seeds generated from a 24 h fuzzing campaign. The total number of unique bugs triggered across all 10 trials/fuzzer is also shown, with the best performing fuzzer highlighted in green.

Target	Bugs	Overheads (%)			Total bugs triggered (#)						
		LLVM IR	Size	Runtime	afl	aflfast	afl++	moptafl	fairfuzz	honggfuzz	symccafl
base64	44	107.9	57.2	9.7	1	0	48	0	3	33	0
md5sum	57	60.2	46.1	9.5	0	1	40	1	1	29	0
uniq	28	63.6	27.8	11.6	3	0	29	1	0	13	3
who	2136	1786.7	2409.1	42.9	1	1	819	1	1	750	1

Comparison to LAVA-M

In addition to our MAGMA evaluation, we also evaluate the same seven fuzzers against LAVA-M, measuring (a) the overheads introduced by LAVA-M's bug oracles, and (b) the total number of bugs found by each fuzzer (across a 24 h campaign, repeated 10 times per fuzzer). These results—presented in table 2.3—show that LAVA-M's most iconic target, *who*, accounts for 94.3 % of the benchmark's bugs. This high bug count reduces the amount of functional code (compared to benchmark instrumentation) in the *who* binary to 5.3 %, impeding a fuzzer's exploration capabilities. Notably, we found that the evaluated fuzzers spent (on average) 42.9 % of their time executing oracle code in *who* (this percentage is based on the final state of the fuzzing queue, and may not represent the runtime overhead of *all* code paths). Finally, the bug counts found by each fuzzer show a clear bias towards fuzzers with magic-value detection capabilities (due to LAVA-M's single, simple bug type, per section 2.1.2).

2.5.4 Discussion

Ground Truth and Confidence

Ground truth enables us to determine a crash's root cause. Unlike many existing benchmarks, MAGMA provides straightforward access to ground truth. While ground truth is available for all 118 bugs, only 45 % of these bugs have a PoV that demonstrate triggerability. Importantly, only bugs with PoVs can be used to confidently measure a fuzzer's performance. Regardless, bugs without a PoV remain useful: any fuzzer evaluated against MAGMA can produce a PoV, increasing the benchmark's utility. Widespread adoption of MAGMA will increase the number of bugs with PoVs. Notably, table A.3 shows that running the benchmark for longer indeed yields more PoVs for previously-untriggered bugs. We leave it as an open challenge to generate PoVs for these bugs.

Beyond Crashes

While MAGMA's instrumentation does not collect information about *detected* bugs (detection is a characteristic of the fuzzer, not the bug itself), it does enable the evaluation of this metric through a post-processing step (supported by fatal canaries).

In particular, bugs should not be restricted to crash-triggering faults. For example, some bugs result in resource starvation (e.g., unbounded loops or `mallocs`), privilege escalation, or undesirable outputs. Importantly, fuzzer developers recognize the need for additional bug-detection mechanisms: AFL has a hang timeout, and SlowFuzz searches for inputs that trigger worst-case behavior. Excluding non-crashing bugs from an evaluation leads to an under-approximation of real bugs. Their inclusion, however, enables better bug detection tools. Evaluating fuzzers based on bugs *reached*, *triggered*, and *detected* allows us to classify fuzzers and compare different approaches along multiple dimensions (e.g., bugs reached allows for an evaluation of path exploration, while bugs triggered and detected allows for an evaluation of a fuzzer's constraint generation/solving capabilities). It also allows us to identify which bug classes continue to evade state-of-the-art sanitization techniques (and to what degree).

MAGMA as a Lasting Benchmark

MAGMA leverages software with a long history of security bugs to build an extensible framework with ground truth knowledge. Like most benchmarks, the widespread adoption of MAGMA defines its utility. Benchmarks provide a common basis through which systems are evaluated and compared. For instance, the community continues to use LAVA-M to evaluate and compare fuzzers, despite the fact that most of its bugs have been found, and that these bugs are of a single, synthetic type. MAGMA aims to provide an evaluation platform that incorporates realistic bugs in real software.

2.6 MAGMA Summary

MAGMA is an open ground-truth fuzzing benchmark that enables accurate and consistent fuzzer evaluation and performance comparison. We designed and implemented MAGMA to provide researchers with a benchmark containing *real* targets with *real* bugs. We achieve this by forward-porting 118 bugs across seven diverse targets. However, this is only the beginning. MAGMA's simple design and implementation allows it to be easily improved, updated, and extended, making it ideal for open-source collaborative development and contribution. Increased adoption will only strengthen MAGMA's value, and thus we encourage fuzzer developers to incorporate their fuzzers into MAGMA.

We evaluated Magma against seven popular open-source mutation-based fuzzers (AFL, AFLFast, AFL++, FairFuzz, MOpt-AFL, honggfuzz, and SymCC-AFL). Our evaluation shows that ground truth enables systematic comparison of fuzzer performance. Our evaluation provides tangible insight into fuzzer performance, why crash counts are often misleading, and how randomness affects fuzzer performance. It also brought to light the shortcomings of some existing fault detection methods used by fuzzers.

Despite best practices, evaluating fuzz testing remains challenging. With the adoption of ground-truth benchmarks like Magma, fuzzer evaluation will become reproducible, allowing researchers to showcase the true contributions of new fuzzing approaches.

Chapter 3

IGOR: Crash De-duplication Through Root-Cause Clustering

> Chaos is order yet undeciphered.
>
> *José Saramago*

This chapter delves into the challenges of accurately identifying and grouping software bugs when dealing with fuzzer-generated bug reports. It scrutinizes the limitations of current sanitizers and automatic crash de-duplication methods, which often overwhelm developers with redundant or misleading information. Through IGOR, we introduce a novel technique for crash de-duplication based on execution trace clustering, aimed at mitigating bug-count inflation and aiding developers in the triage process.

Hypothesis 2

To trigger a bug, it is necessary to execute its code. Bug-triggering inputs can thus be distinguished by the code they execute.

The contents of this chapter are adapted from Jiang et al. [75].

The main focus of software-testing research is finding bugs. Maximizing bug discovery is a key subject of interest across the development stack, from the physical layer [28, 120, 130, 163] to the application layer [12, 29, 98, 133, 153, 171]. The assumption "we can always afford to fix bugs" powers the drive for bug-finding techniques that yield large numbers of crashes in short time frames, the most prominent of which is fuzz testing. Big players in the software industry also motivate this movement, providing open-access reporting platforms [7, 53, 111] and bug bounties for their large user-bases. With little incentive to triage crashes, users and software testers frequently submit raw findings/crash-dumps, leaving it to software maintainers to bear the weight of distilling crashes and fixing bugs. With more crashes reported, more time is spent on crash triage and pruning duplicate reports, leaving maintainers with less time for fixing bugs and improving software quality. Large fuzzing farms (e.g., ClusterFuzz [56], OSS-Fuzz [142])—which run around the clock and automatically submit crash reports—exacerbate this problem. For example, as of late April 2021, there were 979 open Linux kernel bugs, with the earliest submitted in November 2017 (based on syzbot statistics [59]).

Solutions to this problem range from collaborative, where maintainers rely on the community to provide actionable analysis in their reports, to systematic, where heuristics are used over large crash dumps to filter out redundancies and duplicates [3, 34, 36, 80, 167]. These heuristics typically rely on dynamic program behaviors to identify root causes and answer the question "given a crashing test case, what is the most likely cause for the crash?" For example, AURORA [19] presents a root cause identification method based on *delta debugging* [174]: both faulty and benign test cases are executed and a disjunction in program behaviors marks the bug. In contrast, *crash bucketing* (*cf.* grouping) shifts the focus from pinpointing the cause of a crash to grouping crashes based on their root cause. Crash bucketing answer the question "given a number of crashing test cases, which crashes trigger the same bug?" Crash bucketing techniques are widely used in practice [83] and rely on *crash sites*, *coverage profiles*, or *stack hashes*, to cluster crashes.

Crash Sites All bugs crash at a particular program location. These crash sites (e.g., the address stored in the instruction pointer at the time of crash) serve as a coarse-grain bug identifier. Unfortunately, crash sites are imprecise and lead to bug misclassification (e.g., for use-after-free bugs, objects may be arbitrarily reused, triggering a broad set of "unique" crashes). Crash sites both under- and over-estimate bug counts and are not used in practice.

Coverage Profiles Coverage-guided fuzzers commonly use their coverage data to uniquely identify crashes. For example, AFL [171] considers crashes that exercise new control-flow edges or omit common edges as "unique". The fine-grained nature of edge coverage makes it sensitive to small changes in control-flow and causes crash count inflation; slight modifications in the path to a bug result in a new crash. Coverage profiles overestimate bug counts by 2–3 orders of magnitude [83].

Stack Hashes Stack hashes provide function-sensitive labels for crashes and are commonly used by fuzzers (e.g., honggfuzz [153]) and fuzzing farms (e.g., ClusterFuzz [56]) alike. Stack hashing accumulates (the last N) function calls (on the stack) leading to the crash site, hashing these traces to form unique bug identifiers. Compared to coverage profiling, stack hashing is more coarse-grained and results in fewer crash buckets. However, stack hashes are prone to misclassification, both over- and under-approximating the number of bugs (the former is due to different paths to the crash site [19, 37], while the latter is due to bugs sharing the same call sequence). Stack hashes overestimate bug counts by 1–2 orders of magnitude [83].

Inaccurate crash grouping techniques (such as those previously described) waste precious developer time. In particular, bug identifiers (e.g., coverage profiles, stack hashes) are susceptible to fluctuations in program behavior. Coverage-guided fuzzers—which typically aim to maximize code coverage—amplify such fluctuations, yielding many crashing test cases that exercise "noisy" code extraneous to the underlying root cause. This noise impedes control-flow-based de-duplication techniques, inflating bug counts. Moreover, accurate bug classification relies on identifying a bug's *characteristic trigger*. Identifying this trigger in a partial execution trace is made difficult if (a) the trigger is missing from the execution trace (e.g., if the trace is too coarse-grained), and/or (b) the execution trace contains extraneous elements (e.g., if the trace is too fine-grained). This calls for a new approach to crash grouping; one that removes noise from the execution trace—to accurately group paths—and preserves critical control-flow information—to accurately isolate different root causes.

Whereas a coverage-maximizing strategy is ideal for finding bugs through fuzzing, we propose that the counterpart—*coverage-minimizing fuzzing*—is key to minimizing an execution trace and enabling effective crash grouping. We make crash labels more precise by trimming unnecessary execution trace elements (i.e., noise), leading to more concise de-duplication. We also address the shortcomings of stack hashes (which operate at function-call granularity) by using control-flow graphs (CFG) for a more complete view of a crash's execution trace. Using CFGs preserves critical control-flow information and allows for aggressive pruning of redundant (executed) code, leading to more accurate crash grouping.

We present IGOR[1], a dual-phase crash de-duplication technique that leverages a coverage-reduction fuzzer and a CFG similarity metric to cluster crashes by their critical behaviors. By simplifying each crash's execution trace, we obtain test cases that exercise the minimized behavior necessary for triggering a bug. Then, we perform a graph similarity comparison over the CFGs of all minimized execution traces to group them into closely-packed clusters, each mapping back to a unique root cause.

---

[1] In Terry Pratchett's Ankh-Morpork, the *Igors* are a group of humble professional servants (often to mad scientists) that are proficient transplant surgeons.

We answer the following research questions:

RQ1 What constitutes ideal crash grouping and why is it not achievable in practice?
RQ2 How strong is the effect of dense execution traces on crash-count inflation, and can sparsity promote precision?
RQ3 What metric is best suited for capturing and isolating root causes?

We make the following contributions:

- a *coverage-reduction* fuzzer which applies a *minimizing fitness function* over the collected edge coverage to shrink test cases;
- a *new metric for crash grouping*, based on control flow graph similarity using the Weisfeiler-Lehman Subtree Kernel algorithm [85]; and
- a *ground-truth benchmark* for evaluating crash grouping techniques, containing 52 CVEs[2] and 254,000 crashing test cases from 14 real world programs (generated over 58.7 CPU-years of fuzzing).

3.1 Background

Crash bucketing groups test cases to isolate a crash's root cause. Accurate crash bucketing requires: (i) *grouping together crashes with the same root cause* (minimizing type I errors); (ii) *creating new groups for crashes with different root causes* (minimizing type II errors); and (iii) capturing accurate *bug context*. Capturing accurate bug context requires complete modeling and analysis of program behavior. However, existing modeling/analysis techniques (e.g., symbolic execution) do not scale [16]. Instead, practical crash bucketing relies on error-prone heuristics. Reducing these errors requires (a) *behavioral metrics* that correlate with bug context, and (b) *execution trace trimming* to minimize noise.

3.1.1 Behavioral Metrics

Bug context is the critical set of program behaviors accumulated and leading up to the crash site. Capturing bug context is key to accurate crash grouping. However, approximating program behavior is a three-way trade-off between sensitivity, accuracy, and scalability. On the upper end of the sensitivity spectrum, full path coverage (control and data flow) is the most precise: each test case exercises a unique path. This metric thus has a minimal type II error rate, achieving high accuracy in distinguishing different bugs. However, its precision also results in fine-grained grouping

---

[2] In this chapter, when two programs share a CVE, we count it as two CVEs

of crashes, and thus boasting a high type I error rate and a low accuracy in identifying similar root causes. Collecting precise path coverage is also infeasible in practice due to the large state space most programs have, and the scalability challenges that arise from this. On the other end of the spectrum, in singling out what program behavior led to a crash, Boolean function coverage is imprecise: it only captures the set of functions possibly involved in triggering a crash, and does so without conserving the order of code execution. Test cases resulting from the same bug or from different bugs are likely to display identical coverage, thus reducing type I error rates but raising those of type II. Nevertheless, Boolean function coverage requires minimal resources to measure and collect, improving scalability.

Research Context

According to Dhaliwal et al. [39], 80 % of the root causes of a bug are located on the call stack at the time of the crash. Several research projects have taken this notion and explored different stack hashing techniques [24, 27, 36, 81, 86, 136, 138]. Stack hashing is fast and easy to deploy, and is suitable for large-scale campaigns to batch-process a large number of test cases, but is prone to both type I and II errors and lacks high-quality classification.

Execution traces can also be used to classify test cases. This approach typically uses binary translation [100] or interposition [115]. Several taint-based approaches have also looked into classifying test cases as a form of crash analysis. For example, CrashFilter [73] automatically classifies a failing test case based on static taint analysis. RETracer[35] recovers early program state from a memory dump based on a reverse taint analysis, after which an analyst manually groups test cases. REPT [34] extends RETracer and achieves a more accurate classification result by reconstructing the data flow. Finally, POMP [167] classifies crashes from the perspective of the different contributions of data flow to program crashes.

These taint-based methods group crashes from the perspective of data flow. They are more precise than stack hashing. However, taint-based methods often do not scale to complex programs as they need to record a detailed program trace, which quickly exceeds several 100 GiBs for even simple programs, and require complex symbolic reasoning over paths that use tainted input data to make control-flow decisions. Taint-based methods can be effective if the bug only depends on data flow and not on control flow decisions based on tainted input data [29, 133, 162].

Challenges

Due to the complexity of control flow paths in programs, methods based on call stacks or crash sites cannot distinguish between different and similar bugs, resulting in a high mis-identification rate [83]. In practice, finding the balance between sensitivity, accuracy, and scalability requires identifying which behavioral metric most closely correlates to the root cause. We conclude from our results that minimized execution traces give the best insight into bug triggers, making them amenable for similarity matching.

Dubious crashes

Example 3.1

```c
1 char *buggy_concat(char *dst, const char *src) {
2     size_t dst_len = strlen(dst), src_len = strlen(src);
3     // Common bug: No room for null terminator
4     char *new_dst = (char *) malloc(dst_len + src_len);
5     if (!new_dst) return NULL;
6     strcpy(new_dst, dst);            // Crash site 1: when src_len == 0
7     strcpy(new_dst + dst_len, src);  // Crash site 2: when src_len > 0
8     return new_dst;
9 }
```

Listing 3.1 – Two crash sites observed for the same bug.

Listing 3.1 illustrates how a single bug can manifest as a crash in multiple locations, breaking location- and call-stack- based grouping tools. `new_dst` is allocated on line 4 without accounting for the C-string null terminator. If `src` is an empty string, the allocated space can fit `dst`, but `strcpy` appends the null terminator, causing a crash due to an out-of-bounds write at line 6. Otherwise, the space allocated for `src` in `new_dst` would accommodate for the overflow in the first copy, but a similar crash would be observed when copying `src` on line 7.

```c
1 char *buggy_strdup(const char *src) {
2     size_t len = strlen(src);
3     char *dst = (char *) malloc(len + 1);  // Bug 1: no check for NULL
4     for (size_t i = 0; i <= len + 1; i++)  // Bug 2: off-by-one OOB access
5       dst[i] = src[i];                     // Common crash site
6     return dst;
7 }
```

Listing 3.2 – Two bugs sharing the same crash site.

In contrast, listing 3.2 shows how different vulnerabilities can crash at the same location. In the first bug on line 3, the function fails to verify that the allocation succeeded, allowing for `dst` to be a `nullptr`. In the second bug on line 4, the loop bounds are off by one, copying one more byte past the null terminator on the heap. Both bugs manifest when a write access to `dst` is made at line 5. In other words, it is impossible to distinguish between the two bugs solely based on the crash location.

3.1.2 Test Case Reduction

Imperfections in behavioral metrics can be amplified by extraneous data points in the recorded execution trace. While reducing the sensitivity of the metric can improve its resilience against noisy measurements, this only addresses a symptom of the noise but not its source: entropy. In their study on fault localization, Christi et al. [31] showed that the accuracy of localization benefits greatly from reduced test cases. *Test case reduction* refers to continuously reducing the size of the crashing input, under the premise that the same crash is always triggered. In practice, streamlined test cases improve the software development process; e.g., in *vulnerability mining*, a minimized test case improves mutation efficiency and guides the process towards interesting behaviors; and in *crash analysis*, it eliminates extraneous bytes to simplify control flow leading up to the crash.

Research Context

The two main methods used in test case reduction are *delta debugging* [174] and *taint analysis* [97].

Delta debugging is a popular approach that automatically minimizes test cases by using two algorithms: *simplification* and *isolation* [108]. The former continuously shrinks the size of the original input file until it cannot find a smaller file that crashes the program, and the latter searches for a passing input, which will become a failing input again after satisfying additional constraints. The most popular test case simplification tool for fuzzing campaigns, `afl-tmin` [171], is based on this principle.

Taint analysis is a method that marks accessed registers and memory as tainted when the program crashes, and then tracks the source of the taint to mark all crash-related components in the input, thereby reducing the crashing test case to the relevant bytes. In general any forward or backward taint analysis can be used [32], but in practice, the length of the input or trace is often prohibitive. Long traces and complex inputs result in over-tainting or under-tainting along with difficulties of keeping control of the control-flow dependencies given the synthetic input. In practice, the application of taint analysis to crash grouping is limited.

Challenges

Aurora [19] is a root-cause identification method that leverages delta debugging to distinguish between critical and benign execution traces resulting from a crash. The process begins with a crash diversification phase where the faulty input is mutated to generate both crashing and non-crashing test cases. However, diversification carries the risk of introducing new bugs that are unrelated to the root cause under study. This is because Aurora follows an exploratory fuzzing process (i.e., increasing coverage) to generate new inputs.

Test case minimization may also introduce new bugs unrelated to the original bug. This is combated with a stricter fitness function: only a subset of the original trace must be executed, heavily limiting exploration. While this does not provide a guarantee against the introduction of new bugs, our evaluation shows that the error rate is negligible in practice. In comparison to AFL's crash mode [171]—which shares the fuzzer's fitness function—IGOR introduces 90 % fewer false positives.

The problem of minimizing crashes is also non-convex. Typically, the fitness function for test case reduction is the size of the input (in bytes). We instead propose minimizing the size of program's *execution trace*, thus reducing the execution complexity, rather than the input's size. Pruning the execution trace of a crashing test case can yield more concise inputs that exercise the same desired program behavior. A bug is triggered by a subset of all possible execution traces, and through a minimization function over the input space, a critical path to the root cause can be found and used to triage the crash. However, program behavior is complex, and a hill-climbing minimization process is likely to converge to a local minimum. Intuitively, traces that converge to different local minima may suggest different root causes, further complicating analysis. However, we found that combining this process with a clustering procedure led to minor variations in traces being overlooked in favor of the global trace overlap.

3.1.3 Our Approach: IGOR

While existing methods provide an initial step at reducing the large number of crashing test cases requiring triage, the type I and type II errors that these methods introduce result in uncertainty and may even increase developer effort and/or cause missed bugs. We address this imprecision by introducing a dual-phase approach for crash analysis. This approach builds on a key observation: each bug has a *core behavior* that must be executed to trigger the bug. A technique that extracts and matches this behavior can distil the large amount of crashes into a precise set of unique bugs. While analyzing large numbers of crashing test cases (together with their ground-truth), we observed that test cases for the same bug partially overlap in essential phases of their execution trace; *the bug trigger*. Our technique, IGOR, extracts an approximation of bug triggers and then leverages topological graph matching to group similar test cases into bug classes. This dual-phase approach for efficient and effective crash grouping through root-cause analysis thus combines and extends two important areas of research: *test case reduction*—minimizing and simplifying test cases—and *crash grouping*—determining if two inputs trigger the same or different bugs. Broadly speaking, IGOR leverages test case reduction to simplify the execution traces observed by the test cases, so that we can group crashes based on similarities in their coverage (i.e., we improve the latter by leveraging the former).

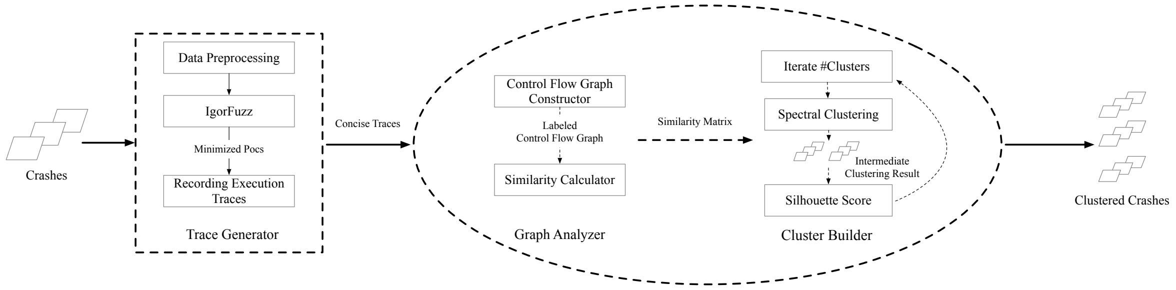


Figure 3.1 – IGOR overview.

3.2 IGOR Design

Figure 3.1 depicts the main components and workflow of IGOR. Starting with a set of crashing test cases, the *preprocessing* stage reduces unnecessary analysis costs by leveraging sampling and `afl-tmin`. Following this, the *trace generator* (IGORFUZZ) records and minimizes the execution traces of each preprocessed test case. The *graph analyzer* then constructs control-flow graphs from the minimized traces and extracts graph similarity metrics that describe each test case. Finally, the *cluster builder* classifies the test cases into separate groups, each identifying a unique root cause, and leverages a validation loop to find an optimal clustering configuration.

3.2.1 Data Preprocessing

IGOR's key objective is to distil crashing, proof-of-concept test cases (PoC) into unique bugs. To be practical, IGOR must scale with increasing numbers of PoCs. However, minimizing and grouping several hundred thousands PoCs is time-consuming. Therefore, we employ a two-stage preprocessing phase to reduce processing cost while maintaining accuracy. First, we sample the PoC corpus to reduce the number of analyzed test cases. Second, we leverage `afl-tmin` as an initial test-case size minimization tool, allowing IGORFUZZ to converge to a solution faster.

Sampling Although stack hashing is generally imprecise, it is rare that two *different* root causes overlap in the entire call stack. In other words, one bug may result in many diverse stack hashes but one stack hash generally only maps to one bug. In our dataset, only six (of 71) bug pairs in three programs contain shared stack hashes. We leverage this observation to reduce the PoC corpus by grouping PoCs based on their full-length call stack hashes. If there are many PoCs mapping to the same unique stack hash, we only process the 50 most diverse PoCs. This allows us to remove highly-similar PoCs that map to the same bug, lowering the cost of processing without impacting precision.

Minimization Minimized test cases benefit fuzzing in two ways: (i) the reduced size of the input leads to faster parsing and processing by the target, yielding higher fuzzing throughput; and (ii) by removing extraneous bytes from the input, the fuzzer's mutations are more likely to modify bytes critical to the behavior of the program. The case of minimum-coverage fuzzing is no different and can thus be improved by preprocessing test cases to remove extraneous bytes. We achieve this with the aid of `afl-tmin` [171], a lightweight test case reduction tool included with AFL. Although `afl-tmin` focuses exclusively on reducing the length of the input (i.e., this is the sole metric it optimized for), it indirectly shortens the length of the execution trace, further assisting IGORFUZZ in finding a minimal PoC. `Afl-tmin` also allows us to merge PoCs that reduce to identical byte sequences. Using `afl-tmin` in the second preprocessing stage increases IGORFUZZ's effectiveness in exploring shorter execution traces. As discussed in section 3.1.2, `afl-tmin` may introduce new bugs. However, uncovering a new bug during minimization with `afl-tmin` is rare: in our evaluation, only one out of the 5,531 PoCs triggered a different bug after minimization through `afl-tmin`. Our evaluation shows that the other nine PoCs (for the same bug) remain correct, allowing IGOR to correctly cluster the nine PoCs to that root cause. While we lost one PoC (out of 5,531) during minimization, enough PoCs remained to correctly identify all unique bugs. This effectively means that we did not introduce any false negatives in our evaluation through `afl-tmin`.

3.2.2 IGORFUZZ: Minimum-Coverage Fuzzing

Coverage-guided fuzzing typically incorporates feedback to *maximize* code coverage and to trigger crashes. The highly-stochastic nature of fuzzers means that they often find many diverse test cases that trigger the same bug. This results in extraneous execution traces that amplify the imprecision of the underlying metric and hampers clustering.

The intuition behind minimum-coverage fuzzing is that a bug only manifests when the faulty code is executed. Minimizing the code executed before reaching faulty code reduces the amount of state that the developer has to analyze during debugging. Given a mechanism to measure what code was executed when a bug is triggered, and a mechanism to sort the measurements across different samples, it is possible to find the minimal code trace required to trigger the bug, which we identify as the **shortest bug-triggering path** (with the absolute minimum code trace being the empty set). This observation accounts for the possibility that a bug can be triggered through different paths. Through fuzzing, we perform a search over the state space in the vicinity of the different bug paths, with the objective of finding the shortest (simplest) execution trace. Since fuzzing is a dynamic technique, it is inevitable that multiple suboptimal solutions will be found; however, our evaluation shows that the different solutions display features that are similar enough to be grouped under the same root cause.

Ideally, the single shortest path to a bug is its best identifier. An oracle that determines the shortest path would solve the crash grouping problem, as all PoCs for the same bug would reduce to the same path. Reducing an arbitrary PoC to the minimal path is challenging, because it requires determining the shortest path that still satisfies all of the bug context's constraints. In practice, this results in long execution traces that exceed the capabilities of today's solvers. For example, the shortest paths in our evaluation contained over 91,000 basic blocks (in *LibPNG*), while the shortest execution trace contained up to 22,563,000 basic blocks (in *Poppler*).

Existing test case reduction techniques use an objective function that favors a smaller input size. This is an artifact of bug reporting guidelines, which typically require a *minimum working example* as a PoC. However, reducing the input size does not always translate to reduced trace complexity. To highlight this effect, we conduct a study on `afl-tmin` using five targets and ten bugs. While test cases generally reduce in size, this does not necessarily guarantee a reduction in the length of the execution trace. For example, `afl-tmin` reduces *OpenSSL*'s `x509` input size by 41.10 %, while only pruning 4.83 % of CFG edges. Moreover, although `afl-tmin` reduces the input size of `pdfimages` by 82.45 % (on average), 15.18 % of the test cases execute more edges after reduction by `afl-tmin`. This demonstrates that a smaller input size does not guarantee a simpler execution trace. See table B.4 and section B.3 for our full results.

Similar to vulnerability mining, fuzzing provides an opportunity for an efficient and scalable process that finds simpler inputs which exercise similar behavior. To reduce the complexity of the execution trace, we propose a new type of fuzzer with subtly different goals than existing fuzzers. The fitness function in our fuzzer, IGORFUZZ, favors a *simpler execution trace*, by exercising fewer edges while still crashing the target at the same location.

The goal of reduction is to find simpler test cases that trigger the same bug. Multiple bugs can share the same crash site, and minimized test cases present the risk of triggering different bugs at the same location. However, in practice, the probability of *simplified* test cases triggering different bugs is insignificant. It is important to distinguish the use of crash sites between *grouping* and *simplifying* crashes. When grouping based on crash sites, inputs are diverse and presumably exercise different program behaviors. Due to this noise, misclassified bugs will be common. However, when simplifying a test case based on its crash site, we reduce the complexity of the execution trace while exercising similar program behavior. This minimizes the likelihood of the new derived test case triggering a different bug.

State-of-the-art fuzzers aim for maximal coverage to aid bug discovery. Existing work in *maximizing coverage* can be grouped into three areas: seed retention strategies [51, 133, 159, 165], seed selection strategies [22, 29, 41], and seed scheduling strategies [21, 22, 168]. We guide the design of IGORFUZZ based on insights from these three areas and modify a generic feedback-guided greybox fuzzer such as the one presented by Böhme et al. [22]. Algorithm 1 outlines our coverage-minimizing algorithm and fig. 3.2 gives an example of IGORFUZZ's effectiveness.

Algorithm 1 IGORFUZZ algorithm

```
procedure FuzzTest(Prog, Seeds)
    Queue ← Seeds
    while true do
        for input in Queue do
            if ¬isWorthFuzzing(input) then
                continue
            mutated ← Mutate(input)
            if ¬isInteresting(mutated) then
                continue
            mutated.score ← calcScore(Prog, mutated)
            addToQueue(mutated)
procedure isFavorite(input)
    return trimsDisjunctEdges(input)
procedure isWorthFuzzing(Prog, input)
    return isFavorite(input) or isLarger(input.score)
procedure isInteresting(Prog, input)
    return isCrashing(input) and
        (hasSmallerBitmapSize(input) or
        isEdgePruned(input) or
        hasSmallerHitCountSum(input))
procedure calcScore(Prog, input)
    score ← input.score
    if executesFaster(Prog, input) then
        score ← calcScoreFast(input)
    if hitsFewerEdges() then
        score ← calcScoreSimple(input)
    return score
```

Seed Retention Principles

We modify the `isInteresting` method so that it only retains crashing test cases. We also introduce two complementary rules for seed retention:

Rule 1 The seed does not exercise a common edge (i.e., at least one edge is no longer executed, although overall coverage may increase).

Rule 2 The seed exercises some edges with fewer hit counts (i.e., at least one edge is executed fewer times and no edge is executed more times).

Through a hill-climbing optimization process over the coverage space, IGORFUZZ incrementally approaches a minimal execution trace. To reduce the risk of converging at local minima and to promote diversification, IGORFUZZ retains seeds that meet any of the two rules. Note that it does not suffice to measure if the new input executes strictly fewer edges than the original input, as IGORFUZZ may get stuck at a local minimum from which it cannot easily escape. Rule 1 therefore prioritizes a seed that prunes common edges, but permits it to execute more edges if at least one

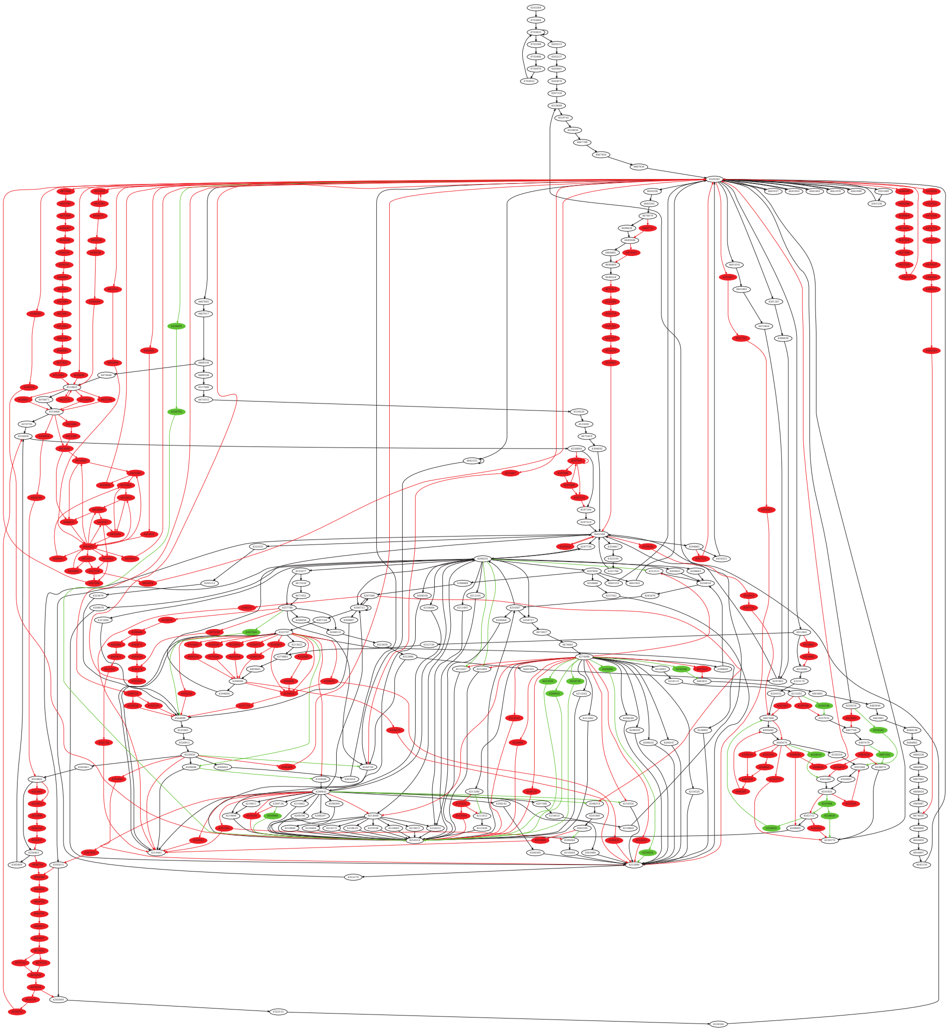


Figure 3.2 – Function-call graph of `tiffcp`. This figure shows the nodes and edges removed (highlighted in red) or added (highlighted in green) by IGORFUZZ.

previous edge is no longer executed. Rule 2 minimizes hit counts to simplify loops and recursive function calls. Together these rules drive coverage downwards, one removing edges (but potentially adding new ones), the other reducing edge counts. Note that at any point in time, multiple seeds are being scheduled for mutation depending on the seed selection principles below.

Seed Selection Principles

To achieve a higher efficiency at exploring the coverage space of a target, AFL [171] marks the seeds which exercise disjunct sets of edges as favorites. By finding a minimal set over the sets of edges executed by each seed (e.g., as performed by Karp [78]), AFL determines the seeds which are more likely to increase coverage in different directions of the code and more favourably mutates those seeds. Previous research on seed selection [66] also validates this strategy: the fewer and more distinct the seeds are, the better the fuzzer is able to maximize coverage.

In contrast to AFL's approach, IGORFUZZ attempts to minimize coverage by favoring seeds which *trim* disjunct sets of edges. New seeds that trim groups of edges are more likely to prune those edges from the execution trace upon further mutation. For that reason, we mark such seeds as favorites and assign them higher energy.

Seed Energy Scheduling

Seed energy determines how many times a seed will be fuzzed after it is selected. Higher seed energy results in more mutations and executions. By mutating seeds with a smaller number of recorded edges, IGORFUZZ can process inputs faster and is more likely to discover even simpler PoCs. We use a greedy heuristic to allocate more energy to seeds with a shorter execution length.

We also dynamically allocate energy to different seeds based on their coverage profile. We assign more energy to seeds with shorter execution paths. Not only do we assign more energy to simpler seeds, but we also ensure that the assigned energy is biased towards larger reductions in coverage.

Fuzzing Output Selection Criteria

To explore different code regions, fuzzing typically employs a meta-heuristic optimization over the target's coverage space. During the search process, fuzzing naturally encounters solutions that diverge from the original objective. In the context of IGORFUZZ, that objective is finding the simplest PoC that triggers the same initial bug. With the help of BITMAPSIZE—which AFL++ [49] uses to show how many unique edges were activated—we pick the simplest PoC by finding the PoC whose bitmap size is the smallest. Despite limiting search to the vicinity of the original execution trace, it is inevitable that other bugs are triggered. To filter out those erroneous solutions, we rely on crash sites as the penultimate selection criterion, and among the remaining seeds, we select the PoC with the smallest bitmap size. According to our evaluation, IGORFUZZ has a low probability of discovering crashes caused by a different bug (section 3.4.5).

3.2.3 Test Case Similarity Measurement

Existing approaches for measuring test case similarity leverage information available at the crash site (e.g., the call stack, instruction pointer, register contents). A fundamental limitation of these approaches is that a bug may cause the program to crash in many different locations, resulting in an over-approximation of the unique bug count. Backward slicing may alleviate this problem by tracing the flow of the crashing condition to its origin. Unfortunately, backward slicing does not apply to all types of vulnerabilities (e.g., in a use-after-free vulnerability, the location where data is reused can be independent from where it is freed). Moreover, scaling backward slicing to large and complex program traces across 100,000 basic blocks is an unsolved problem. Studying the limitations of current approaches led us to the following insight: all crashes that are due to the same bug *must* have a (partially) overlapping execution trace. Furthermore, methods for analyzing the topological structure of the execution trace can be used to extract a signal for crashes of the same bug.

The program execution trace is a *sampled sequence* of program addresses sorted by their execution order. To calculate their similarity, existing approaches leverage *Levenshtein Distance* [169] and *Longest Common Subsequences* [67]. Unfortunately, program loops increase the difference between execution traces, limiting the utility of these techniques. Even if these execution traces come from crashes of the same root cause, their sequence similarity is low. However, collapsing the execution trace onto a graph means that control-flow similarity is no longer affected by different loop iteration counts. Thus, our approach uses graph topology to calculate similarity between traces.

Control-flow Graphs (CFG)

Control-flow graphs describe the execution process of a program. A node in a CFG represents a program address, and edges connect two successive addresses. To construct the CFG, we sample the execution trace at a predefined granularity. On one end of the spectrum, fine-grain instruction-level traces introduce redundant nodes into the CFG, since edges between consecutive instructions are implicitly captured by the sequential nature of program execution. Additionally, recording instruction-level traces imposes scalability challenges, rendering the approach intractable. On the other end of the spectrum, function traces are easily collected, but they are coarse-grained and can overlook key behaviors in the program that capture the bug context. Basic block-level granularity provides a balance between precision and scalability, and we use it to construct the CFGs for similarity matching.

Execution traces contain noise that must be filtered. There are two types of noise: (i) code executed in external shared libraries (e.g., glibc); and (ii) superfluous code after the program's crashing point (e.g., ASan's crash handler). Since we focus on a specific target program, the execution trace of external code is unnecessary. Thus, we filter any addresses that are outside our target code. Sanitizers detect bugs early by introducing additional software guards. When a bug is detected, the sanitizer executes additional code to collect and report information. This information collection and reporting adds unnecessary noise and is filtered as well. We discuss how we filter these two types of noise (during the execution and after the bug trigger) in section 3.3.

Calculating Graph Similarity

After recording traces, we construct a CFG for calculating graph similarity. Efficient graph similarity estimation has been studied extensively [84], and the results show that *kernel methods* are most suitable for estimating graph structure similarity. At present, most kernel methods for graph similarity estimation include two types: *graph embedding* and *graph kernel*. The former leverages traditional vector-wise kernel algorithms based on dimensional reduction of input graphs, which leads to a loss of structural information. The latter directly performs kernel algorithms in a high-dimensional Hilbert space, so that the structural information of graphs remains intact.

After surveying different graph kernels (e.g., labelled graphs, weighted graphs, or directed graphs), we found that the *Weisfeiler-Lehman Subtree Kernel* algorithm demonstrated the highest ability to differentiate test cases of varying root causes while assigning high similarity measurements to test cases of the same root cause. Thus, we adopt the Weisfeiler-Lehman Subtree Kernel to estimate the similarity between CFGs. The Weisfeiler-Lehman Subtree Kernel algorithm requires the nodes of a graph to be labelled. We use the basic block addresses (from the execution trace) as node labels.

3.2.4 Bug Clustering

Fuzzers are highly non-deterministic in how they discover bugs and paths through the target program. The fuzzer may discover one or more crashes for a bug, but it is not known *a priori* how many crashes it will find. Consequently, IGOR cannot know how many bugs are discovered during a fuzzing campaign. For example, a given set of 100 crashes may map to 100 bugs with one crash each, one bug with 100 crashes, or anything in between. The key challenges for clustering are to (a) *discover the exact number of bugs*, and (b) *assign crashes to the correct bug*. As stated above, the number of expected clusters is not known and must be inferred during clustering. Our approach determines the number of clusters by running the clustering process multiple times, refining the assumed number of bugs based on a heuristic (see section 3.2.4).

Data Characteristics

The Weisfeiler-Lehman Subtree Kernel algorithm produces a similarity matrix $\mathbf{M_s}$; let $\mathbf{M_d} = \mathbf{1} - \mathbf{M_s}$, where $\mathbf{M_d}$ is the corresponding distance matrix. We build our clustering algorithm on the two matrices.

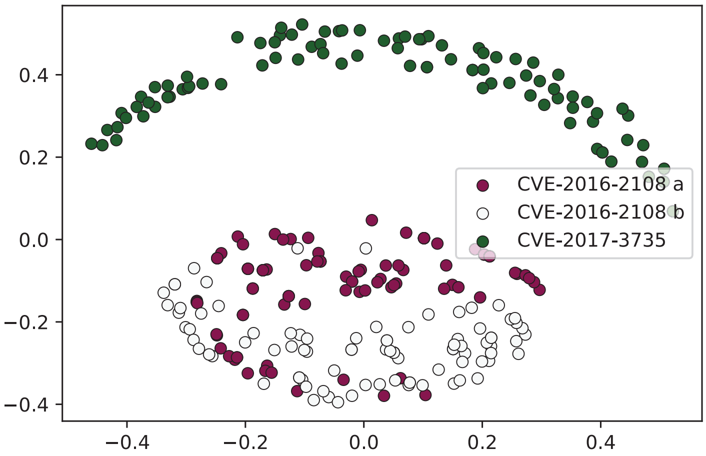


Figure 3.3 – Distribution of samples for `x509`. The suffixes ‘a’ and ‘b’ indicate two call stacks for one CVE.

The matrices $\mathbf{M_s}$ and $\mathbf{M_d}$ are not sufficient to determine the distribution of samples. Multidimensional Scaling (MDS) [23] is an algorithm for dimensionality reduction. It projects high dimensional data to a lower dimensional space by finding abstract Cartesian coordinates in the lower dimensional space that keeps the distance between samples almost unchanged. We leverage MDS to visualize the distribution of samples in a two-dimensional plane.

We analyze distributions based on these visualizations. For example, fig. 3.3 shows how samples of `x509` are distributed across the Cartesian plane. The shapes of clusters are not spherical, indicating that k-means-like methods are not suitable, and we therefore rely on alternative methods to recognize clusters of arbitrary shapes.

IGOR leverages the *silhouette score* [134] to describe the quality of cluster structures. By simplifying test cases, the samples tend to have better cluster structure, see fig. 3.7.

Clustering Algorithm

DBSCAN [46] is a density-based clustering algorithm that uses a similarity/distance matrix. The DBSCAN algorithm takes two parameters, ϵ and $MinPts$, that define the density threshold. The primary challenge is how to determine these parameters. Although heuristics exist [137, 139], changes in the density distribution of samples result in a failure of the clustering process. Alternative algorithms like OPTICS [6] and HDBSCAN* [25] perform hierarchical analysis to obtain better results. However, they still require predefined density descriptive parameters, which are must be tuned when analyzing new programs (because the samples' density distribution varies between programs).

We found that the *Spectral Clustering* [101] algorithm addresses the parameter-tuning challenges brought on by the high-dimensional information stored in the Weisfeiler-Lehman Subtree Kernel similarity matrices. Spectral Clustering takes only one parameter: the number of clusters. Given the number of clusters and a similarity matrix, Spectral Clustering builds a graph Laplacian on the basis of the similarity matrix. Eigenvectors of the graph Laplacian are calculated to realize dimensionality reduction. Then, the algorithm automatically groups samples into the designated number of clusters in a lower dimensional space.

Since the real number of clusters is unknown, we need a metric for evaluating the clustering result. For this purpose, we again use the silhouette score. Liu et al. [96] describe different clustering validation measures. In their study, they indicate that the optimal cluster number can be determined by maximizing the value of the silhouette score.

As the silhouette score is undefined when there is only one cluster, we assume that there are at least two clusters among the dataset. Then we enumerate the number of clusters to run the clustering process, and calculate the silhouette score of the result. Afterwards, we select the clustering result with the highest silhouette score. However, this approach may undercount the number of clusters. We therefore develop a heuristic to decide if we need to repeat clustering based on the number of full-length crash call stacks: for less than 20 call stacks, running the clustering process once suffices, while for more than 20 call stacks, we cluster twice. Tables 3.1 and B.5 show our experimental results based on this heuristic.

Misclustering for single-bugs

The presence of tightly interspersed sub-clusters indicates that there is likely only one cluster. The minimum number of detectable clusters is two. If there is only one, then the number of clusters is raised artificially during enumeration to find a higher silhouette score. It is rare in practice for a fuzzing campaign to find hundreds of test cases for only one vulnerability. After thousands of CPU hours of fuzzing, we have never observed this case. In this case, using current crash grouping methods—e.g., stack bucketing—is sufficient.

To mitigate against this rare situation, IGOR compares its cluster result with a call-stack based approach (e.g., `afl-collect`). If IGOR reports more clusters than `afl-collect`, we indicate to the analyst that they should refer to the `afl-collect` results.

3.3 Implementation

Our prototype of IGOR demonstrates the practical feasibility of our approach. We briefly explain important implementation details in the following sections. We implement IGORFUZZ on top of AFL++, record execution traces using Intel Pin, and develop several Python scripts that orchestrate `gdb` to select and filter execution traces. We construct CFGs from traces using NetworkX [63], visualize them with Graphviz [45], and calculate graph similarity using GraKeL [147]. The clustering phase leverages `scikit-learn` [118]. Our implementation consists of approximately 1,000 lines of C++ code and 2,500 lines of Python code, along with several small scripts.

Recording Execution Traces and Noise Filtering We developed `smart-tracer`, a trace recorder based on Intel Pin that records execution traces at function call-, basic block-, and instruction-level granularity. During post-processing, we filter out function calls (a) into auxiliary code (e.g., calls into libc), and (b) that occur after the crash is triggered (e.g., sanitizer information collection).

CFG Similarity Metric The filtered traces are first used to construct the CFG. Then, we use this CFG to calculate graph similarity using the Weisfeiler-Lehman Subtree Kernel algorithm.

Clustering The clustering procedure runs up to fifteen times by default, enumerating the number of clusters from 2 to 16, and (re)calculating the silhouette score each time. This process outputs the clustering result with the highest silhouette score. Finally, a post-clustering scatter diagram is created to help analysts visually assess the quality of the clustering result.

3.4 Experimental Evaluation

We evaluate IGOR on 12 servers running Ubuntu 18.04 LTS, each with 200 GiB of RAM and an Intel(R) Xeon(R) Gold 6254 CPU @ 3.10GHz with 40 cores. We source our benchmark dataset from 58.7 CPU-years of fuzzing campaigns. The IGOR-specific evaluation took 6,143.7 CPU-hours and 30 human-days across all presented experiments. The IGORFUZZ evaluation took 5,531CPU-hours. Importantly, this is not necessarily indicative of the actual time required for minimization: we run IGORFUZZ one hour for each sampled PoC (5,531 CPU-hours in total) to show the changes over time. In practice, we would let IGORFUZZ run for 15 min per PoC (we demonstrate this in section 3.4.4); thus minimizing all 5,531 PoCs requires 1,382.75 CPU-hours. Compared to the total cost of fuzzing, minimization is negligible and consumes less than 0.3 % of the length of a fuzzing campaign.

3.4.1 Benchmarks

We evaluate IGOR on the Magma benchmark [64] and the MoonLight dataset [65, 66]. These benchmarks contain 52 CVEs that belong to 14 target programs, containing more than 254,000 crashing PoCs that map to 39 unique bugs. We exclude four programs containing only a single CVE: `pdftotext` (*Poppler*), `exif` (*PHP*), `client` (*OpenSSL*), and `libxml2_xml_read_memory_fuzzer` (*LibXML*). As discussed in section 3.2.4, IGOR falls back to `afl-collect` in this case (we still report these results in section B.2). Additionally, we exclude (a) two CVEs that result in prohibitively large trace files (over 10 GiB per PoC), and (b) six CVEs that are duplicates, semantically equivalent, or partial fixes to one of the 39 unique bugs (see section 3.5.3 for details). We augment these benchmarks with precise ground-truth data that verifies the root cause for each PoC.

Following the methodology outlined by Klees et al. [83], we map crashes to bugs through the patches that fix the crash (when applied). Importantly, both the Magma and the MoonLight datasets contain targets with multiple bugs. Unfortunately, multiple bugs (in a single target) may interfere with each other (e.g., a bug may mask/enable other bugs that would not manifest in a normal program execution). To minimize the risk of interference, we relabel our initial PoC dataset according to the patches that disable the crash. We show this process in fig. 3.4.

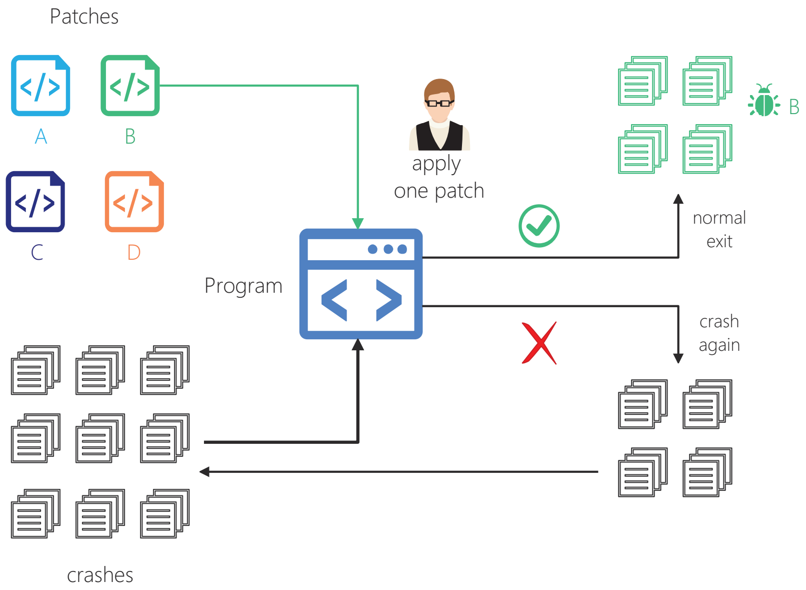


Figure 3.4 – Tagging PoCs by applying patches.

Starting with an unpatched version of the target, we apply patches one at a time, each time processing the entire PoC corpus to flag changes in crash status. If a change is detected, that patch is considered a ground-truth label for the PoC, and is used for measuring the performance of our clustering method. We manually verified the completeness of patches in fixing the root cause of a crash. For each crash in our benchmark, we label root cause and call-stack hash. Our ground-truth data allows us to determine whether crashes are grouped correctly (i.e, if they correspond to specific CVEs).

3.4.2 Summary of Results

Table 3.1 – Evaluation results. B represents the number of unique bugs in the program, N the number of samples in one experiment, C the number of unique crash sites, S the number of unique full-length call stacks, and K the number of clusters Igor generates. The *Top Frame*, *BFF-5*, *honggfuzz*, and `afl-collect` columns show crash grouping scores obtained by comparing call stacks of length 1, 5, 7, and full-length. *P*, *IP* and *F* are abbreviations of *Purity*, *Inverse Purity*, and *F-measure*, respectively. Finally, *cut-off* represents the running time we limit IgorFuzz's minimization for a single PoC. The best performing entries are highlighted.

Program	B	N	C	S	K	**Top Frame** (%)			**BFF-5** (%)			**honggfuzz** (%)			**afl-collect** (%)			**cut-off (mins)**	**Igor** (%)		
						P	IP	F	P	IP	F	P	IP	F	P	IP	F		P	IP	F
pdfimages	3	410	12	23	3	100	48	63	100	47	62	100	47	62	100	47	62	15	100	100	100
					3													30	99	99	99
pdftoppm	3	161	3	5	2	100	100	100	100	95	98	100	95	98	100	95	98	15	69	100	80
					2													30	70	100	80
tiffcp	5	991	3	20	6	85	89	80	88	76	74	88	67	68	88	67	68	15	100	98	99
					6													30	100	99	99
tiff2pdf	3	385	2	8	3	92	91	89	99	89	93	99	89	94	99	89	94	15	98	98	98
					3													30	98	98	98
x509	2	150	2	3	2	100	100	100	100	100	100	100	75	83	100	75	83	15	100	100	100
					2													30	100	100	100
png_read	2	150	2	3	2	100	72	81	100	72	81	100	72	81	100	72	81	15	100	100	100
					2													30	100	100	100
xmllint	8	1581	14	901	14	78	59	64	78	55	62	78	55	62	79	3	4	15	96	67	77
					18													30	97	66	77
char2svg	5	1087	20	67	8	100	66	72	100	14	20	100	14	20	100	14	20	15	100	67	79
					8													30	100	67	79
sox (MP3)	4	260	6	8	4	100	63	74	100	58	71	100	58	71	100	58	71	15	100	100	100
					4													30	100	100	100
sox (WAV)	4	356	6	8	4	100	67	77	100	66	76	100	66	76	100	66	76	15	100	100	100
					4													30	100	100	100

Table 3.1 shows the results of Igor's clustering evaluation with different IgorFuzz cut-off times. *Cut-off* refers to the time we allow IgorFuzz to minimize a single PoC. Here we only report 15 min and 30 min cut-off times, but we provide results for the other cut-off times in table B.5 in section B.4.

For Igor, we report clustering results for basic-block-level traces. We also evaluate Igor's clustering results at function-call (cheap and least precise) and instruction level (expensive and most precise) granularity. While the results are similar across the three tracing granularity, we see a correlation between better results and bugs with a larger number of crash sites. Specifically, our results indicate that IGOR should switch to instruction-level tracing when the number of crash sites exceeds ~ 20 (e.g., for `xmllint` and `char2svg`).

While grouping crashes based on stack hashes is common practice, different tools use different stack depths. For example, `afl- collect` hashes all stack frames, honggfuzz leverages the last seven stack frames, and CERT BFF [69] uses the last five frames. We compare IGOR against these three configurations and a baseline of a single stack frame (called "Top Frame" in table 3.1). To measure classification quality, we calculate precision, recall, purity [5, 151], inverse purity [5] and F-measure [88, 151]. These metrics are all standard approaches from the machine learning community.

Our results show that—compared to Top Frame, BFF-5, honggfuzz, and `afl-collect`—IGOR achieves the highest F-measure in 90 % of our experiments (and is 20 % off in the remaining 10 %) and achieves higher purity and inverse purity scores in most cases. From a quick glance at table 3.1, it may appear that crash sites are the best grouping method, but crash sites are often not unique because different bugs crash at the same program location (see the unique bug addresses in the crash address column in section B.2).

We also assess the loss of precision caused by our bug count inference process (which may be inaccurate in itself). We rerun the clustering process with ground-truth bug count, and compare these counts to the results of clustering that relies on an inferred bug count. Table B.5 (in section B.4) shows that IGOR's results are as accurate as the ground-truth bug counts.

3.4.3 Test Case Reduction

We evaluate IGORFUZZ (over all 39 bugs) to demonstrate its ability to remove redundant paths traversed by each PoC. For each bug we use `afl-cmin` to select representative PoCs from our corpus and run IGORFUZZ for 15 min. Per section 3.4.4, we found that 15 min is sufficient for each minimization. Figure 3.5 shows the *mean* edge count (i.e., the fuzzer's bitmap size) executed by the newly-minimized PoC, while fig. B.3 shows how IGORFUZZ reduces traces across three dimensions (basic block counts, edge counts, and trace length). Due to space constraints, we limit our results to these figures.

Figure 3.5 shows that the bitmap size monotonically decreases over time. We found that the length of the reduction process is proportional to the length of the program trace executed by the PoC. Larger bitmaps provide more opportunities for shrinkage, as evident from the topmost plots in fig. 3.5. The larger the bitmap size of the initial PoC, the slower the compression process.

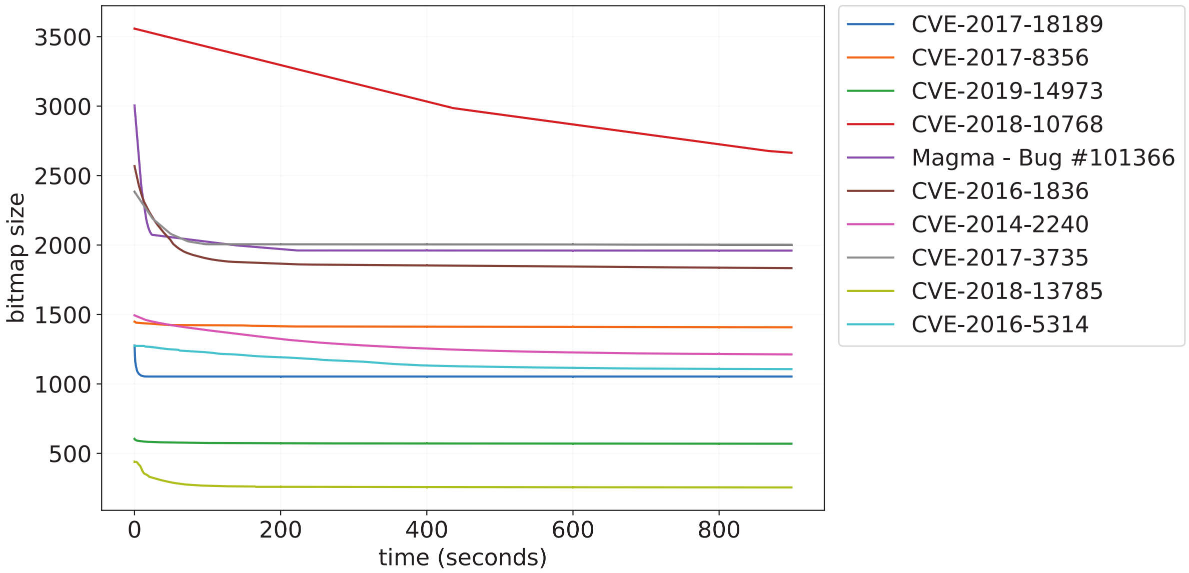


Figure 3.5 – Bitmap size (number of edges) trend over time.

The mean values in fig. 3.5 ignore outliers. Figure 3.6 highlights how IGOR reduces outliers for CVE-2016-5314. Figure 3.6 shows that the interquartile range drops significantly after IGORFUZZ's trace reduction. Furthermore, the outliers' bitmap size gradually converges to the mean, and the number of outliers continues to decrease over time (until they disappear completely).

3.4.4 Graph Similarity After Minimization

We cluster and compare the original and reduced PoCs to assess whether IGORFUZZ improves similarity analysis. This involves recording execution traces of the target program and calculating similarity based on the CFGs constructed from these traces.

Our results (across our ten target programs) demonstrate that coverage-reduction fuzzing makes PoCs more distinguishable. For example, fig. 3.7 shows the distribution of CFGs before and after performing coverage-reduction fuzzing on `tiffcp`. The cluster contains four vulnerabilities. Per section 3.4.4, because the bug-irrelevant paths have not been pruned, the test cases of different vulnerabilities are not distinguishable; they are combined with each other, resulting in nine clusters and a surplus of five misclassified bugs.

Clustering precision greatly improves after using IGORFUZZ to shrink PoCs. Per section 3.4.4, clustering shrunken PoCs leads to an accurate grouping of the 842 test cases into four clusters (based on their root causes). Table 3.2 shows the change in silhouette score before and after reduction.

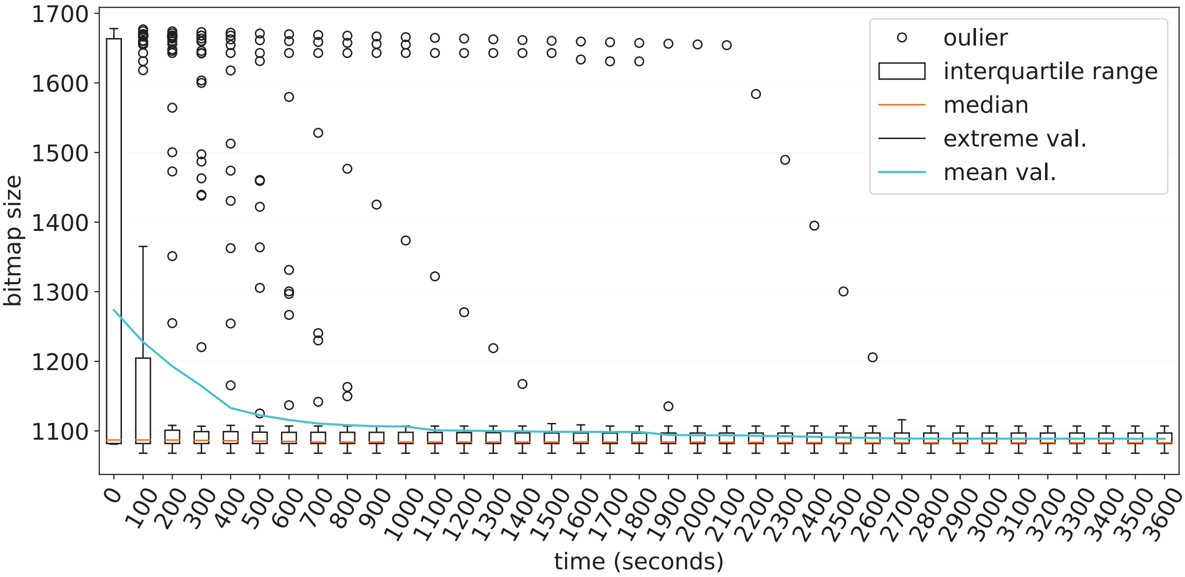


Figure 3.6 – Box plot of bitmap size for CVE-2016-5314.

Data distribution affects the results of the cluster [139]. Unfortunately, this distribution of data cannot be predicted in many real-world scenarios. We study IGOR's ability to process different data distributions, and our results show IGOR achieves ideal results under different data distributions (see section B.1).

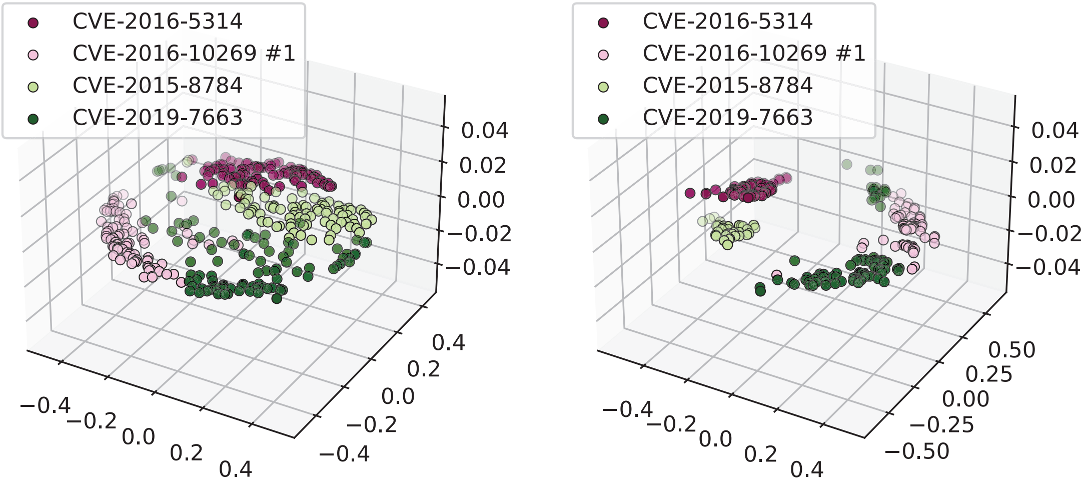
Suitable Cut-off Time

The time/cost tradeoff is an important factor for determining the practicality of IGOR. Given more time, IGORFUZZ can generate more concise PoCs, and our results show that pruning more bug-irrelevant paths leads to more accurate clustering results. Our results also show that each time IGOR's running time is doubled, we achieve a higher (mean) silhouette score (indicating more bug-irrelevant paths are pruned). But improving silhouette scores results in diminishing returns (considering performance of the second stage). After reaching a threshold, the cluster structure is highly identifiable and clustering results no longer improve.

According to our silhouette scores (table 3.2), IGORFUZZ increases the average silhouette score by 14.4 % during the first 15 min (the 0 min column corresponds to the time before any reduction has occurred). The next two intervals increase the average silhouette score by only 1.8 % and 0.5 %, respectively. This yields minimal benefits for clustering. We use the minimized PoCs generated by these three cut-off times for clustering (see tables 3.1 and B.5). These results show that although longer times will give us more precise results, it is sufficient to run IGORFUZZ with 15 min. Therefore, we suggest 15 min as the default cut-off time.

Table 3.2 – Silhouette scores.

Program	**0 min** (%)	**15 min** (%)	**30 min** (%)	**60 min** (%)
`pdfimages`	52.9	70.2	70.0	69.4
`pdftoppm`	59.0	80.2	80.0	79.0
`tiffcp`	33.2	70.2	73.9	77.4
`tiff2pdf`	66.0	70.6	70.7	71.2
`x509`	76.9	94.1	94.3	94.6
`png_read`	68.5	96.6	96.6	97.7
`xmllint`	38.3	49.1	55.3	54.7
`char2svg`	44.5	53.7	53.8	54.0
`sox` (MP3)	78.1	74.6	81.1	81.3
`sox` (WAV)	83.1	85.0	86.6	88.0
Average	60.0	74.4	76.2	76.7



(a) Original distribution.

(b) Distribution after reduction.

Figure 3.7 – Data distribution before and after reduction.

3.4.5 Verifying Minimization Results

Here we answer a key minimization question: does a minimized PoC trigger the same bug as the original PoC? We label a result a *false positive* if the minimization process discovers a new bug with a different root cause *and* this root cause replaces the original test case. We verify minimization results via the process described in section 3.4.1 (i.e., by successively applying patches).

We designed two experiments from two perspectives. First, we verify whether a minimized PoC introduces an error. We recorded *zero* false positives out of our 5,535 minimized PoCs. Secondly, we checked whether errors (i.e., PoCs triggering a new bug) were generated during fuzzing. If there are many error seeds in the queue, IGORFUZZ may be misled by these error seeds, and generate more error seeds during the fuzzing process. The number of queues containing error seeds (namely, "error queues") is shown in fig. 3.8. This figure shows the number of error queues among all the queues of their corresponding programs. Only 2.10 % of all 5,531 queues contain errors. This shows that IGORFUZZ has a low probability of introducing errors during the reduction process. Further, we also studied the proportion of error seeds in the error queues and found, with the exception of `xmllint`, this proportion is less than 10 % (see section B.4). This means IGORFUZZ will not waste time on error seeds. Although errors appeared during minimization in these programs, it does not affect the correctness of IGORFUZZ's output (due to our seed selection principles, detailed in section 3.2.2). According to our evaluation results, all errors were filtered out successfully.

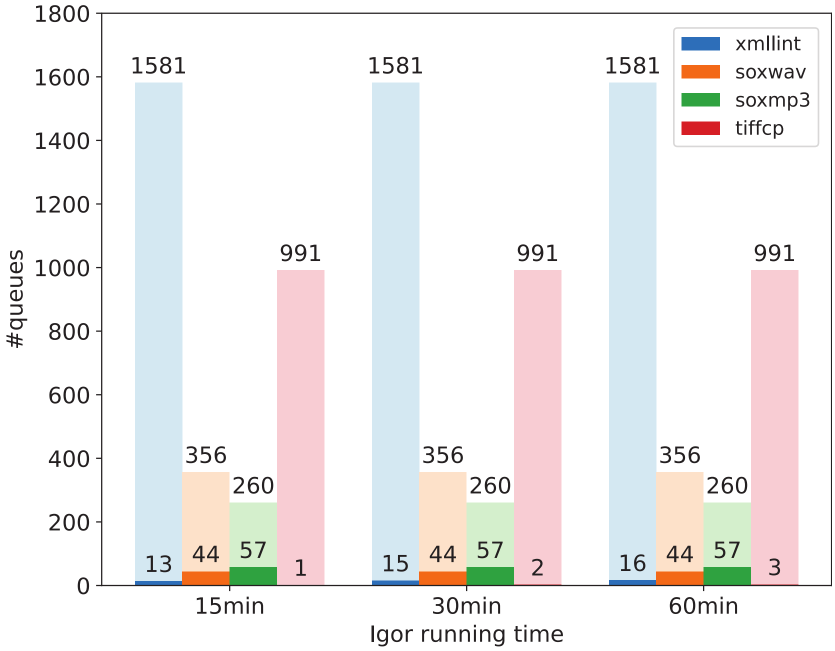


Figure 3.8 – Queue error rate.

3.5 Case Studies

We now present three case studies that—while challenging for classic crash grouping techniques—demonstrate the effectiveness of IGOR. These case studies include: (i) assessing the accuracy of CVEs; (ii) highlighting rare crashes; and (iii) detecting semantically equivalent bugs.

3.5.1 Assessing CVE Accuracy

Software maintainers manually analyze reported PoCs, assigning CVEs according to their root cause. However, a recent study [95] shows that even software developers misjudge root causes when faced with a large number of PoCs, resulting in imprecise CVEs where either different vulnerabilities are assigned the same CVE or different CVEs are assigned to the same vulnerability. IGOR's benchmark includes both cases.

For CVE-2016-10269_1 and CVE-2016-10269_2 (`tiffcp`), we sampled 400 PoCs and grouped them according to the root cause. IGOR clusters these PoCs into two distinct groups. After verification, we discovered that there are indeed two different vulnerabilities. As the functions executed before the crash are similar, the developer mistakenly classified these two vulnerabilities into one category.

For CVE-2019-8354 and CVE-2019-8366 (`SoX`), IGOR clusters 57 PoCs into a single group. These two CVEs have different crash addresses and call stacks, but the developer confirmed that these two vulnerabilities are indeed caused by the same root cause, the assigned CVEs are duplicates.[3]

For CVE-2015-9290 and CVE-2015-9381 (`char2svg`), we found that their patches both partially fix the same vulnerability. IGOR correctly clusters the corresponding 41 PoCs into a single group.

These three scenarios demonstrate how IGOR successfully highlighted the inaccuracy of assigned CVEs (inaccurate CVEs are labelled with superscripts in section B.2).

3.5.2 High-value Needles in Test Case Haystacks

Fuzzers cannot guarantee a similar number of PoCs for each bug. For bugs that are harder to find (e.g., due to deeper code depth, or harder trigger conditions), the number of PoCs generated by the fuzzer is often small. Such rarely-triggered bugs may be lost because crash grouping may wrongly merge them with other bugs.

IGOR ensures that such rare PoCs are correctly analyzed. By first bucketing PoCs based on the crash address, IGOR detects crash locations with few PoCs and uses AFL's crash mode to amplify them. In our current implementation, IGOR amplifies the number of PoCs at crash locations with fewer than ten PoCs to at least 50.

As an example, we study CVE-2016-5314 (`tiffcp`). Consider our analysis for trial 3 in table B.1. If we instead limit the number of PoCs for CVE-2016-5314 to three, then this CVE is subsumed by another CVE. When we amplify the three test cases to 50, the CVE is successfully separated.

---

[3] https://gitlab.alpinelinux.org/alpine/aports/-/issues/10523

3.5.3 Detecting Semantically Equivalent Bugs

Our benchmark distinguishes vulnerabilities based on unique patches. If the patch used to eliminate the failing test case is different, it is considered to be a different vulnerability. Based on this rule, we found that IGOR's result misclassifies PoCs for `char2svg`. The `char2svg` PoCs come from eight CVEs with a different patch for each CVE, indicating that these are eight different bugs (see section B.2).

However, we discovered that IGOR groups the PoCs from CVE-2014-9663, CVE-2014-9669, and CVE-2015-9383 into a single group. We reviewed the corresponding code and patches in `char2svg`, and found that these three CVEs share the same semantic root cause. Specifically, we found that all three CVEs check whether `FT_INVALID` is too short to avoid heap-buffer-overflows and all three patches are in the same source file. There are six similarly named functions that validate the input (e.g., `tt_cmap4_validate`, `tt_cmap8_validate`, `tt_cmap10_validate`). All six functions are buggy, and share the same root cause. The developer first committed the patch for CVE-2014-9669, in which five of the six vulnerable functions were fixed. The forgotten vulnerable function, `tt_cmap4_validate`, was fixed in the patch for CVE-2014-9663 ten days later. However, the patch for CVE-2014-9669 is an incomplete fix, because it misses one aspect of `FT_INVALID`. This went unnoticed until CVE-2015-9383 was reported, and the patch for CVE-2015-9383 finally correctly fixes this bug.

We argue that these three CVEs fix the same single bug from a semantic point of view: the same bug condition is shared among all three CVEs, and can be described as "as long as `FT_INVALID` is too short, it will cause the same variable be over-flowed". IGOR noticed that the control flow of the three CVE are indeed similar when they trigger the bug and, thus, IGOR grouped them into a single bucket.

3.6 Limitations

Using silhouette score to determine the number of clusters is not guaranteed to distinguish all clusters. Therefore, after the initial clustering is completed, it is necessary to manually review the clustering results. If there are still gaps between samples within the same cluster, it indicates that a second clustering process is required to obtain more reasonable clusters. Additionally, when there is only one bug, *intra*-class differences are regarded as *inter*-class differences. The clustering algorithm then may divide the input samples into a large number of clusters. IGOR will invoke `afl-collect` to group the crashes when it finds the group number given by itself is larger than that of `afl-collect`. In this case, IGOR falls back to the precision of `afl-collect`.

IGORFUZZ requires more computing resources than simple stack hashing. We argue that the increased precision of IGOR is worth the additional computation cost (less than 0.3 % in our experiments) due to the large amount of saved developer cost.

3.7 Related Work

Crash bucketing builds on a diverse background in fuzzing, test case reduction, crash grouping, crash de-duplication, and fault localization. We highlight how IGOR compares against these areas.

3.7.1 Test Case Reduction

`Afl-tmin` [171], a popular *test case minimizer*, removes as much data as possible from a seed while keeping the target in a crashing state. While fast and easy to use, it has limitations: although it can significantly reduce the input size, it cannot effectively reduce the number of edges that the program executes before crashing. Compared to `afl-tmin`, our method emphasizes the reduction of the execution trace instead of input size, which means the vulnerability trigger becomes more direct after the reduction. MacIver et al. [103] presented internal reduction by manipulating the behavior of the generator that produced them. Its input and purpose is different from ours, as it tries to shrink redundant operation of input and keep the same behavior, while we want to execute fewer edges before the program crashes.

3.7.2 Crash Grouping and De-duplication

Crash grouping reduces the crash analysis cost by leveraging specific metrics to measure the affinity between test cases, grouping similar test cases in one group.

Chen et al. [30] calculate edit distances and coverage profiles between test cases to group them. Both Tonder et al. [154] and Pham et al. [126] utilize symbolic execution technique to collect crash constraints to assist grouping. Molnar et al. [110] introduces *fuzzy stack hashing* by collecting multiple crash metrics and then hashing these information together as a grouping metric. Holmes et al. [68] proposes an alternative to crash grouping: if two failing test cases can be fixed by applying the same mutant, those two tests are likely related to the same root cause.

CrashLocator [166] discovers faulty functions that do not reside in the crash stack by expanding the given crash stack based on a function call graph. Although this method also takes advantage of static call graphs, it only uses them to recover traces, while our method uses dynamic call graphs to measure the similarity of PoCs.

ReBucket [36] proposed a method for clustering duplicate crash reports based on call stack similarity, the problem with this method is that it only relies on the local information at the time of the crash, when categorizing vulnerabilities with more number of call stacks at the time of the crash, it is easy to divide the PoCs that have same root cause into multiple different groups. Besides, the inputs of ReBucket are crash reports, but our inputs are PoCs, so the application scenarios of the two systems are different. SPIRiT [156] combines various methodologies to compute a specialized test case distance measurement, and drives a customized hierarchical test suite clustering algorithm that groups similar test cases together.

The primary difference between our method and the existing methods for crash grouping is that IGOR evaluates the similarity of executions of a program from a global perspective by utilizing the simplified execution trace, while the aforementioned ones mainly focus on local features of programs.

3.7.3 Fault Localization

The purpose of fault localization is to determine the root cause of bugs through crashes or core dump files. Statistical methods and Delta debugging are the main two technical solutions. AURORA [19] is a representative work of fault location based on statistical methods. Similar to ours, it uses crash mode to increase the number of test cases. The difference is that AURORA collects both crash and non-crash test cases and locates root cause by comparing them. The risk of this method is that crash mode does not guarantee that the generated test cases are still caused by the same vulnerability. When the crash sample is caused by other vulnerabilities, this method will cause false positive. Xu et al. [167] propose a method for locating the root cause through data flow analysis, the key of this method is to recover the data flow from core dumps automatically to provide richer debugging information and reduce the difficulty of manual analysis. Zamfir et al. [173] uses execution traces to help developers locate the root cause. CrashLocator [166] locates faults based on call stack. REPT [34] uses reverse debug to locate fault. Kim [81] uses multiple crashes to diagnosis root cause from the perspective of crash graphs.

3.8 IGOR Summary

Fuzzing has become ubiquitous and is today's key driver for bug discovery. Unfortunately, the number of reported crashes outpaces developers' ability to triage and fix bugs. Nonetheless, the number of crashes is generally an over-approximation of the total number of bugs, and crash grouping can drastically reduce the burden on developers. Existing approaches are light-weight but prone to misclassification. We argue for trading little computational time to effectively reduce the number of crashes by two orders of magnitude, close to the number of real bugs, lightening developer workload.

Our approach, IGOR, prunes bug-irrelevant paths and calculates the shortest bug-triggering path using a novel coverage reduction fuzzing approach. Crashes are then grouped based on the topological similarities between the CFGs of minimized execution traces.

Our evaluation demonstrates that IGOR outperforms existing approaches, resulting in the most precise bug clusters for 90 % of our evaluated programs. Based on a ground-truth comparison, we show that IGOR groups crashes of various root causes and vulnerability types precisely.

Chapter 4

Tango: Extracting Higher-Order Feedback through State Inference

> The real voyage of discovery consists not in seeking new landscapes, but in having new eyes.
>
> *Marcel Proust*

This chapter addresses the unique challenges of fuzzing stateful systems, where traditional coverage-guided techniques often fall short, since behavior depends on current and past interactions. We introduce Tango, an extensible framework for state-aware fuzzing that treats state as a first-class citizen of the fuzzing process. By leveraging a novel cross-validation technique called *state inference*, we aim to uncover hidden path dependencies that better guide the exploration of complex, stateful systems.

Hypothesis 3

The state of a system influences its behavior. Stateless feedback metrics are thus insufficient for the effective fuzzing of stateful systems, and state can be approximated by observing patterns in the system's behavior.

The contents of this chapter are adapted from a paper under submission to IEEE EuroSP'24.

Fuzzing is largely an exploratory process. Coverage-guided fuzzing excels at finding bugs as long as the feedback on covered code is strongly tied to the explored functionality in the target. While this intuition holds true for simple programs that accept a single input throughout their life cycle, systems of interest often do not exist in isolation, whereby the effects of consuming an input persist beyond the lifetime of that input. The behavior of such *stateful* systems is not solely described by the amount of covered code.

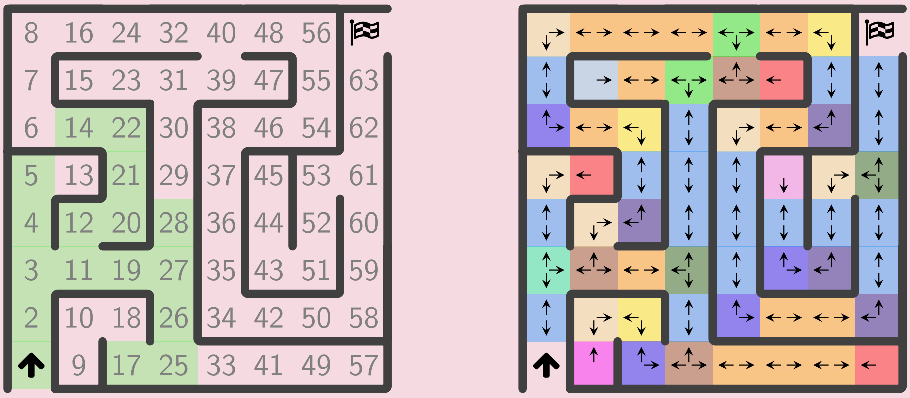
State

Definition 4.1 *State* is the sum and summary of a system's past operations and interactions at runtime, represented through its memory contents, and driving its future behavior. *Stateful fuzzing* is thus the branch of fuzzing that targets systems by accounting for their state, often through ensuring interactive communication between the fuzzer and the target, as opposed to the singular-input approach of traditional fuzzing.

State is encoded largely in data, and it influences how the system reacts to inputs. Yet, state-of-the-art fuzzers continue to rely only on code coverage guidance when evaluating stateful systems. To explore the vast state space more effectively, the search should be guided by state-aware metrics.

Behavior as a state indicator

Example 4.1



(a) Given labeled cells, a fuzzer can systematically explore the maze.

(b) Without labels, cells can still be identified by their surroundings.

Figure 4.1 – Exploring a maze is an example of a stateful process. Knowledge of the currently occupied cell guides a fuzzer towards new locations.

We motivate stateful fuzzing with the example in fig. 4.1a. If a fuzzer had access to the last reached cell as feedback, it can leverage that knowledge to prioritize paths which uncover new cells, as is the case of coverage-guided fuzzers. A stateless fuzzer would mutate a path—from the set of interesting paths it had already found—and execute it in one shot. The mutated path may or may not introduce moves along the way that would render the rest of the path uninteresting (e.g., walking into a wall). In contrast, a stateful fuzzer would select an interesting path as a prefix, follow it, then generate a move in some direction. The key difference between the two fuzzers is that the latter explores the maze incrementally, whereas the former performs a random walk. The stateful fuzzer is then more likely to solve the maze earlier, since it wastes fewer iterations on invalid moves.

However, in the absence of labels, a fuzzer may be misguided. Consider the example of an unlabeled maze in fig. 4.1b. In a running session, we assume the fuzzer can request one move at a time, and its feedback is restricted to *"whether or not the player moved"*. Without knowledge of the current cell, the fuzzer attributes the feedback only to the last generated move. For instance, if the fuzzer arrives at cell 28 through an upward move from 27, it would consider «upward» as an interesting direction, and may select it more often. Yet, whether or not a player can move depends not only on the attempted move, but also on its surroundings. In another iteration, if the fuzzer starts from cell 5, moving upward would yield no interesting results. Unaware of its surroundings, the fuzzer quickly exhausts the set of interesting behaviors it can observe, and proceeds with a random walk for the rest of the campaign.

Nonetheless, such boolean feedback can still be used to model the player's local surroundings. Having found a few initial paths, the fuzzer can infer characteristics of the cells it has arrived at by trying out, at each path, all the different interesting moves it has discovered so far. This allows it to label each path by the set of possible single-step moves from the cell at the end of that path. In essence, the fuzzer would measure the *response pattern* of each cell to a set of known inputs. This allows it to group its known paths by their common characteristics, e.g., paths which lead to a cell in a vertical corridor. A complete exploration would yield the classification represented in fig. 4.1b: cells are annotated by the possible set of paths that can be followed through one move from each cell. All paths known to the fuzzer then fall into one of 14 categories (colors), based on their surroundings. Increasing the number of steps-ahead yields a more accurate classification such that, in the limit, each path maps uniquely to its cell's original label. Through this process, the fuzzer can leverage a uni-dimensional metric to extrapolate multidimensional feedback for guiding stateful exploration.

In this chapter, we propose *state inference* to address the challenges arising from fuzzing stateful systems. State inference is a technique to produce groupings of snapshots that occupy the same implicit system state, based on similarities between input-response pairs[1]. The key idea is

---

[1] *Response* refers to a measured observable property of the system, i.e. through instrumentation and feedback, as opposed to network server reply messages.

to cross-test inputs snapshots against inputs and observe their behavior to determine a mapping between snapshots and states. In practice, we find that seed queues are often biased to a subset of the functional groups discovered by the fuzzer. State inference achieves an average reduction of 80% in the size of a fuzzer's scheduling queue by grouping together program paths which display equivalent behavior when subsequently probed with different inputs. By enabling the fuzzer to benefit from the knowledge that different snapshots share the same state, the fuzzer can better model the relations between snapshots and distribute energies more equally among the inferred states. As a result, the fuzzer can schedule from the state queue first, to ensure an even exploration, and cycle faster through discovered behavior; e.g., with 10 states describing 50 snapshots, the fuzzer can cycle through the state queue 5 times as fast. This novel technique offers a more hands-off and effective approach to stateful fuzzing, paving the way for improved security testing of complex systems.

We implemented state inference on top of TANGO, our versatile framework for stateful fuzzing. Additionally, we extended current state-of-the-art fuzzers with feedback from TANGO. Our evaluation reveals that state exploration is significantly improved with state feedback when fuzzing targets such as network servers, streaming parsers, and DOOM (see section C.2). We summarize our key contributions as follows:

- Design of TANGO, a framework for fuzzing stateful systems in a state-aware manner.
- Introducing the concept of state inference, which enables the identification of snapshot groups that belong to the same state based on their responses to inputs.
- Implementation of state-aware scheduler extensions, for AFL++ and Nyx, that leverage inferred states to reduce wait times and to disperse feedback.
- Careful evaluation of our approach, comparing its performance with existing fuzzing techniques and demonstrating its effectiveness in analyzing complex systems.
- Open-source access to our framework and results to foster adoption and provide value to the research community.

4.1 Background

Previous work on stateful fuzzing reveals the benefit of state labels on fuzzer performance. IJON [11] is an annotation framework that allows fuzzers to incorporate complex state into their feedback loop. It was used to fuzz Mario Bros. in a process not too different from exploring a maze: knowledge of the last reached location guides the fuzzer towards unexplored regions. Manually annotating a target requires effort and is often skipped in favor of readily-available feedback like code coverage. While alternative techniques attempt to extract state variables from certain types of targets, with certain properties, and varying success [14, 112, 131], code coverage remains the preferred mode of instrumentation.

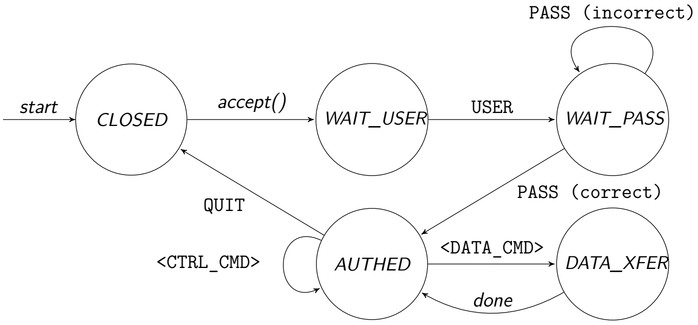


Figure 4.2 – A simplified FTP server state diagram.

Further, stateful fuzzers such as AFLNet [124], SNPSFuzzer [91], and Nyx [140],introduce a key feature for exploring complex systems: *resumability*. It entails the ability to restore the target to a certain state and use that state as a starting point for further fuzzing. They achieve that through restore points, referred to as *snapshots*, which span different granularities, from whole-system VM snapshots, through process restore points, to record-and-replay techniques. Essentially, at each snapshot, the target occupies an implicit state as a result of the path traversed by the input. Perfect resumability ensures reproducibility of behaviors in their respective states, and it allows the fuzzer to efficiently explore different paths without loss of progress.

The complexity of fuzzing stateful targets arises from their dependence on the system's state. Seeds generated due to interesting feedback represent *paths* through the target from which later fuzzing iterations can continue. This presents an opportunity for path explosion that quickly degrades the performance of a fuzzer. The key to tackling path explosion in stateful fuzzing lies in more efficient scheduling, which faces three core challenges:

Feedback Attribution: The fuzzer gradually develops a model of the target through collected feedback, incorporating it into its input generation for further exploration. Consider the example of an FTP server in fig. 4.2. If the fuzzer initially sends a correct sequence of `USER-PASS` commands, it ends up in the *AUTHED* state, where control and data transfer commands are accepted. Now in the *AUTHED* state, the fuzzer sends a control command, e.g. `PWD`, and receives positive feedback reinforcing the use of `PWD` in future iterations. However, the `PWD` command is only valid in an authenticated session context. Starting the session with any command other than `USER` yields a completely different behavior. Thus, without accounting for state, the fuzzer's model of the target receives conflicting or misleading feedback.

Exploration: The discovery of an interesting input can expose many new paths to the fuzzer, since that input may set up the context required to traverse those paths. Following the FTP example, if the fuzzer saves the `USER-USER-PASS-PWD` sequence as an interesting input, it may apply further mutations to it, leading to an input like `jUnK-PASS-PWD`. Unaware of the state set up by the `USER` command, the fuzzer mutates and destroys that part of the input, trapping itself in an error path.

To avoid this loss of progress, the fuzzer should treat previous inputs as part of the state setup. Seen differently, the history of interactions should be considered as one way to restore the current state in the target, e.g. through record-and-replay. Fuzzing is then performed *incrementally* along one path of exploration. Note that, while generating invalid inputs is equally important in fuzzing, this methodology does not rule out the possibility of doing so. To test for an invalid command, it suffices to start from an empty prefix path, or alternatively, only mutate past a selected prefix. This ensures that the state set up by the prefix is not destroyed.

Soundness: If an input triggers a crash, it may not be sufficient to generate a reproducer from that input alone, since the target may have crashed due to previously accumulated state. To guarantee the soundness and reproducibility of crashes, the fuzzer must produce an input that accounts for that state build-up. This requires the fuzzer to track the history of consumed inputs within the lifetime of the target, keeping in mind that fuzzers typically generate millions of inputs.

These core challenges can be addressed by treating *"state" as a first-class citizen* and anchoring the fuzzer's operations around it. Introducing this new dimension to fuzzing requires careful consideration and handling in the form of *snapshot management*, *state modeling*, and *state-aware behavior*.

4.1.1 Snapshot Management

For stateful fuzzing, it suffices that the target persists between successive fuzzing iterations. This allows the target to build up state through processing the consumed inputs. Nevertheless, to tackle the aforementioned challenges with feedback attribution, exploration, and soundness, stateful fuzzers typically divide their progress into increments by taking snapshots. Through these snapshots, the fuzzer can then save and restore a previously discovered path.

Feature

Definition 4.2 A *feature* is a measurable quantity that is influenced by state (e.g., edge hit count).

Feature map

Definition 4.3 A *feature map* is a mapping from a feature identifier (e.g. edge label) to its measured value.

When fuzzing stateful targets, it is common to maintain a tree of snapshots, where each node has successors representing other snapshots. The tree is constructed by appending new snapshots as children of the current node being fuzzed whenever new features, e.g., control-flow edges, are discovered. Stateful fuzzers, such as Nyx-Net [141], AFLNet [124], NSFuzz [131], and SGFuzz [14], have built on this idea, employing different approaches to managing their snapshot tree. Nyx-Net maintains an implicit tree by backtracking input sequences and injecting snapshot commands along the path. Its snapshot tree is then encoded in the input corpus itself. In contrast, fuzzers like SGFuzz, AFLNet, and NSFuzz attempt to approximate the protocol state graph by explicitly constructing a tree based on observed changes in state labels. Nodes then represent unique values of the state variable or response code, and within each node, the system maintains a set of inputs that allow the fuzzer to restore the target to a snapshot of that state. The approximate state graph is then obtained by merging tree nodes with overlapping state labels.

4.1.2 State Modeling

State is a semantic identifier of the system's dynamic nature. Any event or interaction with the system may update its state and modify its behavior. This abstract definition does not expose a direct measure of state, because more often than not, there is none. Although some implementations observe global variables as explicit state identifiers, this does not constitute a generic abstraction over states, as there are often other contributing factors.

In practice, the state of a system boils down to variable memory contents which influence its responses. In the case of computer software, the contents of the call stack, the heap, or function-local variables all factor into the state of the system. While there are attempts at isolating and capturing those variables as state feedback [14, 112, 131], there is no one-size-fits-all method for precisely measuring state without under-approximation or over-approximation. It is often easier for a developer who is familiar with the target to specify their own definition of state that fits the testing goals they are trying to achieve.

State Identification: Recovering the behavioral model of a system is not straightforward. To recover states and the relations between them, AFLNet requires patching the target to augment its responses such that state identifiers are explicitly indicated through the response codes. It also requires that the fuzzer is aware of how those identifiers can be extracted from the received responses. Alternatively, SGFuzz and NSFuzz instrument global variables as state indicators, but they often require manual effort to filter out noise by adding irrelevant variables to an elaborate ignore-list. They also fundamentally assume that state can be consolidated to *global variables*, when in fact, it can span any mechanism for managing persistent memory contents, such as the call stack, function-local variables, or heap objects. On the other end of the spectrum, Nyx-Net foregoes state identification and relies solely on its high reset rates to explore more of the input space, following the blackbox fuzzing school of thought that favors execution speed over introspection.

It is crucial to note that fuzzing is a stochastic process geared toward evaluating the implementation, not the specification. While prior knowledge of the protocol state graph can help guide the fuzzer to explore unvisited states, it is not a characteristic property of the implementation,

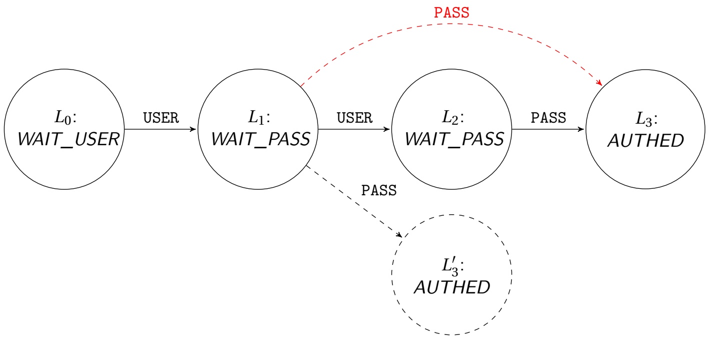


Figure 4.3 – A snapshot tree constructed for an FTP server.

which may hide implicit states or diverge from the prescribed protocol. Although such divergence is interesting for fault detection, it is only discernible where the specification is available, or when a baseline is used for comparison (as is the case of differential testing [106]). A fuzzer cannot know what it does not know. In the absence of a specification, it suffices that the fuzzer schedules its exploration more evenly across the observed feature space. Precise state recovery is thus orthogonal to fuzzing; response codes and state variable annotations are just other forms of feedback about the features that the fuzzer has discovered.

Response-based Grouping: Fuzzers that rely on annotations as state feedback—including global variables [14, 112, 131], response codes [124], or manual labels [11]—typically incorporate it as a signal for grouping distinct snapshots. In the lack of a grammar specifying what transitions are valid at each state, the value provided by these annotations to an evolutionary fuzzer is thus limited to the grouping of snapshots among distinct behaviors. This reduces the load on the scheduler by iterating over states rather than snapshots and allows the fuzzer to aggregate feedback from fuzzing snapshots within the same state. We observe that it is then sufficient to recover groupings among snapshots which display similar behavior to bootstrap the evolutionary fuzzing process.

Consider again the example of the FTP server in fig. 4.3. When the fuzzer first sends the `USER` command, it takes a snapshot L_1 (in the *WAIT_PASS* state), since it observes new feedback relating to the discovery of a new command. The fuzzer then follows with `USER` again, taking a new snapshot L_2 due to executing the error handler in *WAIT_PASS*, and it remains in that state. Sending the `PASS` command then results in L_3 (in the *AUTHED* state). The fuzzer has now discovered a path to the *AUTHED* state as `USER-USER-PASS`. Notably, the fuzzer discovered the *WAIT_PASS* state “twice”, but the *AUTHED* state only once. Depending on the type of feedback collected by the fuzzer, it may also be incapable of discovering the `USER-PASS` sequence from L_1, since the feedback for discovering the `PASS` command had already been attributed to L_3, and it is no longer considered an interesting signal for taking a snapshot. In effect, the discovery of the `USER-PASS` sequence was *over-shadowed* by `USER-USER-PASS` due to overlapping feedback. This phenomenon likely occurs many times throughout the campaign, stacking up the snapshots and diluting the scheduling pool.

Yet, despite L_1 and L_2 traversing different paths in the target, we know they represent the same state: the target expects a `PASS` in either case to transition to the next state. If we assume L_3 was never created, and we send `PASS` at L_1, we would discover *AUTHED* again and would create a snapshot L'_3. Conveniently, through this approach, the fuzzer discovers a shortcut to *AUTHED* which involves the minimal number of commands required to reach it.

This observation is not limited to authentication routines but generalizes to any stateful system whose behavior is primarily influenced by its inputs. At each snapshot, the system occupies *some* implicit state. By probing the system with different inputs and measuring its responses, we can develop a model of the state it is in. If two snapshots share the same responses across all tested inputs, we can then consider them as belonging to the same state, as far as the fuzzer is concerned.

4.1.3 State-aware Operation

A stateful target is one whose behavior changes depending on the state of the system, implying that inputs consumed by the target must be generated by accounting for the current state. Beyond using snapshots as a prefix for incremental exploration, none of the mentioned state-of-the-art fuzzers incorporates state into its operations, such as mutator schedules or state-specific dictionaries. Incorporating feedback in a state-specific manner helps develop a more accurate model of the target in each state, better guiding the fuzzer for exploring further within each state. Granted, feedback can become overwhelming for the fuzzer [48, 50, 158]. State-dependent feedback can be even more challenging to handle, as the target's behavior evolves dynamically, necessitating that feedback be incorporated dynamically as well. Nonetheless, to make the most of the collected feedback, a state-aware fuzzer should treat state as a distinct *dimension* for its operations across all phases of the fuzzing process.

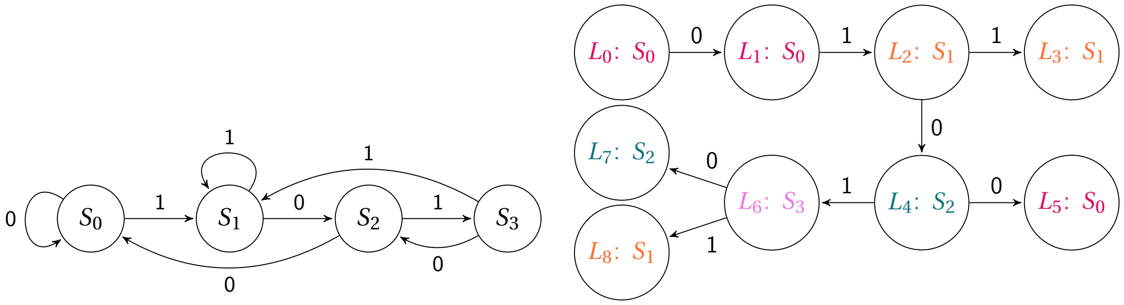
4.2 State Inference

Inspired by system identification and hypothesis testing, *state inference* models the behavior of the target through its responses to a set of inputs.

Response

Definition 4.4 A *response* is a value instance of the feature map obtained by executing the target with a given input.

At each snapshot, the target occupies a unique hidden state that drives its behavior and influences its response to inputs. For a stateful system, we posit that *two instances of the system occupy the same state—as far as the fuzzer is concerned—if both instances share the same response pattern across all tested inputs*, given a sufficient number of samples. We use this insight to



(a) A state diagram of a system that detects the string *b'1011'*.

(b) One example of a corresponding snapshot tree constructed by the fuzzer. Each node is annotated with the implicit state that the system occupies at that snapshot. Nodes are colored according to the equivalence states they belong to after *subsumption*.

Figure 4.4 – An example application of the state inference procedure on a simple FSM-based system.

develop a systematic approach for evaluating snapshots, grouping them by their observable response patterns, and extrapolating relations among them to guide exploration. This grouping yields a hierarchy of states and snapshots that enables fairer and more targeted seed scheduling.

This process requires additional executions to probe the target along all the interesting paths discovered by the fuzzer. Fortunately, the benefits can outweigh the overhead costs due to the non-linearity of exercising new coverage: while fuzzing iterations grow in linear time, new features are only discovered in exponential time [20]. This means that, as the fuzzing campaign progresses, the cost of state inference is amortized over the time spent by the fuzzer between successive coverage findings. In the meantime, the fuzzer can leverage the learned state model and the relations between snapshots for better input and mutator scheduling. As we discuss in section 4.3, the overhead of processing new features tapers off, while still allowing the fuzzer to benefit from the refined feedback. Coverage-guided fuzzing is an evolutionary learning process, and as with any learning algorithm, filtering out the noise from the data comes with a cost but an even higher reward.

4.2.1 Cross-Pollination

In the process of generating and testing inputs, the fuzzer discovers and records those that trigger new features within the target. Working under this assumption, every recorded input is guaranteed to elicit a response in at least one snapshot, that which was active at the time the input was first discovered. Using the battle-tested mechanism of a cumulative global feature map, where observed features are only considered interesting the first time they are encountered, the fuzzer is incapable of rediscovering the same feature across different contexts. In other words, if the same behavior can be triggered along different paths, only one of those paths can be discovered by the fuzzer. Through cross-pollination, we aim to discover these hidden *capabilities* by re-applying the same input at different contexts (i.e., snapshots) and observing overlaps in feedback.

Capability

Definition 4.5 A snapshot is said to have a *capability* when the response to an input measured in that snapshot matches the expected response, under the null hypothesis that two snapshots in the same state cannot be distinguished based on their responses.

We illustrate this procedure with an example: consider the system with the state graph prescribed in fig. 4.4a. After some simulated rounds of fuzzing, we arrive at the snapshot tree depicted in fig. 4.4b. Whereas the fuzzer observed unique features—namely, state transitions in the system's FSM—that resulted in this snapshot tree, we note that multiple snapshots occupy the same state. Unaware of this overlap, the fuzzer would schedule each of these snapshots individually and would integrate feedback into state-agnostic models. This results in duplicate efforts, as the fuzzer wastes many cycles on testing the same states along different paths. However, given this initial snapshot tree, the fuzzer can leverage cross-pollination to discover hidden relations between snapshots, allowing it to better model their overlap and optimize its exploration.

We define the response pattern of an input applied at a snapshot as the set of features triggered while executing that input. Given enough input samples, the behavior of a snapshot can be modeled by measuring its different response patterns. Conveniently, the fuzzer is continuously searching for inputs which trigger a non-empty set of unique features, recording those as new snapshots in the tree. This provides the cross-testing process with an initial corpus of interesting inputs. We leverage these recorded inputs, whose response patterns in the parent snapshot are known, to cross-test the behavior of all snapshots. Repeated application of this procedure throughout the fuzzing campaign ensures that snapshots are assessed against an increasing amount of representative samples, while avoiding the need to generate new samples solely for the purpose of cross-testing, as that part is covered by the fuzzing process itself.

If a snapshot can generate the same response pattern as the input's parent snapshot, we say that the snapshot is *capable* of reproducing that response. To discover the capabilities of all snapshots, we iterate over all inputs in the snapshot tree and apply them at every snapshot, measuring its response pattern, and recording its ability to reproduce the original response. In constructing the snapshot tree, every response pattern is associated with an edge between a parent and a child snapshot. As such, if a snapshot is capable of a response, it is equivalent to having an edge between that snapshot and the corresponding child node in the snapshot tree. To record capabilities, we thus extend the tree's adjacency matrix into a *capability matrix*. We dub this process of exploring capabilities as *cross-pollination*.

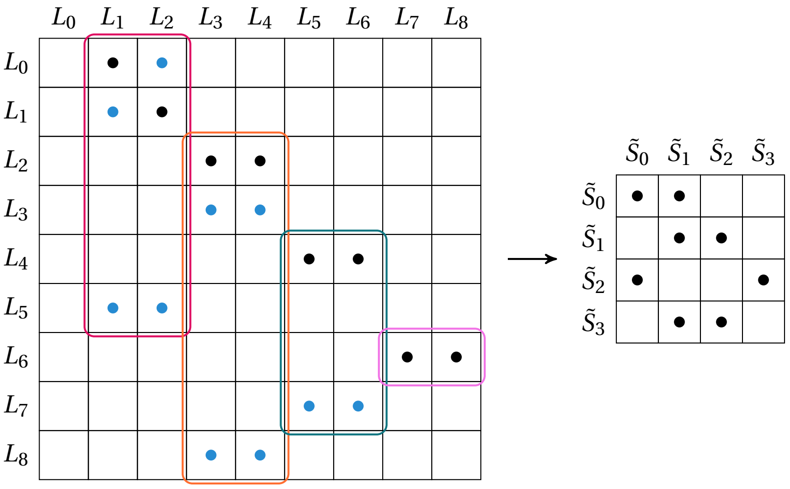

L_0 L_1 L_2 L_3 L_4 L_5 L_6 L_7 L_8
$\tilde{S}_0$ $\tilde{S}_1$ $\tilde{S}_2$ $\tilde{S}_3$

Figure 4.5 – The capability matrix A_C (left), which is obtained after *cross-pollination* by uncovering edges—marked in blue—between snapshots. With *subsumption*, we group up nodes which mutually overlap in capabilities. Finally, we *collapse* the matrix onto a graph of equivalence states that model the relations between snapshots (right).

Capability matrix

Definition 4.6 A *capability matrix* is a matrix of Snapshots × Responses → Capability, where non-empty cells store the input that triggers a known response.

The cross-pollination procedure constructs a directed bipartite graph, $G_C = (L, R, E)$, where L is the set of all snapshot nodes, R is the set of responses associated with inputs from the snapshot tree, and E is the set of edges mapping L to R, each carrying the enabling input, representing the capability of each snapshot in replaying a response. The number of responses is $r = |R| = |L| - 1 = l - 1$, since every snapshot is associated with a response except the root. The adjacency matrix of the snapshot tree is a starting point for a capability matrix, A_C. By looping over the snapshots and inputs, we iteratively extend the capability matrix with new edges. Dismissing the first empty column, $A_C : L \times R$ has the dimensions $l \times r$, and the adjacency matrix of G_C then has the dimensions $(l + r) \times (l + r)$ and is defined as:

$$B_C : L \times R = \begin{bmatrix} 0_{l,l} & A_C \\ 0_{r,l} & 0_{r,r} \end{bmatrix}$$

Since B_C is an upper right diagonal matrix, it is then practical to only consider A_C in the following steps. In implementation, it also helps to keep the original dimensions of the adjacency matrix $(l \times l)$ as depicted in fig. 4.5. It should be noted that this simplification implies that, in applying *subsumption* and *collapse*, matching involves only the outward edges in A_C, i.e., row-wise matching. However, in the general case of a full adjacency matrix, matching should be applied over both outward and inward edges.

4.2.2 Subsumption and Elimination

The populated capability matrix provides insight into which snapshots overlap in behavior, and consequently, how distinct snapshots are. This knowledge is essential for better guiding the scheduler towards exploring unique functionality without duplicating efforts and stalling progress. To find that overlap, we develop the subsumption operator over graph vertices ($\prec$). In short, a node u is subsumed by v iff v can replace u without affecting reachability, i.e., v has at least all the same edges as u. This implies that, in the adjacency matrix, where u is present along either axis, v is also present. In the context of the capability graph G_C, and its matrix A_C, v must overlap u where the successors of u are concerned.

Capability graph

Definition 4.7 If we transform the extended capability matrix into a vertex adjacency matrix by replacing the columns (responses) with the snapshots which were first attributed to discovering those responses, we can construct a *capability graph* based on this adjacency matrix.

Taking fig. 4.5 as an example, we observe that $\{L_0, L_1, L_5\}$, $\{L_2, L_3, L_8\}$, $\{L_4, L_7\}$, and $\{L_6\}$ form sets of snapshots with mutually overlapping responses. Each set of snapshots is then called an *equivalence state*. Notably, equivalent snapshots may traverse different paths through the system and thus cannot be pruned solely through power schedules [22], since state information is not captured by code coverage alone.

In practice, a fuzzer may also encounter snapshots whose response sets are proper subsets of others' response sets. From the fuzzer's perspective, such snapshots are less capable and thus not worth dedicating a scheduling slot for, despite having non-conforming behavior. Any response elicited in that snapshot can be reproduced in another having at least one additional capability.

To formalize the approach, we propose the following definitions:

$N^{\square}(u)$
: The $\square$-*neighborhood* of u, where $\square \in \{+, -\}$ represents outward and inward directions, respectively.

$u \preceq^{\square} v \iff N^{\square}(u) \subseteq N^{\square}(v)$
: u is $\square$-*subsumed* by v.

$u \prec^{\square} v \iff (u \preceq^{\square} v) \land (v \not\preceq^{\square} u)$
: u is *strictly* $\square$-*subsumed* by v.

$u \preceq v \iff (u \preceq^{+} v) \land (u \preceq^{-} v)$
: u is *subsumed* by v.

$u \prec v \iff (u \preceq^{+} v) \land (u \prec^{\square} v)$
: u is *strictly subsumed* by v, for any $\square \in \{+, -\}$.

$u \sim^{\square} v \iff (u \preceq^{\square} v) \wedge (v \preceq^{\square} u)$

u and v are $\square$-*equivalent*.

$u \sim v \iff (u \sim^{+} v) \wedge (u \sim^{-} v)$

u and v are *equivalent*.

For each proposed operator op, we can construct a set of vertices $u \in U$ which satisfy $u\ op\ v$ as: $op_v = \{u \in U | u\ op\ v\}$. Of particular interest are the equivalence sets $\sim_v$ and strict subsumption sets $\prec_v$.

To extract the sets of equivalence states from a capability matrix, it suffices to construct the set of out-equivalence sets:

$$\tilde{S} = \{\sim_v^{+} | v \in L\}$$

Each element in $\tilde{S}$ represents an equivalence state, a set of snapshots that overlap in behavior. To further reduce the number of states, the fuzzer calculates the set of non-empty strict subsumption sets:

$$\check{S} = \{\prec_v \neq \phi | v \in L\}$$

Following that, any snapshot belonging to any set in $\check{S}$ can be safely eliminated from all sets in $\tilde{S}$, without loss of capabilities. Alternatively, to avoid inadvertently losing quality inputs, we choose to include strictly subsumed snapshots in the equivalence states of the subsuming nodes. This ensures that the fuzzer experiences the same reduction in state counts while maintaining access to all interesting snapshots.

4.2.3 Colorful Collapse

After subsumption and elimination, we obtain a reduced set $\tilde{S}$ of equivalence states $\tilde{S}_i$. Within each state lies a collection of snapshots that share the same behavior across all tested inputs; the set of $\langle$*input*, *response*$\rangle$ pairs is called the *capability set* of the state.

Capability set

Definition 4.8 The *capability set* of a snapshot is the set of $\langle$input, response$\rangle$ pairs constructed from the non-empty columns along the snapshot's row in the capability matrix.

In applying an input $I_{a,b}$ from the capability set at any snapshot in $\tilde{S}_i$, we can reproduce the response pattern which, if non-empty, was initially displayed by applying that input to the active snapshot L_a in the snapshot tree. By construction, that input led to the addition of a new child snapshot L_b. We can then say that all snapshots in $\tilde{S}_i$ can reach L_b through $I_{a,b}$, because they all elicit the same characteristic response of L_b.

This is directly observed when the capability matrix A_C is cast to an adjacency matrix of snapshots, as depicted in fig. 4.5. To simplify the aggregation of snapshots, such that scheduling and feedback are performed at the *state* level, we can collapse the adjacency matrix through vertex contraction [119] over $\tilde{S}_i$.

Since the capability matrix specifies the response of every snapshot to a *single* input, we consider those as *first-order responses*; recall that in the maze analogy, we also limited the inference to one step away from each cell. Behavior of the snapshot after applying the first input can only be modeled by further cross-testing, along an additional dimension, to obtain capability tensors of higher-order responses. However, going beyond the first order significantly increases the overhead of state inference and may only partition the snapshot groupings even further; its added value is higher accuracy. Fuzzing is often tolerant to errors, due to the dampening effect of random sampling.

The *collapsed matrix* models the first-order relations between states. In our procured example, this matrix recovers the original state graph presented in fig. 4.4a. While certainly desirable, this was mainly driven by two factors:

Lossless features: Transitions in the snapshot tree coincide with those in the state graph; there are exactly 8 of each. This implies that the feature feedback to the fuzzer was capable of capturing these state transitions, without a loss in accuracy or precision. While it is not a coincidence that our tailored example displayed this behavior, it is unlikely that features are lossless in practice. The nature of the feedback (e.g., code or data coverage) dictates the accuracy and precision of the response patterns. In the FSM example, the fuzzer also managed to explore all transitions through different generated inputs. However, completeness is not guaranteed under fuzzing.

Smooth transitions: In constructing the snapshot tree, the fuzzer encountered only *new* features at any active snapshot, corresponding to triggering only unseen transitions in the state graph. Had the fuzzer triggered multiple transitions in the target without observing new features, then the edge to the next recorded snapshot is not guaranteed to overlap a transition in the state graph. In effect, the fuzzer would have discovered a *path* through the state graph, rather than a single transition; yet, it would record it as one edge in the snapshot tree. It is similarly difficult to achieve "smoothness" in practice. In the maze analogy, we enforced smoothness by limiting the fuzzer to one move at a time.

To mitigate the imprecision and inaccuracy of feedback, and to avoid misguiding the fuzzer with false assumptions about state equivalence, we propose to maintain the original snapshot tree and color it with state labels. This ensures that implicit state build-up in equivalent snapshots is not lost during collapse, and that consequent application of the state inference process does not compound the errors, but rather reduces them as more response patterns are measured and evaluated. This also allows us to iteratively collapse the same matrix to obtain a *minimal recovered*

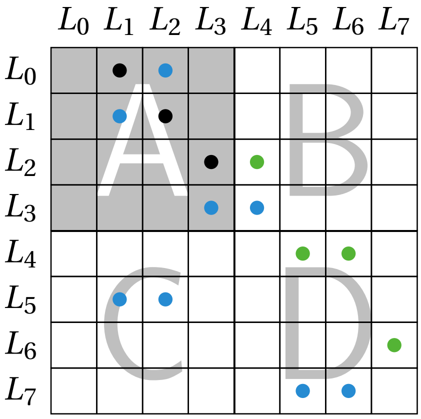


Figure 4.6 – The capability matrix of the second round, with $m = 4$ new snapshots. The shaded quadrant Q_A contains the populated matrix from the previous round. The Q_B and Q_D quadrants are overlaid with edges—marked in green—from the latest adjacency matrix. Q_C is always initially empty, since new snapshots are only successors of old ones.

model of state relations, on average obtaining a further 10% reduction in the number of states in practice, atop the 80% reduction due to subsumption.

4.2.4 Successive Rounds of Inference

After applying state inference, we obtain a capability matrix which carries the fruits of cross-pollination, along with a collapsed matrix which models the relations between labeled snapshots. This labeling, however, is static, and applies only to the cross-tested snapshots. As the fuzzer progresses, it will discover more snapshots which remain unclassified and do not benefit from the results of state inference. It is then necessary that the process is continuously applied throughout the fuzzing campaign.

One straightforward approach is through batching: for every batch of m new snapshots, we re-apply the inference routine, extending and updating the capability matrix from the previous round. We illustrate the state of the extended capability matrix of the second application round in fig. 4.6. Note that entries in quadrant Q_A need not be revisited, since capabilities are assumed to be reproducible. After overlaying discovered edges from the latest snapshot tree onto the new capability matrix, we can continue to apply the state inference procedure as prescribed. Alternatively, batching can be scheduled in time slots, e.g. every N minutes, accommodating for the non-linear increase in coverage by performing inference on the new snapshots generated since the last run. An adaptive hybrid approach can combine the two strategies to reduce the startup cost of inference and maximize information gain throughout a fuzzing campaign.

While we present our approach as an independent step post-discovery, it can actually benefit from continuous integration with the fuzzing campaign. The fuzzer spends much of its time generating inputs that do not trigger new features. Nonetheless, the feedback is often non-empty: these uninteresting inputs likely elicit known response patterns which may overlap those of other snapshots. This observation allows the fuzzer to dynamically extend the capability matrix at a minimal cost: the computational overhead of matching.

4.3 Overhead, Optimizations, and Trade-offs

To uncover hidden capabilities, state inference relies on cross-validation, a technique notoriously known for its quadratic complexity. For every batch of m new snapshots, given n existing ones, the fuzzer must perform additional executions on the order of $\mathcal{O}(m \times n)$.

In particular, we define the following quantities:

Time-step k: the point in time at which the kth application round of state inference is performed.
Snapshots $n_k = n_{k-1} + m = m \times k + n_0 = mk$
States $s_k = s_{k-1} + \underbrace{\alpha}_{\text{average reduction ratio}} m = \alpha mk$
Cross-tests $c_k = c_{k-1} + \overbrace{m(2n_{k-1} + m - 1)}^{\text{additional executions per round}} = n_k^2 - n_k$

Applying state inference in batches of m new snapshots requires that the fuzzer discovers m new features. But coverage growth is linear in exponential time [20]: to discover m new features, the fuzzer spends on the order of $\exp(m)$ more time than for the last batch. Hence, during the kth step, at time t_k, the total number of cross-tests performed is $\mathcal{O}(\log^2(t_k))$. Meanwhile, the fuzzer generates and tests new inputs in linear time, diminishing the ratio of time spent on state inference as:

$$\lim_{t \to \infty} \frac{\log^2(t)}{t} = 0$$

As the fuzzer continues to progress, the overhead cost of state inference is amortized over the executions performed between rounds. In the meantime, it reaps the benefits of modeling state relations in scheduling and in generation. We assess these costs and benefits through our evaluation in sections 4.5.2 and 4.5.3.

The ramp up cost of state inference is, however, high. A fuzzer finds the most coverage in the first few epochs of the campaign, necessitating frequent rounds of cross-testing. Whereas the overhead tapers off at the tail, the initial costs can overwhelm the fuzzer, making it spend most of its time in the beginning just on inference, and bringing its progress to a slow halt. The cost of ramping up greatly varies with initial seed coverage, the complexity of the target, and the execution speed, among other variables.

To address the ramp up cost, we propose several optimizations that reduce the overhead of state inference, at the cost of some accuracy in grouping snapshots. The prescribed cross-testing procedure requires that all cells of fig. 4.6 outside of Q_A be tested for adjacency. We propose several optimizations (one for each quadrant) to introduce *savings* in the form of skipped tests, thereby reducing the number of resets and executions required for each round of inference.

4.3.1 State Broadcast (O_B)

If we were to assume that inferred states in Q_A are correct once constructed, then cross-testing the capabilities of individual snapshots within the same state becomes unnecessary. Working under that assumption, equivalent snapshots always have the same capabilities, which we can evaluate as a property of the state they belong to. For a state with N snapshots, it thus suffices to perform at most m cross-tests, instead of $m \times N$. Discovered capabilities in Q_B are then broadcast to all snapshots within a state, reducing the number of cross-tests:

$$\begin{aligned}\tilde{c}_k &= \tilde{c}_{k-1} + m(n_{k-1} + s_{k-1} + m - 1) \\ &= mk\left[\frac{\alpha+1}{2}m(k+1) - \alpha m - 1\right]\end{aligned}$$

The approximate reduction in total cross-tests is then:

$$(1-\alpha)\frac{k-1}{2k}; \text{when } \frac{1}{m} \ll 1$$

In applying state broadcast as an optimization, the overhead of inference is distributed among states, rather than snapshots. This results in higher prediction accuracy for “uncommon states”, i.e., those with fewer snapshots, which are arguably of more interest to the fuzzer. Beyond the first round of state inference ($k > 1$), cross-pollinating states—instead of snapshots—yields an overall reduction of $\frac{1}{2}(1-\alpha)$ in the number of cross-tests performed. Thus, we make a trade-off in reducing the initial and overall cost of state inference, at the risk of mislabeling snapshots.

4.3.2 Response Fingerprinting (O_C)

After one round of state inference, we obtain a capability matrix in Q_A. Each state and its associated capability set serve as labeled data for training a decision tree (DT) classifier. Such a classifier can optimally divide the input space, enabling us to primarily cross-test inputs which maximize information gain, reducing the number of tests in Q_C.

We fit a DT over this training data to infer *response fingerprints*: minimal subsets of capability sets which are characteristic identifiers of states. The DT classifier yields a binary tree, where each internal node tests for a capability, such that nodes closer to the root yield higher information gain. Leaf nodes consequently carry a value indicating the most likely candidate state.

Following this procedure, for each new snapshot in the inference batch (Q_C), we traverse the DT from the root and query it for the next capability to test, until we hit a leaf node. At this point, the candidate state is identified. The tested capabilities along the path form a subset of the capability set of the candidate. Nonetheless, to reduce false positives, and to satisfy the rules of subsumption, we extend the testing to all non-empty responses in the candidate's capability set.

Consider the second round in fig. 4.6. We fit a DT over the capability sets of $\tilde{S}_0 : \langle L_1, L_2 \rangle$ and $\tilde{S}_1 : \langle L_3 \rangle$. In this simplistic example, a test on any of $\{L_1, L_2, L_3\}$ is enough to yield a classification. With the given labels, it is thus sufficient to perform one test, instead of four, to classify a new snapshot. A snapshot that passes the test has a capability set that matches that of $\tilde{S}_1$, making it at least equivalent. However, failing the test implies an empty capability set, yet the classifier would predict $\tilde{S}_0$. To properly test for equivalence, we must test against $\tilde{S}_0$'s entire capability set in the least (for a total of three tests). This allows us to properly classify L_5 in fig. 4.6. Note that, if the capability set of a new snapshot L_u matches that of the candidate state $\tilde{S}_v$ using DT classifiers, we can only infer that $\tilde{S}_v \preceq^+ L_u$, since we do not have information about what we did not test. Whether or not $\tilde{S}_v \prec L_u$, the result is then the same: a state $\tilde{S}_w = \tilde{S}_v \cup \{L_u\}$. Combined with coloring, this approach ensures that state inference remains consistent under sparsity. On average, the reduction in the total number of crosstests is expressed as

$$\frac{1}{4k}\left[(k-2)(1+\frac{\alpha-1}{k})+\alpha k\right]$$

4.3.3 Test Extrapolation (O_D)

With response fingerprinting, we can reduce the number of tests to be done in the Q_C quadrant of fig. 4.6. However, Q_D remains to be fully tested. Building on top of the DT's predictions, we can leverage Q_B to extrapolate which tests in Q_D need to be applied. Instead of cross-testing all m new snapshots against each other, we can reduce the cost of inference by extrapolating tests from the new state capabilities.

After finding a candidate state to which each snapshot belongs, we populate Q_B to explore the new capabilities of pre-labeled snapshots (i.e., those in Q_A). We follow that by testing new snapshots only against the new capabilities of their candidate states. As with response fingerprinting, this process ensures correct subsumption conditions, minimizing the risk of mislabeling snapshots.

4.4 Tango: The Framework

State inference extends the fuzzer's knowledge base with information about the behavior of the target, which could be leveraged for improving its exploration through (a) higher scheduling efficiency over states; (b) training of state-specific models for inputs and mutators; and (c) better approximation of state relations. Nonetheless, the technique introduces new definitions and requires components which are not explicit in existing fuzzers or frameworks, such as snapshots, states, and transitions. To assess and evaluate the feasibility and benefits of state inference, a whole re-write and restructuring of the typical fuzzing workflow is needed. The lack of an existing framework for state-aware fuzzing and the inflexibility of existing monolithic fuzzers motivated the design and development of Tango: a modular and extensible state-aware fuzzing framework.

4.4.1 Workflow

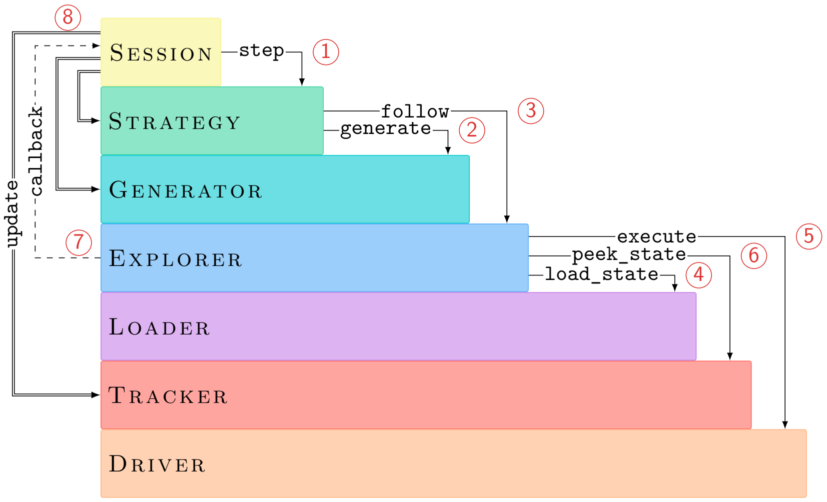


Figure 4.7 – The general workflow of the Tango framework.

Anchored around the notion of state-sensitivity, Tango generalizes over the traditional fuzzer architecture and offers a flexible environment for developing tailor-made fuzzers of stateful systems. The workflow is presented in fig. 4.7. In the context of a fuzzing Session, we ① iteratively step through a Strategy that governs the exploration and exploitation efforts of the fuzzer. Provided with knowledge of the current state of the system, ② the Strategy chooses a target state to fuzz and invokes the Generator to construct a candidate input, suitable for application under that state. The former then ③ forwards the input to the Explorer, which ④ ensures that the system occupies the target state, then ⑤ executes the input through its Driver. With the help of the Tracker, ⑥ peeking into the new state of the system enables the Explorer to record any observed changes. The latter then ⑦ forwards its findings over a callback to the SESSION, which ⑧ broadcasts any updates to concerned components, thus closing the feedback loop.

4.4.2 Implementation

TANGO is implemented in Python 3.11, for its flexibility and ease-of-use. It presented a good back-end for developing TANGO as a research-oriented tool for implementing and evaluating stateful fuzzing techniques.

AsyncIO: Async I/O is a form of cooperative scheduling, where the application specifies when control is returned to the scheduler. We built TANGO as an asynchronous application to enable graceful suspension of the fuzzer and extent its compatibility to event-driven systems, such as DOOM.

Built-in Extensions: TANGO ships with a set of complementary modules that enable it to fuzz x86_64 processes on Linux-based systems, reload state through replayed inputs, measure and classify SanitizerCoverage feedback, communicate over standard file descriptors and network sockets, train state-specific mutators, and perform state inference. The various extensions are summarized in section C.1.

Hotpluggable Inference: We implemented state inference both as a strategy for use in TANGO, and as a plug-in to third-party fuzzers such as AFL++ and Nyx-Net. We slightly augment the scheduling routines of those fuzzers to incorporate the inference results generated by TANGO during a fuzzing campaign and provide them with state-specific feedback.

4.5 Evaluation

State inference is a mechanism for distilling behavioral patterns of the target through cross-testing its snapshots against different inputs, to identify functionally-distinct states. To assess the effectiveness of this technique and pinpoint its potential use cases, we address the following research questions through distinct evaluation campaigns:

RQ1 What is the cost of cross-testing?
RQ2 How well do fuzzers distribute cycles to functionally-distinct snapshots?
RQ3 What is the potential reduction expected in queue sizes?
RQ4 How does code coverage correlate to state coverage in evaluating stateful fuzzers?

4.5.1 Experimental Setup

To quantify the cost and benefits of a new technique, it must be compared against a baseline where only key features are different. These features must then be tested individually. Since we implemented state inference on top of TANGO, we perform the parametric analysis with TANGO itself as the baseline, by toggling new features and sweeping over parameters. This enables us to measure the induced effect of each new aspect by changing one variable at a time, discounting the possible variances from runtime effects, implementation artifacts, or a richer set of mutators and schedulers, such as those available in AFL++ [49].

To that end, we tackle **RQ1** through a set of experiments on TANGO-INFER, a version of TANGO configured to use state inference as a strategy. Furthermore, to assess the potential inefficiencies of scheduling functionally-equivalent snapshots (**RQ2**), we dispatch a set of campaigns to state-of-the-art stateful fuzzers, collect their seed queues, and replay them through TANGO-INFER to find functional groupings among the fuzzers' snapshots and analyze the skew of snapshots in explored states. Additionally, for **RQ3**, we use the results of state inference on fuzzer queues to approximate the expected reduction ratio α encountered across different targets. Finally, to assess the practical advantage of state inference, we set up augmented fuzzing campaigns on top of state-of-the-art baselines, where state inference is run periodically to condense the seed queue. To answer **RQ4**, we report the achieved coverage with and without inference, and we measure the benefit of state inference as the proportion of equivalence states discovered uniquely by each fuzzer.

We perform evaluations against version-anchored targets from the ProFuzzBench [113] suite, in addition to a set of three stateful parsers: `libexpat`, `yajl`, and `llhttp`. While parsers are traditionally considered stateless, our evaluation highlights their stateful nature, as well as TANGO's flexibility in fuzzing diverse data channels. We run 24-hour campaigns, each with five trials to account for randomness.

4.5.2 Empirical Overhead

During inference, the target is reset, and inputs are executed for cross-testing every cell in the capability matrix. To assess the overhead of this operation, we measure the time spent by the fuzzer and the number of tests performed, across varying settings of batch size m and enabled optimizations.

Figure 4.8 shows the evolution of overhead as a function of time. At the start of a fuzzing campaign, coverage grows rapidly, resulting in a high rate of snapshots and frequent invocations of state inference. Initial seed coverage results in many snapshots being generated within the first few epochs, further contributing to the spike in startup inference cost. As exploration speed tapers off, the fuzzer continues to generate and execute inputs, progressively spending less of its time

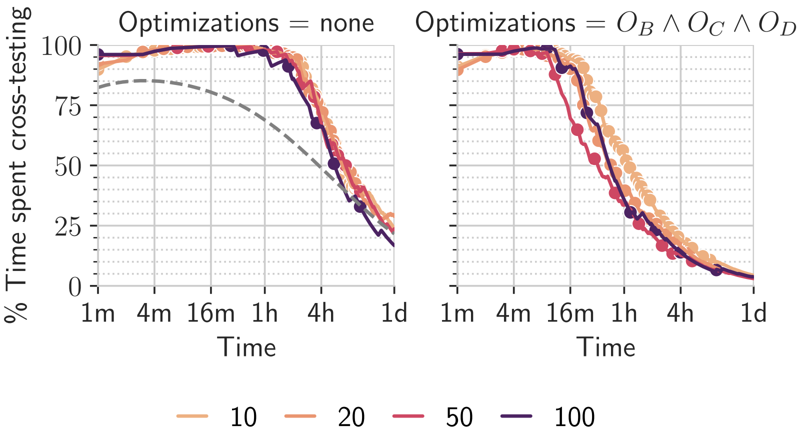


Figure 4.8 – The median overhead of state inference as the proportion of time spent cross-testing, when optimizations are disabled (left) and enabled (right), as a function of time (logscale). Measurements are repeated for increasing values of batch size m. Scatter plots along the curves indicate the frequency of inference rounds: the lower m is, the more frequently inference is performed. The - - - line shows the expected overhead trend in the absence of optimizations.

on cross-testing. However, it leverages the knowledge gained from previous applications of state inference to schedule its exploration more evenly across the functionally-distinct snapshot groups. As discussed in section 4.2.4, it is essential to continuously apply inference on new snapshots. Otherwise, the fuzzer regresses in the direction of high-density regions [22].

Optimizations to state inference reduce the ramp-up cost and allow the fuzzer to resume normal operation faster. Figure 4.9 breaks down the trade-off between the introduced savings and the sacrificed accuracy. Rather surprisingly, the impact on accuracy is limited, even with all optimizations enabled. For a batch size of $m = 50$, enabling all optimizations yields over 80% in savings, while still guaranteeing around 90% prediction accuracy. This explains the accelerated drop in overhead seen in fig. 4.8: with optimization, overhead time drops below 20% by the 4-hour mark, compared to the 24-hour interval observed when optimizations are disabled. Figure 4.9 also highlights the effects of utilizing larger batches: they serve as a stronger basis for making correct predictions, bearing in mind alternative strategies for accommodating to the non-linear nature of exploration (see section 4.2.4).

One notable outlier is `dnsmasq` (✖), which seems to introduce the most frequent misses. Optimizations rely heavily on the results of previous rounds of state inference. If a grouping is incorrectly established, it could remain divergent for the lifetime of the fuzzing campaign, especially since, in our implementation, matching is not error-tolerant. In the case of `dnsmasq`, the inherently stateless nature of the DNS protocol dictates that there is no functional difference when processing successive inputs. However, if the fuzzer falsely generates distinct groupings, then future rounds of inference, and optimizations within them, could propagate the errors (unless groups are merged

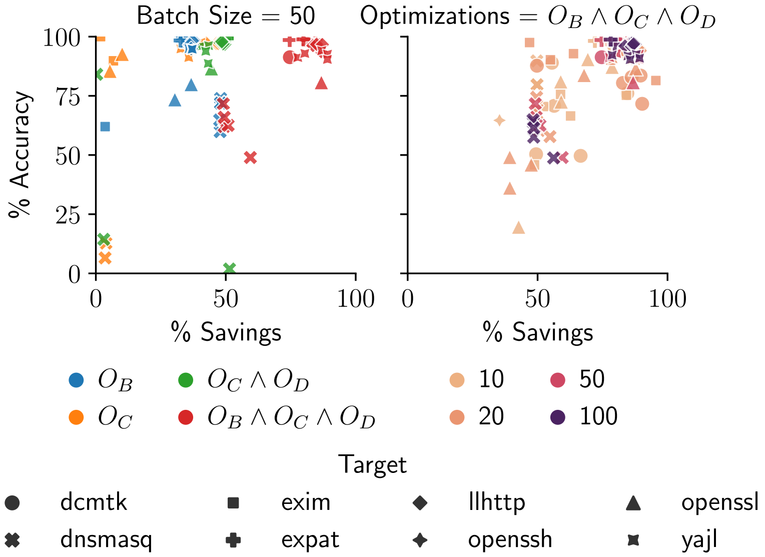


Figure 4.9 – The hit accuracy of optimizations as a function of introduced savings. We fix the batch size to $m = 50$ (left) to observe the effect of each optimization setting. We also measure the effect of a varying batch size (right) with all optimizations enabled.

again under subsumption). Throughout our evaluations, nondeterminism and instability of the targets were the leading cause for providing misleading feedback to the fuzzer. The precision of the feedback metric—in our case, code coverage—and its non-correlation with program state also pose as a source of inaccuracy, which could be alleviated through error tolerance.

4.5.3 Snapshots in Biased Queues

Whenever an evolutionary fuzzer encounters interesting coverage, it saves the input that caused it in its seed queue. For stateful fuzzers, these seeds serve as snapshots to restore the target to the reached state. To quantify the benefits that seed scheduling through state inference could provide, we measure the distribution of fuzzer-generated snapshots across functionally-distinct equivalence states. If we assume that a state-unaware fuzzer selects seeds from the queue with a uniform distribution (i.e., it assumes that seeds are evenly spread across target functionality), then we can assess the “surprise” [144] of sampling uniformly from a skewed population. We calculate $\hat{D}_{KL}(\mathbb{S}||\mathbb{U})$, the Kullback-Leibler divergence [87], of the observed $\mathbb{S}$napshot-in-state distribution against a $\mathbb{U}$niform reference, normalized by $log(N)$, where N is the total number of snapshots observed in each campaign.

In this experiment, we run AFLNet, Nyx-Net, and TANGO-INFER against compatible targets, collect their seed queues, and apply state inference to extract snapshot groupings. We present the

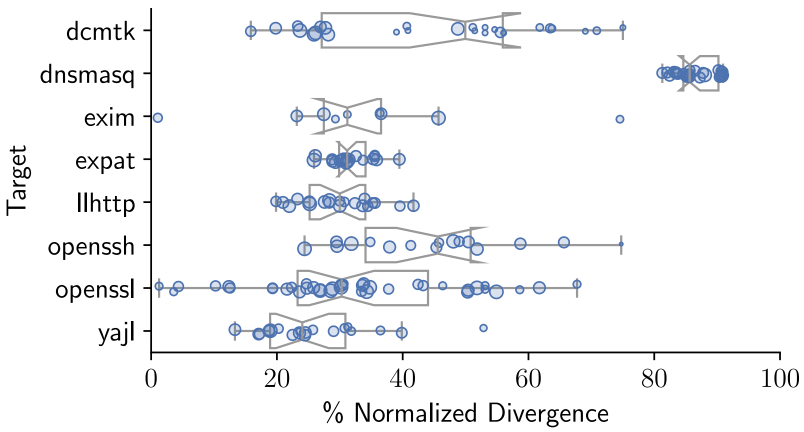


Figure 4.10 – Normalized values of the Kullback-Leibler divergence from uniformity of observed snapshot-to-state distributions, illustrated through notched box-plots at 95% CI. Divergence is calculated individually, for each seed queue, and data points are then overlaid as a ○ scatter plot. The size of each marker is proportional to the number of snapshots N, normalized by the maximum number of snapshots observed for its target across all campaigns.

results in fig. 4.10. A value $\hat{D}_{KL} = 0$ indicates that sampling the seed queue uniformly yields results consistent with sampling a uniformly-distributed population, i.e., where there are equally as many snapshots for every equivalence state discovered by the fuzzer. On the other end of the spectrum, $\hat{D}_{KL} = 1$ implies that uniform sampling yields the highest surprise: whereas the fuzzer would expect to be exploring different functionalities by cycling through its queue, it is likely tunnel-visioned by a majority equivalence state. We observe that most measurements with a significant N possess high divergence from uniformity. A fuzzer that samples its seed queue uniformly would be hindered by duplicate efforts and a self-reinforcing equivalence state.

On the other hand, while fuzzers generally employ more complex seed scheduling mechanisms [22, 94, 145], those do not represent a replacement to state awareness. A schedule that prioritizes snapshots unequally based on observed feedback inherently disregards the possible overlap of those snapshots with others in their equivalence state. Equivalent snapshots exercise overlapping behavior, insofar as cross-testing has not identified discrepancies that necessitate subdividing the group into distinct functionalities. However, since fuzzing is incomplete, it may be that now-equivalent snapshots may diverge in the future, given that they are sufficiently scheduled. A non-uniform scheduling strategy may starve those snapshots of the time needed to explore their potential capabilities, often in favor of the *first* one which uncovered interesting features.

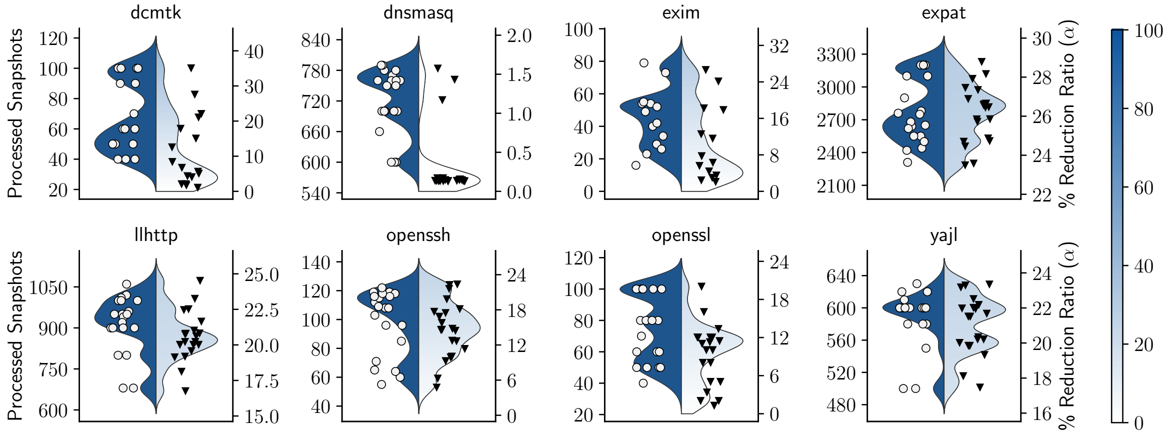


Figure 4.11 – The distributions of discovered snapshots that undergo state inference and the resulting reduction ratio $\alpha = {}^{\text{states}}/_{\text{snapshots}}$. The split kernel density plots display the estimated probability density of the number of processed snapshots (left) and the corresponding reduction ratio (right), whose data points from each campaign are denoted by ○ and ▼ respectively.

4.5.4 Reduction Ratio α

The main advantage of state inference is condensing the seed queue into functionally-distinct islands. This allows seed scheduling to be more balanced and reduces redundant exploration of the same code regions. To assess the effect of state inference on queue size reduction, we measure the number of snapshots generated by every campaign, and the corresponding number of discovered groupings. In fig. 4.11, we report the reduction ratio α as the number of groupings (states) to the number of snapshots.

We observe that `dnsmasq` is again a notable exception: most measurements yield an almost zero reduction ratio. This serves as confirmation that state inference correctly identified the statelessness of the DNS server in most experiments. Otherwise, the average reduction ratio is around 20%, i.e., the queue is five times smaller. Combined with the results from section 4.5.2, this suggests that, during later fuzzing stages, the fuzzer can cycle through its queue five times as fast, at a diminishing cost.

The skewed distribution of seeds in fuzzing queues, combined with the potential for condensing a queue down to 20% its original size, suggests that state-of-the-art fuzzers could benefit from applying state inference to prune their queues and avoid tunnel vision towards high-density regions. Continuous incremental application also ensures that new snapshots are incorporated and that the fuzzer avoids regression towards non-uniformity.

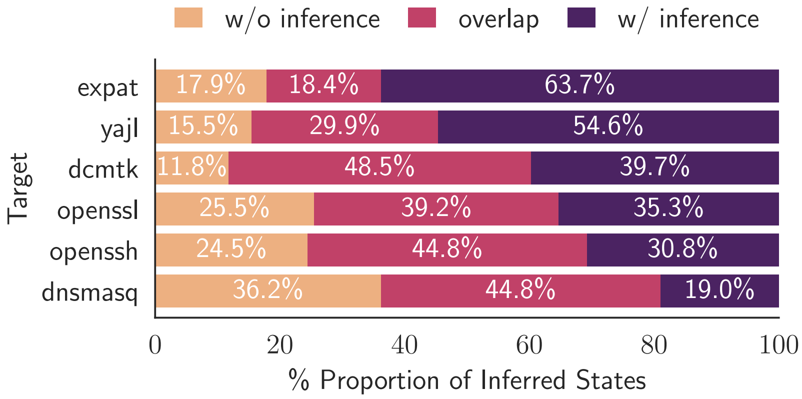


Figure 4.12 – Cross-inference results showing the distribution of overlapping and unique behaviors discovered by stateful fuzzers with and without state inference.

4.5.5 Case Study on Cross-Inference

We implement state inference as a hotpluggable component to introduce state-aware scheduling to two existing fuzzers: AFL++ (for the streaming parsers) and Nyx-Net (for the network servers). The fuzzer and TANGO share one physical core throughout the campaign. TANGO continuously checks for new inputs in the fuzzer's seed queue and applies inference, exporting its results for use by the fuzzer's scheduler. TANGO is also configured to export any interesting inputs it discovers during cross-testing, further reducing its effective overhead.

We measure the code coverage achieved by both the unmodified and the augmented variants of the fuzzers, and we present those in fig. 4.13. Since code coverage alone cannot capture state information, we leverage state inference as a performance metric, through a process we call *cross-inference*. Seeds obtained from two competing fuzzers are used to construct a snapshot tree, upon which state inference is applied. We interpret the overlap and disjunction of those snapshots in equivalence states as a measure of state coverage: states where all snapshots of fuzzer **B** are subsumed by snapshots of fuzzer **A** are considered unique to **A**, without loss of generality. Otherwise, we consider them overlapping. The results of this experiment are illustrated in fig. 4.12.

The experiments yield an interesting result: despite both fuzzers attaining similar code coverage, state inference revealed distinctions in uncovered functionality, favoring the state-guided fuzzers. The additional code covered by the unmodified fuzzers may not directly translate to state coverage: "novel" paths may belong to the same equivalence state. Through our evaluation, state-guided fuzzers uncovered two new bugs: a heap buffer overflow in `dcmtk` and a heap out-of-bounds read in `yajl`.

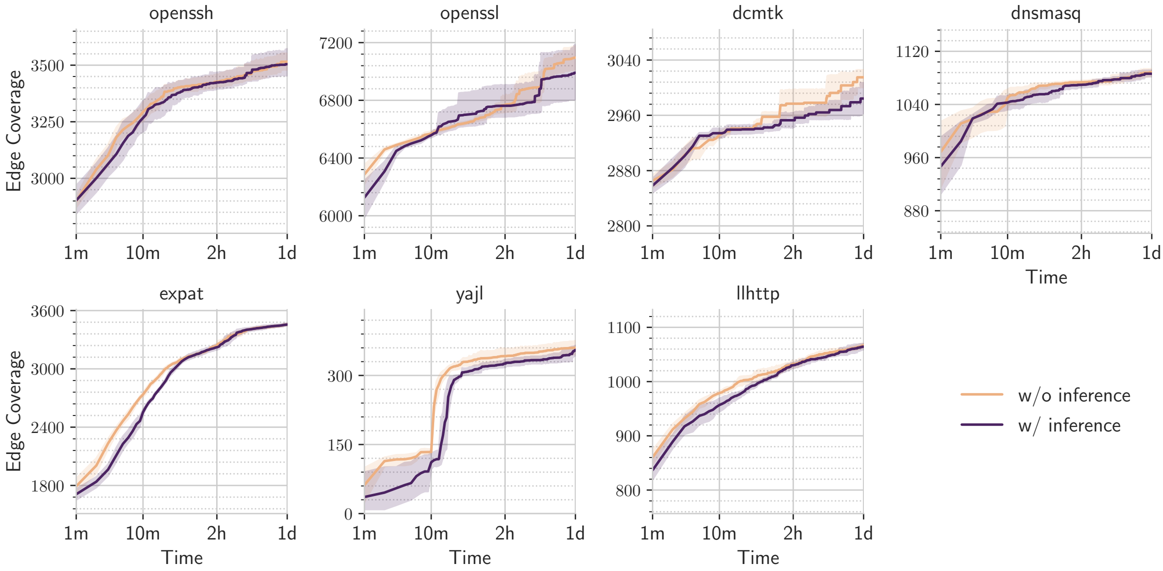


Figure 4.13 – Edge coverage collected from Nyx-Net (for network servers) and AFL++ (for parsers) when running with and without the state inference extension. Plots show the mean coverage of five campaigns, overlaid with the 95% CI.

4.6 Scope

In essence, state inference provides labels for snapshots with distinct behaviors. While it does not come for free, its scope of application, through Tango or otherwise, spans multiple aspects of the fuzzing stack:

Queue distillation: By applying state inference to an existing seed queue, we can identify the smallest subset of seeds which exercises the most unique functionality. At its core, subsumption presents a mechanism for solving the set cover problem, however while preserving cover membership in the form of equivalence states.

Seed minimization: While seeds are typically minimized to achieve the same coverage map, state inference offers an alternative selection criterion: functional equivalence. This offers larger opportunities for minimization, as it expands the search to possibly smaller inputs within the same state.

Out-of-band scheduling: While Tango's flexibility provides a solid framework for prototyping research concepts in stateful fuzzing, it is not optimized for production. However, more mature fuzzers can still benefit by concurrently applying batched inference on their queues, leveraging state awareness while continuing to fuzz at optimal speeds.

State extraction: State inference could also offer a more systematic, less parametric approach to identifying state variables. By processing labeled snapshots, we can search for patterns in memory contents that best fit the provided groupings (i.e., overlaps within states and disjunctions among them).

Performance metric: To assess the performance of a fuzzer relative to a baseline, it is possible to perform cross-inference: test one fuzzer's queue against the other's pre-labeled seeds, in both directions. This approach highlights distinctions in explored behavior by either fuzzer, providing a meaningful measure of performance more suitable for evaluating stateful fuzzers.

4.7 Related Work

State-aware fuzzing: AFLNet [124] was among the first to tackle the problem of fuzzing network targets while allowing state to accumulate. However, its requirement to manually annotate server responses hindered adoption. Alternative techniques presented in SGFuzz [14], NSFuzz [131], and StateAFL [112] addressed this issue through more automated, albeit less precise techniques for state extraction. Nonetheless, those works focused on extracting state, not as a way to generate protocol-compliant inputs, but as a labeling mechanism for discovered inputs. State inference in TANGO makes this mechanism more explicit by exploring functional overlaps.

Seed scheduling: While seed scheduling is a well-researched problem in fuzzing [66, 145, 158], state inference is the first to address it in the context of stateful systems. Existing techniques do not account for persistent effects of executing seeds; they perform their analysis retrospectively. In contrast, state inference runs a prospective analysis, finding overlaps in the traces of inputs executed after the seed instead.

4.8 TANGO Summary

Research on stateful fuzzing continued where its stateless counterpart left off. While much of the progress on the latter was of great benefit to this field, it still managed to imprint methods and assumptions that are otherwise not suited for stateful fuzzing. In this chapter, we re-assess the definition of states and how they fit into the fuzzing stack. We present a method to identify semantic behavior through the use of portable metrics, in a technique we dub "State Inference". In the process, we design and implement TANGO, a state-aware fuzzing framework for bootstrapping research in this domain. Through evaluation, we identify a key observation: fuzzers could potentially spend upwards of 80% of their time being tunnel-visioned or duplicating their efforts. By applying our technique, fuzzers can leverage state awareness for more optimal scheduling, at a diminishing amortized cost. State inference is also applicable in other stages of the fuzzing cycle, from seed minimization and distillation, through unsupervised state extraction, to better-grounded performance evaluation.

Chapter 5

SENSEI: Input Structure-Aware Fuzzing through Selective Feedback

> When all you have is a hammer, everything looks like a nail.
>
> *Abraham Maslow*

Systems of interest are often complex and stateful. To find bugs in them requires well-guided exploration as well as effective input mutation. We aim to tackle the challenges in fuzzing network servers through protocol-specific feedback and parser-guided input generation by leveraging Wireshark dissectors. This chapter serves as an outline for future work in this direction.

Hypothesis 4

Generating high quality inputs is pivotal to a fuzzer's ability to explore a complex system. In the lack of a specification, domain knowledge encoded in consumer programs can be leveraged to guide input generation.

In evolutionary fuzzers, exploration is guided by feedback and executed through input selection and scheduling. Starting from an initial population of seeds, inputs are generated through mutation and cross-over, and filtered based on feedback. This mechanism enables domain-specific exploration without domain-specific knowledge, simply by maximizing an objective function, which is commonly *code coverage*. Despite its simplicity, this genetic algorithm can only be effective if (a) the feedback correlates with the objective, and (b) the mutations drive exploration towards the objective.

The true objective of a fuzzer is to find bugs, but without knowing where the bugs are, it is impossible to define a meaningful objective function. Instead, fuzzers optimize a proxy metric that correlates with the distance to bugs, which is predominantly code coverage: the more code is covered, the closer we are to buggy code and to triggering bugs. But code coverage does not tell the whole story.

Bugs in data

Example 5.1 Consider a fuzzing target which implements the following Python script:

```python
1 # Extract first characters from user's name
2 name = input("What's your name?")
3 try:
4   offset = int(input("Until where would you like to extract?"))
5 except ValueError ex:
6   # if user inputs something other than a number
7   raise RuntimeError("I need a number!") from ex
8 for i in range(offset):
9   print(name[i])
```

A bug exists on line 9, where `i` could have a value outside the bounds of the input `name`.

The developer decides to deploy their script as an executable, so they embed a Python interpreter into their binary. For simplicity, let's assume the interpreter is implemented as:

```c
uint8_t *bytecode = /*...*/;
size_t sz = /*...*/, pc = 0;
while (pc >= 0)
  process_instruction(bytecode, sz, &pc);
```

`process_instruction()` looks up the current instruction in a table of pre-compiled functions and merely calls the function with its arguments.

From the fuzzer's perspective, the code it is fuzzing is that of the interpreter's. Within the first loop, code coverage would essentially be saturated. However, that would translate to executing only the first instruction of the Python script. For the rest of the execution, and in subsequent iterations, the fuzzer would not detect any new coverage. Consequently, its progress is hampered, since maximizing code coverage does not translate to exploring more functionality in the target, where functionality is defined here in the bytecode *data*. The search space to be covered by a fuzzer is vast and limitless, so a guided search is key for effective exploration. If the observed metric was otherwise the value of `pc`, then it would have correlated better with the proxy objective of covering more *Python* code.

The performance of a fuzzer depends on the feedback it observes. But feedback from complex targets can be *noisy*: such targets often implement auxiliary functionalities and error-handling routines which could misguide the fuzzer and drive exploration away from points of interest. Without domain knowledge, it is not possible to distill the metrics of interest from the feedback. Nonetheless, domain knowledge can be readily available, as is the case of network protocol dissectors.

Dissectors, such as those in Wireshark, are information-dense: they encode domain knowledge by mainly implementing the parsing semantics of the protocol. Error-handling is minimal, and inputs are not processed past the point of parsing. Dissectors are also fast, as they run only in user-space. Additionally, they use a unified API for parsing, which is easy to instrument, and the instrumentation would be transferable. Wireshark ships with over a thousand different dissectors, making them a good source of feedback for input generation in fuzzers.

5.1 Background

Fuzzing owes its bug-finding advantage largely to its search mechanisms: genetic algorithms and hill-climbing. Through genetic mutations, a fuzzer ensures a constant stream of diverse, yet well-selected inputs that are likely to discover new behaviors in the target. Hill-climbing then drives the fuzzer in the direction of maximizing a certain objective, which is most commonly code coverage.

The genetic algorithm in fuzzing simply adds a new input to the population of interesting inputs whenever it discovers *any* new behavior, namely in the case of code coverage, whenever, e.g., a new line of code or block of instructions has been executed for the first time. The fuzzer has no insight into what that newly-found code is relevant for, or if it is searching in the "right" direction, i.e., generating inputs that are more likely to trigger bugs. As is often the case, fuzzers get stuck in shallow code regions due to the fact that structured inputs are fragile: a small mutation might invalidate the entire structure, setting the fuzzer back at the early stages of parsing. In that case, a fuzzer is then less likely to be able to break through the parsing component, and is more amenable to exploring other adjacent code regions that cover, e.g., error-handling of invalid inputs. Unable to discriminate, the fuzzer would then inadvertently search in the direction of more invalid inputs to cover more cases in the error-handling code. While there's certainly merit to covering all code—meaning that error-handling code is not guaranteed to be error-free and is possibly worth exploring—it is often the case that we are interested in more complex behaviors in the target, which typically lie past the parsing stage.

Error paths are not the only source of noise in the fuzzer's feedback channel. In the target, components and routines that process the input, prepare the output, or set up internal state, all contribute to the collected feedback. Without being selective about which feedback sources to listen to, or without the ability to assess the relevance of each source to its goal of finding bugs, the fuzzer is overwhelmed with information, which hinders its search progress. Component boundaries are seldom well-defined in software or accessible to the fuzzer. Lacking domain knowledge and manual annotations, it is not possible for the fuzzer to scope out relevant parts from the feedback.

Domain Knowledge

Definition 5.1 Domain knowledge refers to auxiliary information that is not typically encoded in the source code, but is rather available either through written specification or the developer's intent.

While knowledge of component boundaries is not available to the fuzzer, the specification of the input format is directly encoded in the parsing component, and with access to that knowledge (through, e.g., coverage over the parser's code), the fuzzer can generate inputs that better represent the specification.

It follows from these constraints that the most fitting target to fuzz in order to generate valid inputs would be a dedicated parsing component, or in other words, a byte stream dissector. One such source of dissectors is Wireshark, a network protocol analyzer that implements dedicated parsers for message types and binary structures of over 1,500 network protocols and file formats.

In this chapter, we explore the use of dissectors as a selective source of feedback for guiding the exploration of more complex targets.

5.2 Dissector-guided Fuzzing

We propose a design for a fuzzer that leverages dissectors for exploration and mutation. Our design builds on top of two pillars:

Co-evolutionary Exploration To guide the fuzzer in exploring a target, we leverage the dissector as an oracle for the quality of an input before it is sent to the target. While the main goal is to explore and evaluate the target, exploring the dissector provides more direct access to high-quality inputs. Dissectors implement the parsing logic of a protocol, leaving out most of the remaining functionality to the complete implementations. Our design opts for co-evolution of exploration in both the target and the dissector, enabling us to fuzz the dissector *in-breadth*, to cover the different input formats, and the target *in-depth*, to exercise the different functionalities.

Multi-objective Optimization Dissectors are information-rich and follow a unified API for parsing. Knowledge of the parser's state and progress is readily available and provides good metrics for input quality, e.g., parsed fields, enum coverage, input byte coverage, and dissector code coverage, among many others. Since these metrics are often interdependent, the fuzzer would benefit from maximizing them simultaneously. While fuzzers mostly implement single-objective evolution, recent works [107, 161, 175] have explored the incorporation of multi-objective optimization in fuzzers. We leverage it in our design to optimize all the different metrics for the simultaneous exploration of both the target and the fuzzer.

We implement SENSEI as a two-stage fuzzing pipeline: *Specialization* and *Generalization* (later referred to as **S** and **G**, respectively). In the *Specialization* stage, we fuzz a Wireshark dissector, using dissector-specific metrics, namely (i) parse-tree coverage; (ii) input coverage; and (iii) dissector code coverage.

Wireshark dissectors consume a sequence of bytes and dissect it into a *parse tree*. At the root of the tree is the byte sequence itself, the root field, from which sub-fields stem out to form the tree structure. Each *field*, i.e., node, has a certain type (e.g., checksum or bit field) specific to the dissector, and holds a part of the input byte sequence. Parse-tree coverage thus represents the parsed fields and the different nesting levels. Higher parse-tree coverage implies parsing a diverse set of fields at different depths.

An input with high parse-tree coverage could suffer from a large size: to cover many different fields, the fuzzer may search in the direction of larger inputs, possibly inserting redundant or un-parsed bytes in the process. These bytes increase the size of the input and could slow down its transmission to and processing in the target. To coerce the fuzzer to reduce unutilized bytes in the input, we maximize input coverage, representing the ratio of parsed bytes to the total number of bytes in the sequence. Maximizing input coverage alone is also disadvantageous: an input of length one could have 100% input coverage without exploring any interesting fields in the dissector.

Finally, dissector coverage provides an indicator of the different aspects of the input grammar, e.g., different packet types and edge cases, allowing the fuzzer to explore more of the grammar to trigger interesting functionalities in the target. However, as with the other metrics, maximizing code coverage alone does not come without disadvantages: the fuzzer may get tunnel-visioned by one packet type or field type, as it covers more and more edge cases and triggers longer processing paths, without making meaningful progress in exploring the input grammar.

During *Specialization*, we simultaneously optimize all these metrics through multi-objective optimization. We opt for the NSGA-II [38] algorithm, as its mode of operation coincides with our optimization goals: it drives exploration towards Pareto-optimal solutions while ensuring the diversity of selected solutions along the different objectives. The output of this stage is a set of grammar-optimized inputs that serve as high-quality seeds for bootstrapping the next stage, *Generalization*. In the *Generalization* stage, we employ an off-the-shelf fuzzer, compatible with the target, e.g., a stateful fuzzer like Nyx-Net [141]. The output of **S** is continuously synchronized with **G**, as both stages run simultaneously to make progress in both the dissector and the target.

5.3 Preliminary Evaluation and Limitations

We implemented SENSEI's *Specialization* stage as a libFuzzer stub, where we modified the runtime of the fuzzer to incorporate feedback collected from the dissector by hooking its API. Then, we set up the pipeline from **S** to **G** through an inotify-enabled observer that copies the interesting inputs saved in the queue of **S** into a synchronization directory for **G**, applying any necessary transformations in the process (e.g., converting the raw inputs to a structure understood by Nyx-Net).

A preliminary evaluation revealed a major limitation of the two-stage approach: the fuzzer in **G** does not have access to dissector-specific metrics from **S** upon which the interesting-ness of the inputs was decided. As such, using only code coverage to justify whether or not a seed should be imported, **G** ends up discarding most seeds from **S**. We found that, of the hundreds of seeds generated by **S**, around 10–20 were imported into **G** on average, greatly impacting the efficacy of this two-stage architecture.

Furthermore, the lack of synchronization from **G** to **S** could equally impact progress on the dissector. The target fuzzed in **G** is likely to incorporate a more comprehensive parser which encodes finer details of the input grammar as it is often required to be standard-compliant. Dissectors, on the other hand, are implemented with looser requirements, as the specificities of the different fields are not critical to the functional operation of the dissector. This could leave the **S** fuzzer struggling in exploring the dissector, as it tries to make large steps towards the next “best solution”, whereas the **G** fuzzer can explore the input space through more incremental steps. Providing **S** with access to these intermediate solutions from **G** could equally bootstrap exploration of the dissector.

5.4 Future Work

Dissectors encode a plethora of information, of which we covered a part through the metrics we introduced earlier. To make full use of the domain knowledge accessible to the fuzzer, we propose two architectural changes to SENSEI that aim to better incorporate dissector-specific feedback into the fuzzing process.

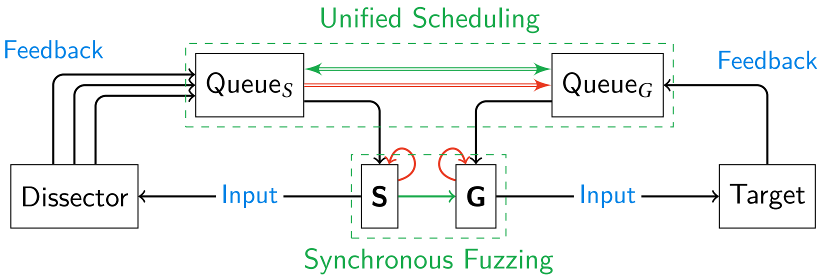


Figure 5.1 – Transforming the two-stage pipeline into a unified architecture. Changes are represented through deletions and additions to the work-flow of the pipeline. Unification achieves synchronization and information sharing between the two stages.

5.4.1 Unified Scheduling

The two-stage pipeline design imposes an inherent limitation: the two fuzzers operate independently without sharing feedback. This drawback impacts the effectiveness of the fuzzing process, since (a) the fuzzers run concurrently, possibly contending on available compute resources; and (b) the first stage poses an overhead with diminishing return, due to the restricted synchronization process.

To address these drawbacks, we propose unifying the two stages into one fuzzing loop, as illustrated in fig. 5.1. In this configuration, the scheduler oversees both the dissector and the target and incorporates all collected feedback into one global multi-objective search over the input space, optimizing for both input quality and exploration of the target. Instead of having the two stages run concurrently and independently, a unified design would run them sequentially, enabling the fuzzer to collect feedback about the same input from both stages. This change also limits the overhead of the first stage to the execution cost of that input, which is typically very small, due to the simplicity and compactness of dissector code.

5.4.2 Type-Aware Mutations

We can extract an additional aspect about parsed fields, which is their shape and type information. By hooking the accessor functions to the input byte sequence, we can intercept the read requests made by the dissector and query the fuzzer to generate data tailored to the field being parsed.

In this approach, the dissector is first invoked to initiate the parsing process. Whenever the input buffer is accessed, parsing is suspended, and the fuzzer receives information about the requested field. By applying type-aware mutations, the fuzzer generates a value for that field and sends it back to the dissector, which resumes parsing and repeats the exchange until termination. This process is further illustrated in fig. 5.2.

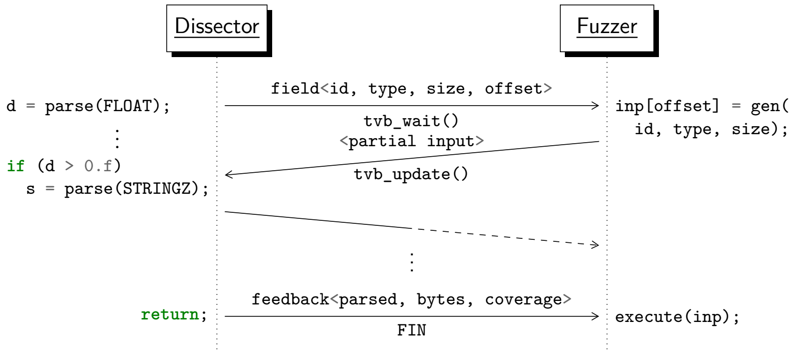


Figure 5.2 – On-demand type-aware input generation.

5.5 Related Work

Multi-objective Optimization State-of-the-art fuzzers continue to use single-objective genetic algorithms. However, multi-objective optimization in fuzzing has been the subject of research in recent years. Co-evolution is the most common approach, where multiple sets of test inputs concurrently evolve through distinct fitness functions. For instance, AFL-HR [107] utilizes a specialized fitness metric, known as headroom, to identify vulnerabilities. It then employs a co-evolutionary algorithm to prioritize test inputs based on both code coverage and headroom metrics. Other metrics like memory usage [164] and execution counts [89] have also been incorporated into co-evolutionary models. FuzzFactory [116] offers a framework for instrumenting and co-evolving various domain-specific goals. The approach adopted by MobFuzz [175] more closely resembles our goals, achieving true simultaneous multi-objective optimization.

Generator and Consumer Guidance The idea of leveraging domain knowledge encoded in parsers like file dissectors, or generators like network clients, is not novel. Namely, FormatFuzzer [43] provides a transpiler that converts dissector specifications, for the 010 Editor commercial software, into a structure-aware generator. Fuzztruction [17] leverages domain knowledge encoded in file generators by modeling those generators themselves as a "structure-aware" mutation function. By injecting faults into the generators themselves, the fuzzer introduces mutations into the input generation routines, yielding semi-valid inputs that probe deeper into the target.

Network Protocol Fuzzing Stateful fuzzing, especially for network protocols, presents many new challenges. State identification and scheduling by itself can be a major hurdle for fuzzers, as we saw in chapter 4. But state scheduling alone is not sufficient, since the fuzzer is still tasked with finding new inputs to explore different states. Previous research in this direction, by Fan et al. [47] and by Zhao et al. [176], focused on learning the message grammars and sequences from recorded traffic. Other research directions [1, 74] tackled grammar extraction from textual specifications for guiding input generation in fuzzers.

To our knowledge, leveraging domain knowledge in Wireshark dissectors, combined with multi-objective optimization, is a novel research area, yet it holds merit, based on the results of previous incremental work in this direction.

5.6 Upcoming Challenges

In implementing and evaluating this design, it is inevitable that obstacles will be encountered:

Instrumentation Although the dissectors follow a unified API provided by Wireshark, the set of provided functions is large and complex. To correctly hook each function and collect the relevant parameters from it may require a great deal of manual effort. Nonetheless, it would be a one-time cost, since the instrumentation is then transferable across all dissectors. Another hurdle in instrumentation could be the unavailability of some field metadata at the time the data is requested through the TVB, i.e., the function call does not provide all the information available to the parser. To address that, it might be necessary to perform data-flow tracking within the dissector, in the vicinity of TVB-accessors, to identify sources and sinks and map the requested data to the field it will eventually populate. An alternative approach would be to "speculatively execute" the dissector past the data request—by giving it dummy bytes—and observe which fields it populates next, ensuring that the dummy data matches the data placed in the field.

Scheduling In employing multi-objective optimization, the fuzzer would have to balance its exploration and exploitation efforts across all optimized metrics. This would require a careful formulation of the power schedules, targeting high-value inputs while avoiding tunnel vision by diversifying the exploration of less optimal inputs. It's important to note that dissectors and targets may respond very differently across the observed metrics, and that the choice of the optimization algorithm should be made to ensure adaptivity of the fuzzer under different configurations. For instance, following the naïve approach of assigning static weights to different metrics could bias the fuzzer towards some targets that respond well to maximizing a certain metric, while hindering its ability to effectively explore other targets where that metric is of much less significance.

Statefulness The challenges arising from stateful fuzzing compound on top of the ones mentioned here so far. Although Wireshark dissectors are written for inspecting and analyzing the traffic from network protocols, they mainly target the parsing of protocol messages, encoding little information about the protocol state or expected behavior. Some dissectors provide *expert info* to encode additional information, such as the validity of a TCP segment sequence number or the lack of a response for an ICMP request. However, this interface is mostly ad hoc and does not follow a specific format from which data can be directly extracted. As such, extracting state information from dissectors may not be possible, or at least not reliably.

Chapter 6

Summary

Fuzzing's utility has been well-recognized for automating test case generation; however, its efficacy is often impeded by the limitations of the underlying progress and performance metrics. To that end, this work undertakes three projects—MAGMA, IGOR, and TANGO—that offer targeted advancements in metrics used for fuzzer evaluation and development.

MAGMA addresses the lack of standard benchmarks for fuzzer evaluation by introducing an objective assessment platform based in ground-truth. Using real-world bugs and carefully-designed metrics, MAGMA reveals the shortcomings of relying solely on crash counts or code coverage for fuzzer performance comparison.

IGOR focuses on refining crash de-duplication techniques. By employing a novel metric based on control-flow graph similarity comparisons over minimized execution traces, IGOR demonstrates its effectiveness in reducing inflated unique bug counts and distilling bug reports, thereby enabling faster development and triaging cycles.

TANGO delves into the exploration of state spaces in complex systems. By incorporating state as an explicit metric, TANGO enhances a fuzzer's ability to navigate stateful systems. This effort also reveals that traditional code coverage metrics are inadequate for dealing with stateful systems, often causing fuzzers to be inefficient in scheduling their seed queues.

Collectively, these projects underscore the importance of employing intricate metrics that go beyond crash counts and code coverage. They offer empirical proof that optimizing for such metrics results in more effective fuzzing across a spectrum of software systems. Advanced metrics, capturing deeper aspects of state and semantics, are thus pivotal for enhancing the efficacy of fuzzing techniques in increasingly complex software landscapes.

Chapter 7

Closing Thoughts

Over the past decade, research on fuzzing has witnessed a shift in philosophies:

- Fuzzing is a random walk → Fuzzing is a guided search
- Targets are programs → Targets are systems
- Inputs are bytes → Inputs are data

What started out as random bit flips in CLI inputs later evolved into data-type- and system-type-agnostic testing. Nowadays, fuzzers are used to assess the security of systems spanning autonomous cars, smart contracts, embedded systems, video games, you name it. The complexity of the systems that fuzzers are exposed to has skyrocketed, yet, we continue to rely on simplistic feedback that stands no chance for capturing the intricacies and complexities of those systems. In short, we are asking too much of current fuzzers.

The contemporary challenges to fuzzing, as outlined in this dissertation, reveal a fundamental weakness in our approach to fuzzing: we assume fuzzers are intelligent, when in fact, they are not.

In detecting bugs, sanitizers provide a good indicator for violations of language semantics. Countless memory corruption bugs have been found and fixed as a result of sanitization, leading to the conclusion that memory safety bugs are the most abundant. But that begs the question, whether the real distribution of bugs is actually shifted towards memory safety, or whether our observations are biased by the lack of tooling to identify other bug types. There is certainly no shortage of semantic and logic bugs, as evident in recent years with exploits like that in log4j and side-channel attacks in CPUs wreaking havoc at a global scale. It is important to reiterate the nature of bugs: they are deviations from expected behavior. Without setting proper expectations, classes of bugs beyond memory and type safety will continue to elude us. We need smarter decision-making, and more informed fuzzers.

If we want to target complex systems in reasonable time, some "intelligent" decision-making ought to be incorporated into fault detection, exploration strategies, and distilling feedback. That could come either from human-in-the-loop design, or outsourced to the rather advanced ML tools that we have nowadays, and which continue to prove their impact in real-world applications.

Appendices

Appendix A

MAGMA Bugs and Reports

Table A.1 – The bugs injected into MAGMA, and the original bug reports. Of the 118 bugs, 78 bugs (66%) have a scope measure of one. Although most single-scope bugs can be ported with an automatic technique, relying on such a technique would produce fewer and lower-quality canaries. PoVs of (∗)-marked bugs are sourced from bug reports.

Bug ID	Report	Class	PoV	Scopes
AAH001	CVE-2018-13785	Integer overflow, divide by zero	✓	1
AAH002*	CVE-2019-7317	Use-after-free	✓	4
AAH003	CVE-2015-8472	API inconsistency	✓	2
AAH004	CVE-2015-0973	Integer overflow	✗	1
AAH005*	CVE-2014-9495	Integer overflow, Buffer overflow	✓	1
AAH007	(Unspecified)	Memory leak	✓	2
AAH008	CVE-2013-6954	0-pointer dereference	✓	2
AAH009	CVE-2016-9535	Heap buffer overflow	✓	1
AAH010	CVE-2016-5314	Heap buffer overflow	✓	1
AAH011	CVE-2016-10266	Divide by zero	✗	2
AAH012	CVE-2016-10267	Divide by zero	✗	1
AAH013	CVE-2016-10269	OOB read	✓	1
AAH014	CVE-2016-10269	OOB read	✓	1
AAH015	CVE-2016-10270	OOB read	✓	4
AAH016	CVE-2015-8784	Heap buffer overflow	✓	1
AAH017	CVE-2019-7663	0-pointer dereference	✓	1
AAH018*	CVE-2018-8905	Heap buffer underflow	✓	1
AAH019	CVE-2018-7456	OOB read	✗	1
AAH020	CVE-2016-3658	Heap buffer overflow	✓	2
AAH021	CVE-2018-18557	OOB write	✗	2
AAH022	CVE-2017-11613	Resource Exhaustion	✓	2
AAH024	CVE-2017-9047	Stack buffer overflow	✓	2
AAH025	CVE-2017-0663	Type confusion	✓	1
AAH026	CVE-2017-7375	XML external entity	✓	1
AAH027	CVE-2018-14567	Resource exhaustion	✗	1
AAH028	CVE-2017-5130	Integer overflow, heap corruption	✗	1
AAH029	CVE-2017-9048	Stack buffer overflow	✓	2
AAH030	CVE-2017-8872	OOB read	✗	2
AAH031	ISSUE #58 (gitlab)	OOB read	✗	1
AAH032	CVE-2015-8317	OOB read	✓	2
AAH033	CVE-2016-4449	XML external entity	✗	1
AAH034	CVE-2016-1834	Heap buffer overflow	✗	2
AAH035	CVE-2016-1836	Use-after-free	✓	2
AAH036	CVE-2016-1837	Use-after-free	✗	1
AAH037	CVE-2016-1838	Heap buffer overread	✓	2
AAH038	CVE-2016-1839	Heap buffer overread	✗	1
AAH039	BUG 758518	Heap buffer overread	✗	1
AAH040	CVE-2016-1840	Heap buffer overflow	✗	1
AAH041	CVE-2016-1762	Heap buffer overread	✓	1
AAH042	CVE-2019-14494	Divide-by-zero	✓	1
AAH043	CVE-2019-9959	Resource exhaustion (memory)	✓	1
AAH045	CVE-2017-9865	Stack buffer overflow	✓	4
AAH046	CVE-2019-10873	0-pointer dereference	✓	2
AAH047*	CVE-2019-12293	Heap buffer overread	✓	1
AAH048	CVE-2019-10872	Heap buffer overflow	✓	3
AAH049	CVE-2019-9200	Heap buffer underwrite	✓	1
AAH050	Bug #106061	Divide-by-zero	✓	1
AAH051*	ossfuzz/8499	Integer overflow	✓	1
AAH052	Bug #101366	0-pointer dereference	✓	1
JCH201	CVE-2019-7310	Heap buffer overflow	✓	1
JCH202	CVE-2018-21009	Integer overflow	✗	1
JCH203	CVE-2018-20650	Type confusion	✗	2
JCH204	CVE-2018-20481	0-pointer dereference	✗	1
JCH206	CVE-2018-19058	Type confusion	✗	2
JCH207	CVE-2018-13988	OOB read	✓	1
JCH208	CVE-2019-12360	Stack buffer overflow	✗	1
JCH209	CVE-2018-10768	0-pointer dereference	✓	1
JCH210	CVE-2017-9776	Integer overflow	✓	1
JCH211	CVE-2017-18267	Resource exhaustion (CPU)	✗	1
JCH212	CVE-2017-14617	Divide-by-zero	✓	1
JCH214	CVE-2019-12493	Stack buffer overread	✗	3
AAH054	CVE-2016-2842	OOB write	✗	5
AAH055	CVE-2016-2108	OOB read	✓	5
AAH056	CVE-2016-6309	Use-after-free	✓	1
AAH057	CVE-2016-2109	Resource exhaustion (memory)	✗	2
AAH058	CVE-2016-2176	Stack buffer overread	✗	2
AAH059	CVE-2016-6304	Resource exhaustion (memory)	✗	3
MAE100	CVE-2016-2105	Integer overflow	✗	1
MAE102	CVE-2016-6303	Integer overflow	✗	1
MAE103	CVE-2017-3730	0-pointer dereference	✗	1
MAE104	CVE-2017-3735	OOB read	✓	1
MAE105	CVE-2016-0797	Integer overflow	✗	2
MAE106	CVE-2015-1790	0-pointer dereference	✗	2
MAE107	CVE-2015-0288	0-pointer dereference	✗	1
MAE108	CVE-2015-0208	0-pointer dereference	✗	1
MAE109	CVE-2015-0286	Type confusion	✗	1
MAE110	CVE-2015-0289	0-pointer dereference	✗	1
MAE111	CVE-2015-1788	Resource exhaustion (CPU)	✗	1
MAE112	CVE-2016-7052	0-pointer dereference	✗	1
MAE113	CVE-2016-6308	Resource exhaustion (memory)	✗	2
MAE114	CVE-2016-6305	Resource exhaustion (CPU)	✗	1
MAE115	CVE-2016-6302	OOB read	✓	1
JCH214	CVE-2019-9936	Heap buffer overflow	✗	1
JCH215	CVE-2019-20218	Stack buffer overread	✓	1
JCH216	CVE-2019-19923	0-pointer dereference	✓	1
JCH217	CVE-2019-19959	OOB read	✗	1
JCH218	CVE-2019-19925	0-pointer dereference	✗	1
JCH219	CVE-2019-19244	OOB read	✗	2
JCH220	CVE-2018-8740	0-pointer dereference	✗	1
JCH221	CVE-2017-15286	0-pointer dereference	✗	1
JCH222	CVE-2017-2520	Heap buffer overflow	✗	2
JCH223	CVE-2017-2518	Use-after-free	✓	1
JCH225	CVE-2017-10989	Heap buffer overflow	✗	1
JCH226	CVE-2019-19646	Logical error	✓	2
JCH227	CVE-2013-7443	Heap buffer overflow	✓	1
JCH228	CVE-2019-19926	Logical error	✓	1
JCH229	CVE-2019-19317	Resource exhaustion (memory)	✓	1
JCH230	CVE-2015-3415	Double-free	✗	1
JCH231	CVE-2020-9327	0-pointer dereference	✗	3
JCH232	CVE-2015-3414	Uninitialized memory access	✓	1
JCH233	CVE-2015-3416	Stack buffer overflow	✗	1
JCH234	CVE-2019-19880	0-pointer dereference	✗	1
MAE002	CVE-2019-9021	Heap buffer overread	✗	1
MAE004	CVE-2019-9641	Uninitialized memory access	✗	1
MAE006	CVE-2019-11041	OOB read	✗	1
MAE008	CVE-2019-11034	OOB read	✓	1
MAE009	CVE-2019-11039	OOB read	✗	1
MAE010	CVE-2019-11040	Heap buffer overflow	✗	1
MAE011	CVE-2018-20783	OOB read	✗	3
MAE012	CVE-2019-9022	OOB read	✗	2
MAE014	CVE-2019-9638	Uninitialized memory access	✓	1
MAE015	CVE-2019-9640	OOB read	✗	2
MAE016	CVE-2018-14883	Heap buffer overread	✓	2
MAE017	CVE-2018-7584	Stack buffer underread	✗	1
MAE018	CVE-2017-11362	Stack buffer overflow	✗	1
MAE019	CVE-2014-9912	OOB write	✗	1
MAE020	CVE-2016-10159	Integer overflow	✗	2
MAE021	CVE-2016-7414	OOB read	✗	2

Table A.2 – Mean bug survival times—both **R**eached and **T**riggered—over a 24-hour period, in **s**econds, **m**inutes, and **h**ours. Bugs are sorted by “difficulty” (mean times). The best performing fuzzer is highlighted in green (ties are not included).

	moptafl		honggfuzz		afl++		afl		aflfast		fairfuzz		symccafl		**Mean**	
Bug ID	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
AAH037	10.00s	20.00s	10.00s	10.00s	10.00s	45.50s	5.00s	15.00s	5.00s	15.00s	5.00s	15.00s	10.00s	25.50s	7.86s	20.86s
AAH041	15.00s	21.00s	10.00s	10.00s	15.00s	48.00s	10.00s	15.00s	10.00s	15.00s	10.00s	15.00s	15.00s	30.00s	12.14s	22.00s
AAH003	10.00s	16.00s	10.00s	11.00s	10.00s	15.00s	5.00s	10.00s	5.00s	10.00s	5.00s	10.00s	10.00s	1.58m	7.86s	23.86s
JCH207	10.00s	1.12m	5.00s	1.57m	10.00s	1.94m	5.00s	2.05m	5.00s	1.60m	5.00s	1.42m	10.00s	1.62m	7.14s	1.62m
AAH056	15.00s	14.57m	10.00s	14.43m	15.00s	19.49m	10.00s	13.07m	10.00s	11.27m	10.00s	8.17m	15.00s	17.80m	12.14s	14.11m
AAH015	32.50s	1.57m	10.00s	13.50s	27.00s	17.50m	1.18m	34.59m	52.00s	10.84m	1.07m	10.86m	15.07m	1.02h	2.76m	19.55m
AAH055	15.00s	40.86m	10.00s	2.71m	15.00s	3.62h	10.00s	25.01m	10.00s	2.24h	10.00s	6.36h	15.00s	2.44h	12.14s	2.26h
AAH020	5.00s	2.32h	5.00s	2.12h	5.00s	31.62m	5.00s	2.01h	5.00s	55.17m	5.00s	49.92m	5.00s	11.22h	5.00s	2.85h
MAE016	10.00s	1.57m	5.00s	10.00s	10.00s	5.79m	5.00s	3.97m	5.00s	4.93m	5.00s	2.21h	24.00h	24.00h	3.43h	3.78h
AAH052	15.00s	3.17m	15.00s	14.10m	15.00s	45.03m	10.00s	3.94h	10.00s	10.56h	10.00s	12.02h	15.00s	5.28m	12.86s	3.95h
AAH032	15.00s	3.38m	5.00s	2.06m	15.00s	1.65h	10.00s	3.22h	10.00s	34.19m	10.00s	9.67h	15.00s	12.95h	11.43s	4.02h
MAE008	15.00s	1.42h	10.00s	9.73h	15.00s	1.44m	10.00s	1.14m	10.00s	1.54m	10.00s	12.08h	24.00h	24.00h	3.43h	6.76h
AAH022	32.50s	54.98m	10.00s	34.86m	27.00s	3.47h	1.18m	9.38h	52.00s	5.66h	1.07m	14.04h	15.07m	15.25h	2.76m	7.04h
MAE014	15.00s	1.11h	10.00s	4.11h	15.00s	14.52m	10.00s	5.58m	10.00s	8.28m	10.00s	21.83h	24.00h	24.00h	3.43h	7.36h
JCH215	2.14m	3.24h	15.00s	40.97m	22.30m	11.97h	2.37h	15.67h	48.87m	11.51h	3.23h	9.86h	1.85h	18.08h	1.24h	10.15h
AAH017	5.19h	5.20h	22.32h	22.32h	13.97h	13.97h	19.84h	19.84h	8.67h	9.20h	5.92h	5.92h	9.92h	9.92h	12.26h	12.34h
JCH232	4.87h	4.87h	43.86m	1.66h	14.87h	20.02h	19.82h	19.82h	14.93h	17.21h	6.23h	10.31h	21.81h	21.81h	11.89h	13.67h
AAH014	12.48h	12.48h	24.00h	24.00h	13.06h	13.06h	6.34h	6.34h	24.00h	24.00h	18.46h	18.46h	10.68h	10.68h	15.57h	15.57h
JCH201	15.00s	14.65h	10.00s	24.00h	15.00s	19.48h	10.00s	16.82h	10.00s	12.98h	10.00s	14.02h	15.00s	14.27h	12.14s	16.60h
AAH007	15.00s	24.00h	5.00s	57.00s	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	23.12m	11.43s	17.20h
AAH008	15.00s	16.51h	10.00s	3.65h	15.00s	23.40h	10.00s	19.44h	10.00s	19.66h	10.00s	15.28h	15.00s	23.43h	12.14s	17.34h
AAH045	20.00s	3.33h	13.50s	1.13h	20.00s	24.00h	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	20.00s	24.00h	16.93s	17.78h
AAH013	24.00h	24.00h	4.05h	4.05h	13.88h	13.88h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	19.70h	19.70h
AAH024	15.00s	9.05h	10.00s	9.27h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	19.76h
JCH209	14.40m	14.41m	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	20.61h	20.61h
MAE115	15.00s	22.64h	10.00s	20.96h	15.00s	24.00h	10.00s	21.32h	10.00s	23.33h	10.00s	21.97h	15.00s	10.13h	12.14s	20.62h
AAH026	15.00s	20.88h	10.00s	7.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	21.13h
AAH001	15.00s	22.57h	10.00s	17.70h	15.00s	24.00h	10.00s	22.60h	10.00s	24.00h	10.00s	24.00h	15.00s	14.58h	12.14s	21.35h
MAE104	15.00s	15.53h	10.00s	24.00h	15.00s	24.00h	10.00s	21.81h	10.00s	17.60h	10.00s	24.00h	24.00h	24.00h	3.43h	21.56h
AAH010	21.35h	21.97h	12.53h	16.40h	14.59m	20.34h	10.18m	24.00h	24.00h	24.00h	13.81h	21.79h	4.76h	24.00h	10.98h	21.79h
AAH016	18.68h	19.66h	24.00h	24.00h	22.59h	22.59h	24.00h	24.00h	17.61h	19.83h	19.89h	19.97h	24.00h	24.00h	21.54h	22.01h
JCH226	23.20h	23.72h	4.09h	10.93h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	21.04h	22.09h
JCH228	12.33h	18.10h	2.47h	20.05h	22.07h	24.00h	22.57h	22.60h	24.00h	24.00h	18.78h	24.00h	22.66h	23.80h	17.84h	22.36h
AAH035	15.00s	19.34h	10.00s	24.00h	21.50s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	19.00s	24.00h	13.64s	23.33h
JCH212	15.00s	24.00h	10.00s	20.42h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	23.49h
AAH025	22.22h	22.22h	22.48h	22.48h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	23.53h	23.53h
AAH053	24.00h	24.00h	35.00s	21.80h	24.00h	24.00h	30.00s	24.00h	29.50s	24.00h	26.00s	24.00h	24.00h	24.00h	10.29h	23.69h
AAH042	39.50s	21.93h	20.00s	24.00h	39.50s	24.00h	40.00s	24.00h	34.50s	24.00h	31.00s	24.00h	45.00s	24.00h	35.64s	23.70h
AAH048	15.00s	24.00h	10.00s	22.72h	16.50s	24.00h	15.00s	24.00h	10.50s	24.00h	10.00s	24.00h	20.00s	24.00h	13.86s	23.82h
AAH049	15.00s	22.82h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	23.83h
AAH043	25.00s	22.91h	16.80h	24.00h	2.41h	24.00h	25.00s	24.00h	20.00s	24.00h	20.00s	24.00h	25.00s	24.00h	2.75h	23.84h
JCH210	30.00s	23.07h	20.00s	24.00h	33.00s	24.00h	30.00s	24.00h	25.00s	24.00h	25.00s	24.00h	32.50s	24.00h	27.93s	23.87h
AAH050	25.00s	24.00h	16.80h	23.71h	29.00s	24.00h	24.00h	24.00h	20.00s	24.00h	20.00s	24.00h	29.00s	24.00h	5.83h	23.96h

Table A.2 – Mean bug survival times (cont.). None of these bugs were triggered by the seven evaluated fuzzers.

	moptafl		honggfuzz		afl++		afl		aflfast		fairfuzz		symccafl		**Mean**	
Bug ID	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
AAH054	10.00s	24.00h	5.00s	24.00h	10.00s	24.00h	5.00s	24.00h	5.00s	24.00h	5.00s	24.00h	10.00s	24.00h	7.14s	24.00h
MAE105	10.00s	24.00h	5.00s	24.00h	10.00s	24.00h	5.00s	24.00h	5.00s	24.00h	5.00s	24.00h	10.00s	24.00h	7.14s	24.00h
AAH011	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h
AAH005	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	24.00h
JCH202	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	24.00h
MAE114	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	24.00h
AAH029	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	24.00h
AAH034	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.14s	24.00h
AAH004	16.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	15.00s	24.00h	12.29s	24.00h
MAE111	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	20.00s	24.00h	12.86s	24.00h
AAH059	20.00s	24.00h	10.00s	24.00h	17.00s	24.00h	15.00s	24.00h	15.00s	24.00h	10.00s	24.00h	20.00s	24.00h	15.29s	24.00h
JCH204	18.00s	24.00h	20.00s	24.00h	15.50s	24.00h	15.00s	24.00h	15.00s	24.00h	10.00s	24.00h	20.00s	24.00h	16.21s	24.00h
AAH031	20.00s	24.00h	15.00s	24.00h	42.00s	24.00h	15.00s	24.00h	15.00s	24.00h	15.00s	24.00h	25.00s	24.00h	21.00s	24.00h
AAH051	25.00s	24.00h	10.00s	24.00h	42.50s	24.00h	20.00s	24.00h	20.00s	24.00h	20.00s	24.00h	30.00s	24.00h	23.93s	24.00h
MAE103	33.00s	24.00h	28.00s	24.00h	33.00s	24.00h	27.50s	24.00h	25.00s	24.00h	20.00s	24.00h	31.00s	24.00h	28.21s	24.00h
JCH214	33.50s	24.00h	45.00s	24.00h	36.00s	24.00h	31.00s	24.00h	26.50s	24.00h	25.00s	24.00h	35.00s	24.00h	33.14s	24.00h
JCH220	4.38m	24.00h	11.50s	24.00h	22.04m	24.00h	2.09h	24.00h	54.77m	24.00h	3.12h	24.00h	2.28h	24.00h	1.26h	24.00h
JCH229	4.53m	24.00h	16.00s	24.00h	24.62m	24.00h	2.80h	24.00h	1.07h	24.00h	3.23h	24.00h	2.32h	24.00h	1.42h	24.00h
AAH018	41.88m	24.00h	4.00m	24.00h	5.77h	24.00h	3.17h	24.00h	59.96m	24.00h	36.01m	24.00h	1.85h	24.00h	1.88h	24.00h
JCH230	4.02m	24.00h	22.50s	24.00h	1.07h	24.00h	3.31h	24.00h	1.36h	24.00h	3.56h	24.00h	5.57h	24.00h	2.13h	24.00h
AAH047	25.00s	24.00h	16.80h	24.00h	2.41h	24.00h	25.00s	24.00h	20.00s	24.00h	20.00s	24.00h	25.00s	24.00h	2.75h	24.00h
JCH233	8.31m	24.00h	12.02m	24.00h	6.16h	24.00h	3.87h	24.00h	1.98h	24.00h	3.59h	24.00h	5.17h	24.00h	3.02h	24.00h
JCH223	16.59m	24.00h	30.50s	24.00h	1.19h	24.00h	3.89h	24.00h	1.33h	24.00h	4.03h	24.00h	10.60h	24.00h	3.05h	24.00h
JCH231	21.88m	24.00h	36.00s	24.00h	2.44h	24.00h	3.96h	24.00h	1.41h	24.00h	4.05h	24.00h	10.62h	24.00h	3.27h	24.00h
MAE006	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	24.00h	24.00h	3.43h	24.00h
MAE004	15.00s	24.00h	10.00s	24.00h	15.00s	24.00h	10.00s	24.00h	10.00s	24.00h	10.00s	24.00h	24.00h	24.00h	3.43h	24.00h
JCH222	1.75h	24.00h	21.97m	24.00h	18.91h	24.00h	15.17h	24.00h	13.39h	24.00h	18.87h	24.00h	20.82h	24.00h	12.75h	24.00h
AAH009	14.61h	24.00h	20.62h	24.00h	24.00h	24.00h	5.67h	24.00h	19.45h	24.00h	17.62h	24.00h	23.42h	24.00h	17.91h	24.00h
JCH227	24.00h	24.00h	20.58h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	23.51h	24.00h
JCH219	23.41h	24.00h	23.22h	24.00h	24.00h	24.00h	24.00h	24.00h	23.79h	24.00h	24.00h	24.00h	24.00h	24.00h	23.77h	24.00h
JCH216	23.48h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	24.00h	23.93h	24.00h

Table A.3 – Mean bug survival times over a 7-day period.

	moptafl		honggfuzz		afl++		afl		aflfast		fairfuzz		symccafl		**Mean**	
Bug ID	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
AAH037	10.00s	20.00s	15.00s	15.00s	10.00s	45.50s	10.00s	20.00s	10.00s	20.00s	10.00s	21.00s	10.00s	25.50s	10.71s	23.86s
AAH041	15.00s	21.00s	15.00s	15.00s	15.00s	48.00s	15.00s	20.50s	15.00s	20.00s	15.00s	21.00s	15.00s	30.00s	15.00s	25.07s
AAH003	10.00s	16.00s	15.00s	17.00s	10.00s	15.00s	10.00s	15.00s	10.00s	15.00s	10.00s	15.00s	10.00s	1.58m	10.71s	26.86s
JCH207	10.00s	1.12m	10.00s	2.16m	10.00s	1.94m	10.00s	2.02m	10.00s	3.73m	10.00s	2.96m	10.00s	1.62m	10.00s	2.22m
AAH056	15.00s	14.57m	15.00s	19.65m	15.00s	19.49m	15.00s	16.75m	15.00s	14.69m	15.00s	11.16m	15.00s	17.80m	15.00s	16.30m
AAH015	32.50s	1.57m	15.00s	21.50s	27.00s	17.50m	1.40m	59.27m	1.12m	8.00m	1.23m	13.34m	15.07m	1.02h	2.87m	23.04m
AAH020	5.00s	2.32h	5.00s	2.37h	5.00s	31.62m	5.00s	2.40h	5.00s	49.68m	5.00s	49.06m	5.00s	11.22h	5.00s	2.93h
AAH052	15.00s	3.17m	18.00s	15.09m	15.00s	45.03m	15.00s	3.83h	15.00s	6.17h	15.00s	13.78h	15.00s	5.28m	15.43s	3.56h
AAH022	32.50s	54.98m	15.00s	19.83m	27.00s	3.47h	1.40m	19.31h	1.12m	3.54h	1.23m	11.50h	15.07m	15.63h	2.87m	7.81h
AAH055	15.00s	40.86m	15.00s	4.07m	15.00s	3.62h	15.00s	4.17h	15.00s	1.74h	15.00s	71.84h	15.00s	2.44h	15.00s	12.08h
AAH017	13.22h	13.23h	66.80h	66.84h	13.97h	13.97h	13.88h	14.38h	6.78h	6.78h	3.50h	3.53h	9.92h	9.92h	18.30h	18.38h
AAH032	15.00s	3.38m	10.00s	2.70m	15.00s	1.65h	15.00s	51.23h	15.00s	15.81m	15.00s	67.23h	15.00s	36.05h	14.29s	22.36h
MAE016	10.00s	1.57m	10.00s	15.00s	10.00s	5.79m	10.00s	2.38m	10.00s	6.25m	10.00s	3.13h	168.00h	168.00h	24.00h	24.49h
JCH215	2.14m	3.24h	23.50s	2.42h	22.30m	13.00h	1.87h	45.83h	21.33m	15.91h	48.42m	38.01h	1.85h	85.15h	45.45m	29.08h
MAE008	15.00s	1.42h	15.00s	14.11h	15.00s	1.44m	15.00s	3.87m	15.00s	2.25m	15.00s	33.70h	168.00h	168.00h	24.00h	31.05h
JCH201	15.00s	17.54h	15.00s	140.14h	15.00s	20.53h	15.00s	11.25h	15.00s	13.41h	15.00s	13.95h	15.00s	14.27h	15.00s	33.01h
MAE014	15.00s	1.11h	15.00s	55.08m	15.00s	14.52m	15.00s	4.37m	15.00s	10.03m	15.00s	154.13h	168.00h	168.00h	24.00h	46.38h
AAH014	14.52h	14.52h	122.24h	122.24h	13.06h	13.06h	18.54h	18.54h	143.74h	143.74h	75.78h	75.78h	38.20h	38.20h	60.87h	60.87h
JCH232	4.87h	4.87h	44.67m	2.35h	26.08h	48.03h	83.73h	117.35h	31.74h	50.30h	34.90h	101.98h	105.04h	117.65h	41.01h	63.22h
MAE115	15.00s	61.48h	15.00s	109.18h	15.00s	133.93h	15.00s	47.46h	15.00s	32.83h	15.00s	94.71h	15.00s	10.13h	15.00s	69.96h
AAH008	15.00s	32.45h	15.00s	3.39h	15.00s	141.24h	15.00s	25.27h	15.00s	126.02h	15.00s	117.94h	15.00s	55.21h	15.00s	71.65h
JCH209	14.40m	14.41m	168.00h	168.00h	63.14h	63.14h	62.32h	62.33h	44.40h	44.41h	154.76h	154.76h	49.70h	49.72h	77.51h	77.51h
MAE104	15.00s	57.64h	15.00s	168.00h	15.00s	109.74h	15.00s	114.85h	15.00s	51.48h	15.00s	40.27h	168.00h	168.00h	24.00h	101.43h
AAH010	41.60h	44.03h	22.39h	42.87h	14.59m	121.14h	14.80m	168.00h	139.00h	150.52h	39.15h	136.06h	19.16h	65.62h	37.40h	104.04h
JCH228	16.37h	35.61h	6.97h	60.72h	94.73h	117.15h	128.84h	153.74h	58.00h	111.25h	104.22h	126.98h	129.87h	146.81h	77.00h	107.47h
AAH007	15.00s	168.00h	10.00s	1.56m	15.00s	167.02h	15.00s	163.55h	15.00s	168.00h	15.00s	168.00h	15.00s	23.12m	14.29s	119.28h
AAH045	20.00s	3.33h	20.00s	21.44m	20.00s	168.00h	20.00s	168.00h	20.00s	164.38h	20.00s	168.00h	20.00s	164.86h	20.00s	119.56h
AAH013	168.00h	168.00h	2.40h	2.40h	13.88h	13.88h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	122.33h	122.33h
AAH016	49.04h	50.76h	140.25h	141.75h	44.30h	137.79h	81.63h	146.06h	110.77h	125.20h	120.45h	120.48h	154.66h	155.00h	100.16h	125.29h
AAH001	15.00s	152.17h	15.00s	12.17h	15.00s	168.00h	15.00s	144.94h	15.00s	168.00h	15.00s	168.00h	15.00s	76.79h	15.00s	127.15h
AAH024	15.00s	9.05h	15.00s	52.37h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	128.77h
AAH026	15.00s	77.04h	15.00s	15.91h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	133.28h
JCH226	54.49h	87.37h	3.58h	19.05h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	152.61h	168.00h	168.00h	168.00h	126.10h	135.20h
AAH049	15.00s	45.24h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	150.46h
JCH210	30.00s	63.56h	25.00s	155.54h	33.00s	168.00h	35.00s	168.00h	30.00s	168.00h	33.50s	168.00h	32.50s	168.00h	31.29s	151.30h
AAH035	15.00s	83.58h	15.00s	168.00h	21.50s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	19.00s	168.00h	16.50s	155.94h
JCH212	15.00s	168.00h	15.00s	94.40h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	157.49h
JCH227	68.10h	121.28h	110.51h	158.09h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	145.52h	159.91h
AAH025	139.13h	139.13h	155.04h	155.04h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	162.02h	162.02h
AAH043	25.00s	126.83h	168.00h	168.00h	16.81h	168.00h	26.50s	168.00h	25.00s	168.00h	25.00s	168.00h	25.00s	168.00h	26.41h	162.12h
AAH048	15.00s	168.00h	15.00s	128.38h	16.50s	168.00h	20.00s	168.00h	15.00s	168.00h	15.00s	168.00h	20.00s	168.00h	16.64s	162.34h
AAH050	25.00s	143.70h	151.20h	159.73h	29.00s	168.00h	33.61h	168.00h	28.00s	168.00h	29.00s	168.00h	29.00s	168.00h	26.41h	163.35h
JCH216	69.76h	142.55h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	153.97h	164.36h
AAH046	75.34h	149.74h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	154.76h	165.39h
AAH042	39.50s	151.53h	24.00s	168.00h	39.50s	168.00h	45.00s	168.00h	40.00s	168.00h	39.50s	168.00h	45.00s	168.00h	38.93s	165.65h
AAH009	21.86h	157.44h	104.71h	168.00h	125.76h	168.00h	6.48h	168.00h	56.27h	168.00h	29.35h	168.00h	24.07h	168.00h	52.64h	166.49h
JCH223	16.59m	158.18h	31.50s	168.00h	1.19h	168.00h	5.90h	168.00h	2.57h	168.00h	1.47h	168.00h	11.94h	168.00h	3.34h	166.60h
AAH029	15.00s	166.95h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	167.85h
JCH229	4.53m	167.67h	24.00s	168.00h	24.62m	168.00h	2.09h	168.00h	49.24m	168.00h	49.67m	168.00h	2.32h	168.00h	56.19m	167.95h

Table A.3 – Mean bug survival times (cont.).

	moptafl		honggfuzz		afl++		afl		aflfast		fairfuzz		symccafl		**Mean**	
Bug ID	R	T	R	T	R	T	R	T	R	T	R	T	R	T	R	T
AAH054	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h
AAH011	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h
MAE105	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	10.00s	168.00h	15.00s	168.00h	10.00s	168.00h	10.71s	168.00h
AAH005	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h
JCH202	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h
MAE114	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h
AAH034	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h
AAH004	16.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.14s	168.00h
MAE111	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	20.00s	168.00h	15.71s	168.00h
AAH059	20.00s	168.00h	15.00s	168.00h	17.00s	168.00h	20.00s	168.00h	20.00s	168.00h	20.00s	168.00h	20.00s	168.00h	18.86s	168.00h
JCH204	18.00s	168.00h	24.00s	168.00h	15.50s	168.00h	20.00s	168.00h	20.00s	168.00h	19.00s	168.00h	20.00s	168.00h	19.50s	168.00h
AAH031	20.00s	168.00h	22.50s	168.00h	42.00s	168.00h	20.00s	168.00h	20.00s	168.00h	20.00s	168.00h	25.00s	168.00h	24.21s	168.00h
AAH051	25.00s	168.00h	15.00s	168.00h	42.50s	168.00h	25.00s	168.00h	25.00s	168.00h	15.00s	168.00h	30.00s	168.00h	25.36s	168.00h
JCH214	33.50s	168.00h	53.50s	168.00h	36.00s	168.00h	35.00s	168.00h	31.00s	168.00h	30.00s	168.00h	35.00s	168.00h	36.29s	168.00h
MAE103	33.00s	168.00h	1.05m	168.00h	33.00s	168.00h	40.00s	168.00h	33.50s	168.00h	30.00s	168.00h	31.00s	168.00h	37.64s	168.00h
JCH220	4.38m	168.00h	19.50s	168.00h	22.04m	168.00h	1.78h	168.00h	46.76m	168.00h	49.27m	168.00h	2.28h	168.00h	52.36m	168.00h
AAH018	41.88m	168.00h	5.81m	168.00h	5.77h	168.00h	1.72h	168.00h	1.60h	168.00h	1.40h	168.00h	1.85h	168.00h	1.88h	168.00h
JCH230	4.02m	168.00h	41.00s	168.00h	1.07h	168.00h	5.67h	168.00h	1.16h	168.00h	1.14h	168.00h	5.57h	168.00h	2.10h	168.00h
JCH233	8.31m	168.00h	23.55m	168.00h	6.16h	168.00h	11.84h	168.00h	1.44h	168.00h	2.41h	168.00h	5.17h	168.00h	3.93h	168.00h
JCH231	21.88m	168.00h	35.00s	168.00h	2.44h	168.00h	5.99h	168.00h	5.22h	168.00h	1.69h	168.00h	11.96h	168.00h	3.95h	168.00h
MAE006	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	168.00h	168.00h	24.00h	168.00h
MAE004	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	15.00s	168.00h	168.00h	168.00h	24.00h	168.00h
AAH047	25.00s	168.00h	168.00h	168.00h	16.81h	168.00h	26.50s	168.00h	25.00s	168.00h	25.00s	168.00h	25.00s	168.00h	26.41h	168.00h
JCH222	1.75h	168.00h	39.17m	168.00h	113.15h	168.00h	151.50h	168.00h	57.91h	168.00h	71.58h	168.00h	136.02h	168.00h	76.08h	168.00h
JCH219	72.02h	168.00h	168.00h	168.00h	162.32h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	168.00h	153.48h	168.00h

Appendix B

IGOR Extended Results

B.1 Robustness under Non-uniformity

Clustering precision is impacted by the data distribution [139]. However, the number of crashes and test cases is unknown for each bug, which means that we cannot guarantee a uniform distribution of test cases across unique bugs. Therefore, whether accurate clustering results can still be obtained under different data distributions is a key indicator to measure the effectiveness of our method. Tables B.1 and B.2 show the clustering results of IGOR under different test case distributions.

Selecting `tiffcp` as the target program, we alter the number of test cases of five bugs respectively, the test case distribution in trial 1 is the same as what it is in table 3.1, the distribution in trial 2 is roughly inverse to trial 1, with a decrease in total number, while in trial 3, the test cases are evenly distributed. Based on trial 3, we alter the number of bugs. The results show that IGOR is stable against varying test case distributions.

Table B.1 – Clustering results of `tiffcp` with different sample distribution.

Bug	**Samples** Trial 1	 Trial 2	 Trial 3
CVE-2016-5314	341	50	100
CVE-2016-10269$_1$	251	90	100
CVE-2016-10269$_2$	149	200	100
CVE-2015-8784	50	278	100
CVE-2019-7663	200	100	100
Result			
Sum	991	718	500
Purity (%)	99.4	99.4	99.6
Inverse Purity (%)	99.4	99.4	99.6
F-measure (%)	99.4	99.4	99.6

Table B.2 – Clustering results of `tiffcp` with randomly sampled bugs. The notation A-E represent the bug ID of the corresponding position in section B.2. The *Clusters* column represents the number of clusters generated by IGOR; the *P*, *IP* and *F* columns are the same as in table 3.1.

Trial	**Bugs** A	 B	 C	 D	 E	**Statistics** **Clusters**	 **P** (%)	 **IP** (%)	 **F** (%)
1	✓	✓	✓	✓	✓	5	99.6	99.6	99.6
2	✓	✓		✓	✓	4	99.7	99.7	99.7
3		✓	✓		✓	3	99.3	99.3	99.3
4	✓			✓		2	100	100	100

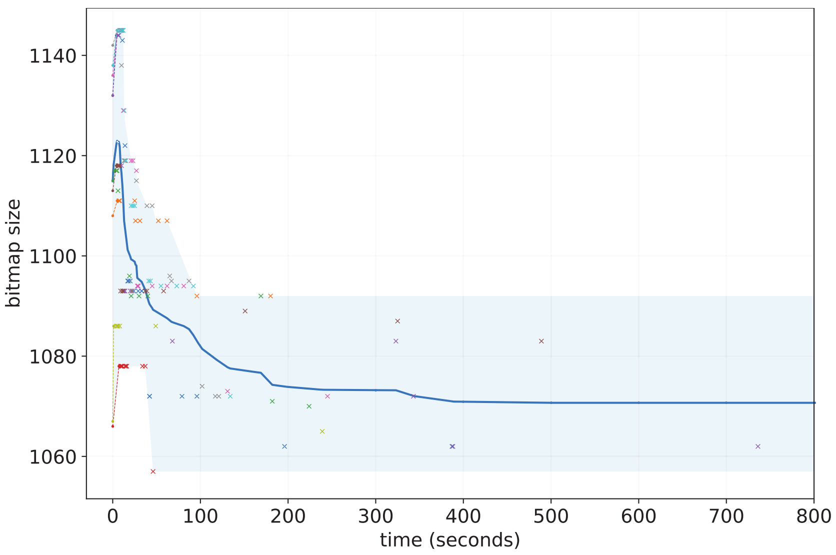


Figure B.1 – `afl-tmin`'s side effect and IGOR's ability to mitigate against it. The colored dots show individual test cases, the blue line indicates mean bitmap size and light blue the error band.

B.2 Data Set Statistics

We surveyed our dataset and counted the number of unique crash addresses and call stacks. These results are presented in section B.2.

B.3 IGORFUZZ Mitigates `afl-tmin` Limitation

As a dual-phase crash de-duplication technique, IGOR requires that the control flow of each PoC is concise after the first phase, so that the second phase can get more precise clustering results. Clustering directly after `afl-tmin`, i.e., without IGORFUZZ results in low precision as `afl-tmin` is non-monotonic and provides inadequate control flow reduction. IGORFUZZ mitigates these limitations.

During our evaluation, we found that `afl-tmin` may increase a PoC's bitmap size, e.g., in *OpenSSL*'s `x509` corpus (CVE-2016-2108). We selected ten seeds with the largest change in bitmap size being processed by `afl-tmin` to demonstrate this phenomenon. As shown in fig. B.1, the bitmap size of the ten PoCs rose in the first 20 seconds, which corresponds to the reduction process of `afl-tmin`. This indicates that as `afl-tmin` deletes bytes from a PoC file, the corresponding bitmap size does not always decrease monotonically. If we pass the `afl-tmin`-reduced PoCs to the second phase of IGOR, we may worsen results because the control flow of the PoCs contains additional bug-irrelevant paths.

However, IGORFUZZ mitigates this limitation. The blue line in the same figure shows mean bitmap size of the ten PoCs during reduction. Taking the `afl-tmin`-reduced PoCs as input, IGORFUZZ reduced the bitmap size monotonically, and all of the ten PoCs finally got smaller bitmap size than their original size.

In summary, `afl-tmin` sometimes increases the bitmap size, which contradicts the requirement of IGOR's first phase. IGORFUZZ mitigates this issue by focusing on bitmap size reduction rather than input size reduction, making it more practical to do classification.

B.4 IGORFUZZ Does Not Waste Time on Error Seeds

To study the minimization process of error introducing PoCs, we explicitly selected all the queues generated during the minimization process of the error PoCs, analyzing the change in the proportion of the error seed in all seeds over time. According to the evaluation result shown in fig. B.2, the proportion of error seeds is small, which accounts for less than 10 % in three programs, and the lowest is tiffcp, which is only 0.93 %. Except for xmllint, the proportion of erroneous seeds decreases over time, which means the effect of error seed on minimization is ever decreasing and minimization mainly explores the paths near the original vulnerability.

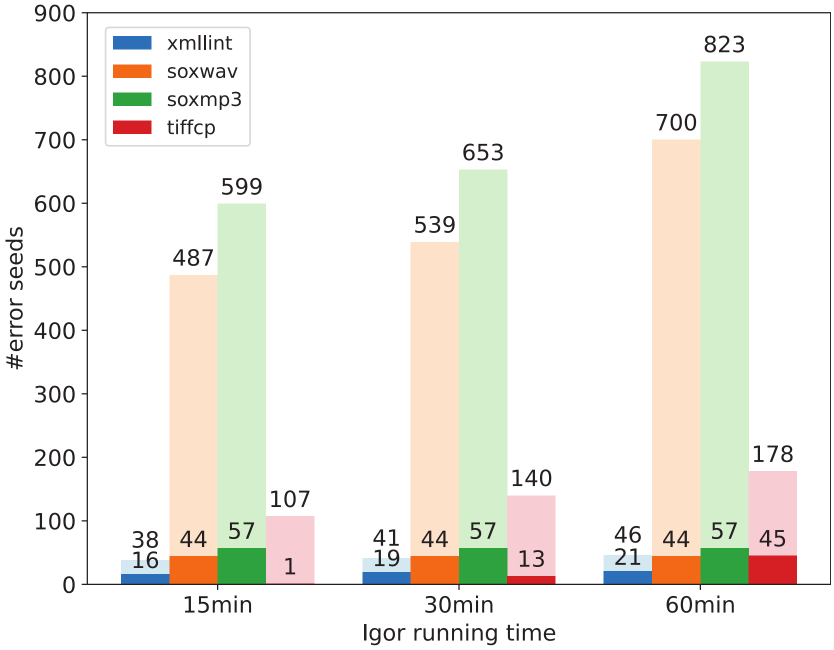
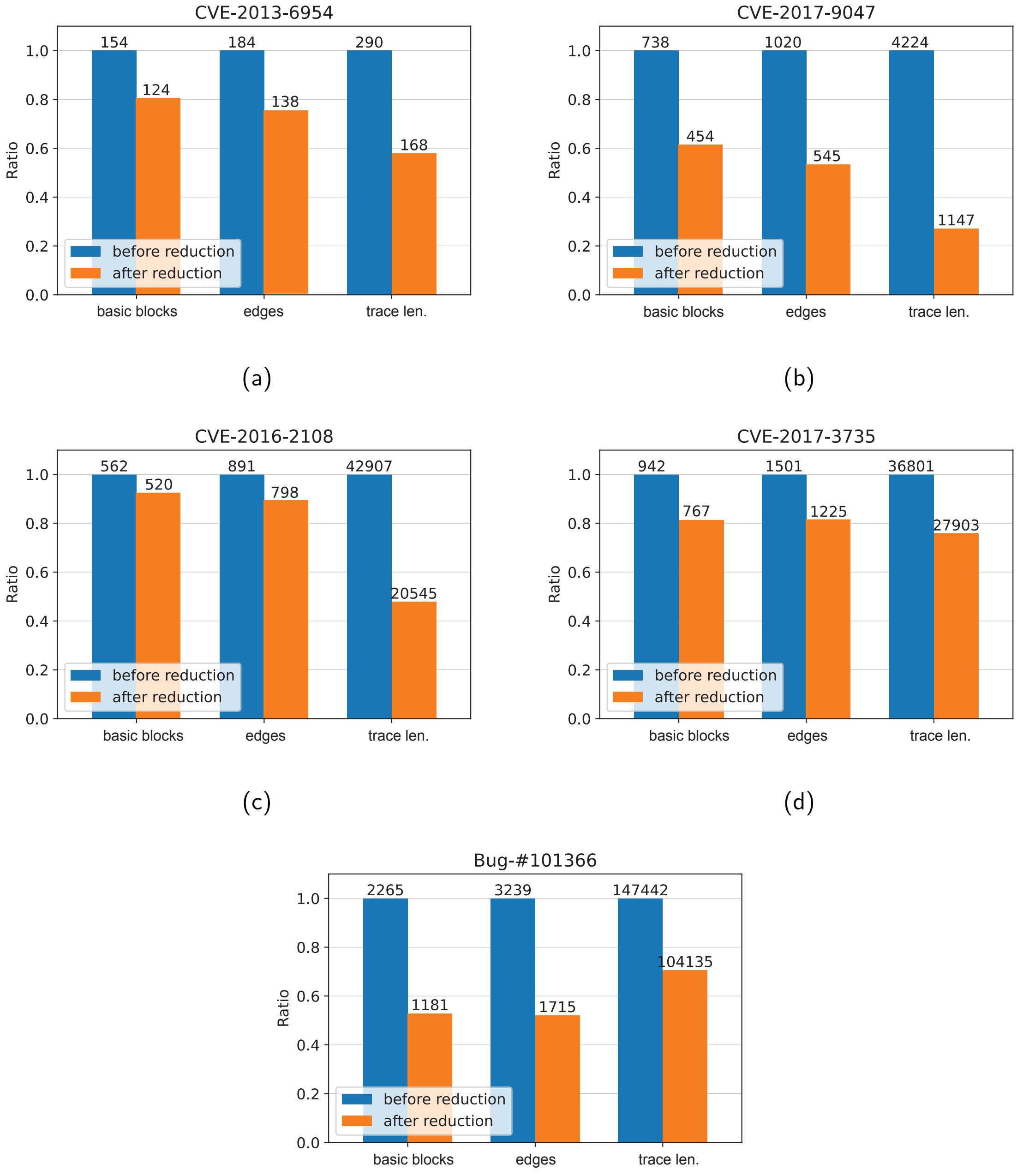


Figure B.2 – Seed error rate. In this figure, the bars in faded colors indicate total number of generated seeds in error queues (including normal seeds and error seeds); the bars in dark colors are total number of error seeds in error queues.

Error seeds occupy a significant proportion in the queue of xmllint, but the proportion decreases over time. We analyzed the reasons for the relatively high error seed in the queue of xmllint: The length of the PoCs from xmllint is much shorter than the length of PoCs of other programs, and the code depth of the vulnerability is relatively shallow. The short seed length combined with shallow bugs makes it likely for a fuzzer to discover alternate bugs through only small mutations.

Still, error seeds occupy a relatively small proportion in the queues overall and individually, which means that they have a small probability of being further mutated. After manual debugging, we found that the crash addresses of these error seeds are all different from those of the original PoCs. Therefore, even if a small part of the error seed is generated during the minimization, all these errors will be eliminated by our selection criteria (see details in section 3.2.2).



(a)

(b)

(c)

(d)

(e)

Figure B.3 – Comparison in three dimensions (basic blocks, edges, and trace length) before and after reduction with IGORFUZZ.

Table B.3 – Statistics of unique crash addresses and unique call stacks. When the crash address count is given as n/m, n is the number of crash addresses, while m is the number of non-unique crash addresses. The bug IDs of a specific program with superscript * or $^+$ share the same underlying vulnerability. There are two unique underlying bugs assigned the same CVE—*CVE-2016-10269*. We use subscripts to distinguish them.

Target	Program	Bug ID	Crash addrs	Call stacks
Poppler	pdftoppm	CVE-2017-14617	1	1
		CVE-2019-7310	1	3
		Bug #101366	1	1
	pdfimages	CVE-2017-9865	1	1
		CVE-2018-10768	1	1
		CVE-2019-7310	1	2
		CVE-2017-9776	10	21
libtiff	tiffcp	CVE-2015-8784	1	1
		CVE-2019-7663	2/2	4
		CVE-2016-10269$_1$	3/2	5
		CVE-2016-10269$_2$	2/2	3
		CVE-2016-5314	1	7
PHP	exif	CVE-2018-14883	13	50
OpenSSL	client	CVE-2016-6309	1	1
	x509	CVE-2016-2108	1	2
		CVE-2017-3735	1	1
libxml2	xml_read_mem	CVE-2017-9047	3	28
libpng	png_read	CVE-2013-6954	1	2
		CVE-2018-13785	1	1
libxml2	xmllint	CVE-2015-8317	15/10	204
		CVE-2015-7497	1/1	198
		CVE-2016-1835	1	3
		CVE-2016-1836	1/1	5
		CVE-2016-1762	6/3	245
		CVE-2016-3627	3	3
		CVE-2015-7942	1	1
		CVE-2015-7499*	1/1	1
		CVE-2015-7498*	7/6	242
libtiff	tiff2pdf	CVE-2019-14973	3/1	6
		CVE-2017-17973	1/1	1
		Bug #C	1	1
Poppler	pdftotext	CVE-2019-12293	1	3
FreeType	char2svg	CVE-2014-9663*	1	4
		CVE-2015-9290$^+$	1	1
		CVE-2014-9658	2	8
		CVE-2014-9669*	11/1	23
		CVE-2014-2240	1	23
		CVE-2014-9659	2	3
		CVE-2015-9383*	3/1	4
		CVE-2015-9381$^+$	1	1
SoX(MP3)	sox	CVE-2019-8355	2	3
		CVE-2019-8357	5/2	5
		CVE-2019-8354*	1/1	1
		CVE-2019-8356*	1/1	2
		CVE-2017-18189	1	1
		CVE-2019-13590	1	1
SoX(WAV)	sox	CVE-2019-8355	2	3
		CVE-2019-8357	5/1	5
		CVE-2019-8354*	1/1	1
		CVE-2019-8356*	1/1	2
		CVE-2017-11332	1	1
		CVE-2017-18189	1	1

Table B.4 – PoC shrink rate with `afl-tmin`: PoC length, bitmap size, edges hit count, and side effects are listed.

Program	**Bug**	**Samples**	**Median**			**Mean**			**Variance**			**Negative**	
			len %	map %	hit %	len %	map %	hit %	len	map	hit	map %	hit %
x509	CVE-2016-2108	100	24.14	4.94	6.28	41.10	4.83	22.57	0.165	0.002	0.103	15	12
	CVE-2017-3735	100	0	15.61	7.13	3.25	15.73	6.57	0.021	0.001	0.006	0	19
png_read	CVE-2018-13785	80	1.23	17.48	1.93	2.81	14.43	3.80	0.010	0.009	0.034	1.25	36.25
	CVE-2013-6954	100	14.37	0.13	-0.69	21.58	1.33	-1.27	0.054	0.0009	0.016	18	65
pdftoppm	Bug #101366	91	99.11	5.88	36.62	82.45	12.17	33.97	0.118	0.021	0.090	15.18	12.04
	CVE-2019-7310	100	0	-1.06	16.12	0	0.85	11.11	0	0.006	0.088	53.5	39.5
	CVE-2017-14617	97	96.05	5.81	32.35	96.31	8.06	31.36	0.0001	0.003	0.030	2.53	4.56
pdfimages	CVE-2017-9865	83	98.20	13.52	12.57	65.18	12.41	5.50	0.199	0.003	0.049	1.20	34.93
tiffcp	CVE-2016-5314	341	0.78	11.85	18.12	14.40	12.56	19.33	0.086	0.002	0.006	0	0
	CVE-2016-10269$_1$	251	10.71	16.32	24.15	14.22	16.16	24.97	0.031	0.001	0.002	0	0
	CVE-2016-10269$_2$	149	1.25	15.08	20.20	10.50	15.13	19.85	0.045	0.001	0.002	0	0
	CVE-2015-8784	50	1.25	15.56	21.08	13.57	19.76	25.54	0.066	0.012	0.026	0	2
	CVE-2019-7663	200	10.71	20.23	32.96	37.33	19.72	38.39	0.134	0.004	0.024	1.5	0

Table B.5 – Evaluation results (0 and 60 min IGORFUZZ running time, the "*" in the cut-off time column indicates clustering results that ground-truth bug labels are assigned).

Program	B	N	C	S	K	**Top Frame** (%) P	IP	F	**BFF-5** (%) P	IP	F	**honggfuzz** (%) P	IP	F	**afl-collect** (%) P	IP	F	**cut-off (mins)**	IGOR (%) P	IP	F
					2													0	88	94	87
pdfimages	3	410	12	23	3	100	48	63	100	47	62	100	47	62	100	47	62	60	100	100	100
																		*	99	99	99
					2													0	70	99	79
pdftoppm	3	161	3	5	2	100	100	100	100	95	98	100	95	98	100	95	98	60	69	100	80
																		*	100	100	100
					17													0	98	65	77
tiffcp	5	991	3	20	5	85	89	80	88	76	74	88	67	68	88	67	68	60	100	100	100
																		*	90	90	92
					3													0	98	98	98
tiff2pdf	3	385	2	8	3	92	91	89	99	89	93	99	89	94	99	89	94	60	98	98	98
																		*	98	98	98
					2													0	100	100	100
x509	2	150	2	3	2	100	100	100	100	100	100	100	75	83	100	75	83	60	100	100	100
																		*	100	100	100
					2													0	100	100	100
png_read	2	150	2	3	2	100	72	81	100	72	81	100	72	81	100	72	81	60	100	100	100
																		*	100	100	100
					4													0	72	95	77
xmllint	8	1581	14	901	19	78	59	64	78	55	62	78	55	62	79	3	4	60	97	63	74
																		*	94	74	82
					6													0	90	67	71
char2svg	5	1087	20	67	9	100	66	72	100	14	20	100	14	20	100	14	20	60	100	67	79
																		*	86	67	69
					4													0	100	100	100
sox (MP3)	4	260	6	8	4	100	63	74	100	58	71	100	58	71	100	58	71	60	100	100	100
																		*	100	100	100
					4													0	100	100	100
sox (WAV)	4	356	6	8	4	100	67	77	100	66	76	100	66	76	100	66	76	60	100	100	100
																		*	100	100	100

Appendix C

TANGO Plays DOOM

C.1 Built-in Extensions

Components in TANGO are left open for customization, to accommodate for arbitrary stateful systems beyond network services. The framework also encourages re-usability by providing components with fixed interfaces, as well as a dynamic component discovery subsystem to ensure that specialized co-dependent modules are instantiated together. Through its supplementary profiling module, TANGO also enables instrumentation of the fuzzer's own functions and variables, to improve debuggability of the fuzzer's operations and to better attribute feature changes to improvements in performance.

C.1.1 `ptrace`-d processes

To ensure synchronicity between the fuzzer and the running process, we use `ptrace` with `seccomp` filters to place catchpoints over the relevant I/O syscalls where necessary, e.g., `read`, `dup`, `close`, and `poll`, among others. Synchronization allows the target to return control to the fuzzer as soon as it becomes ready, instead of busy-waiting and degrading throughput. Moreover, it guarantees reproducibility of results: by encoding the relevant sequences of syscalls into its saved inputs, the fuzzer can reliably reproduce states and coverage measurements, leaving little for the OS to influence when data is delivered to the target.

In addition, to increase fuzzer throughput and exploit the redundancy of resets, TANGO leverages `ptrace` to dynamically inject a forkserver at runtime, just after setting up the communication channel in the target. This relieves the loader of the heavy initialization phase of many network services.

C.1.2 Container isolation

Fuzzing is an embarrassingly parallel process, and it is commonly employed by launching multiple concurrent campaigns on capable machines. When fuzzing network services, this can introduce the problem of overlapping socket bind addresses, since a server's configuration parameters are often identical across campaigns. We therefore isolate each fuzzer instance in a Linux network namespace, allowing them to communicate with their target without aliasing other instances.

We also leverage mount namespaces to achieve filesystem isolation. Some targets may store persistent state through local files, influencing other instances of the target across resets. To avoid that, we mount an `overlay` filesystem on top of a `tmpfs` mount point for storing instance-local data. Then, upon reset, we clear the `upper` filesystem of the `overlay`, effectively destroying any persistent state left by the target.

C.1.3 Record-and-replay

TANGO ships with a default loader which implements a record-and-replay mechanism for loading snapshots. Under the reasonable assumption that the target is deterministic, such a loader can reliably reproduce paths by relaunching or forking the target and re-applying a saved input. More sophisticated snapshot-ing methods exist [141] which can be ported for use under TANGO; however, this remains out of scope of the current extensions.

C.1.4 SanitizerCoverage

In our study on state inference, we primarily model features as code coverage profiles, classified into AFL-style bins. To achieve that, TANGO implements a COVERAGETRACKER which sets up a shared memory region for communicating coverage updates and extracting feature sets. The tracker is equipped with C-based bindings for performing the binning and hashing with minimal impact on the fuzzer's critical path.

C.1.5 `socket+stdio`

TANGO includes a demonstrative set of channels, to communicate with the target over network sockets, such as TCP and UDP, as well as standard input. These channels extend the `ptrace` functionality to synchronize the state of the socket or file in the target with its counterpart in the fuzzer. By capturing syscalls such as `bind` and `accept`, we inject a forkserver at the latest stage in initializing the target. This achieves around a 50x increase in fuzzing throughput over the non-optimized implementation, since socket setup is often expensive, and otherwise, the fuzzer may only successfully connect to the target by reattempting the operation until it no longer fails.

C.1.6 Adaptive mutators

We implement an adaptive model for applying havoc mutators that balances exploration and exploitation using the Exp3 [13] algorithm for distributing rewards and assigning probabilities. For each snapshot, we maintain a set of weights describing the probability that a mutator is chosen in that state. Provided a comprehensive set of mutators, this approach accommodates for an evolving target by adjusting and applying the probabilities in selecting the next mutator, based on how well each one performs in the state context.

C.2 Case Study: DOOM

TANGO is a framework for building state-aware fuzzers, and the main witness of its merit would be using it to develop a state-aware fuzzer for a complex stateful system. In the spirit of hacker culture, we opted to answer the question "Can it run DOOM?" with a resounding "Yes!", using TANGO.

C.2.1 Setup

To add support for DOOM, we extended the following interfaces:

Driver: We implemented an `X11Channel` component which sends keystroke events to a process's window, through a public Python library, `python3-xlib`.

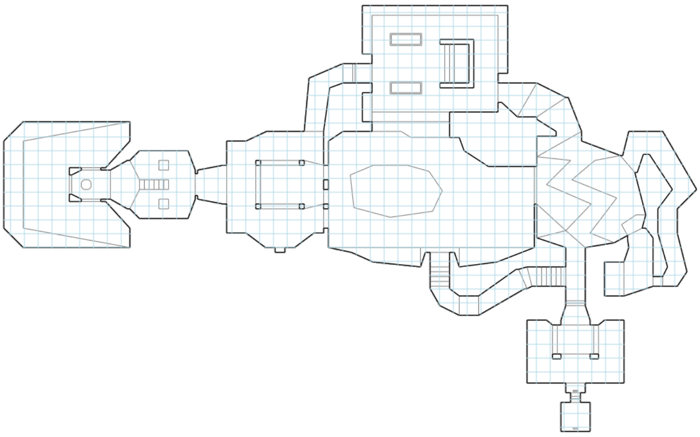
Generator: We extended the input base class with a set of instructions including `Activate`, `Kill`, `Move`, `Reach`, `Rotate`, and `Shoot`. These govern how the player character interacts with its environment, and they are later used by the input generator and the exploration strategy to maneuver around the map and overcome obstacles.

We limited the functionality of the input generator to selecting a possible outgoing or incoming transition of the current state and passing it on to the mutator, which mainly mutates the direction and duration of movement, to explore the level. We also provided it with two helper functions that can yield the correct sequence of commands to follow a path or to aim and shoot at a moving target by incorporating live feedback.

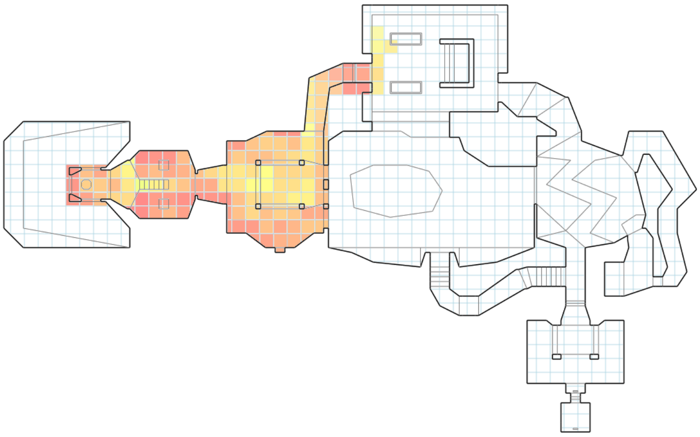
Tracker: We implemented state feedback as a shared-memory struct populated by DOOM and accessed by TANGO. The struct contains all basic user properties such as location, weapons, ammo, pickups, enemies in sight, and doors or switches within reach. Two states are considered equivalent if they have the same player position. We attached extended state variables to each state that describe the pickups collected along the current path, and state attributes describing the current location (e.g. if it is a slime pit or a secret level).

To avoid having a unique state for every single position on the map, the level is divided into a grid of cells, representing the granularity of feedback, as shown in fig. C.1a.

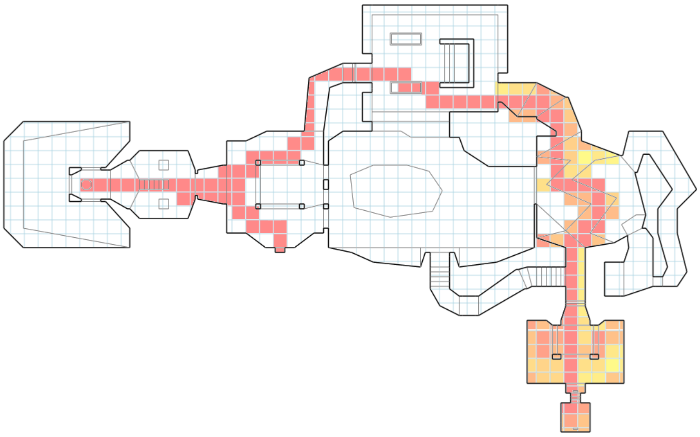
Loader: We extended the loader's two main functions: restarting the target and loading a state. Restarts are simple, as they're only a matter of terminating and relaunching the process. Loading a state is slightly more complex: without a means of snapshotting, actions must be replayed to reach a certain known location, given that the state graph contains at least one path to it. However, a downside of this approach is that nearby states (locations) may be reachable through a bee-line movement, whereas the input generated and discovered by the fuzzer to transition between these



(a) Grid view of the DOOM map as seen by TANGO.



(b) Heatmap of the visited cells during the first 10 minutes of fuzzing the E1M1 level of DOOM.



(c) Heatmap of the visited cells after 30 minutes of fuzzing. The fuzzer had figured out a path to the finish and continues to repeat it to achieve the lowest completion time.

Figure C.1 – The progress of TANGO in playing DOOM. Red shading implies higher hit counts.

two states may involve redundancies that impact fuzzer throughput. Another downside is that if the current location and the target location are close to each other, yet are far enough from the spawn location, restarting the level from that location would be inefficient. The player may simply need to move a few steps in the direction of the target to reach it. Moreover, continuously restarting the target breaks the immersion of the fuzzer “playing” the game, and it would instead spend much of its time replaying actions from the start. To avoid that, we implemented path-finding algorithms, based on the state graph explored by the fuzzer, to move between two locations using a sequence of `Reach` instructions. In essence, to load a state, a path to it from the current state is calculated, and `Reach` instructions are performed piece-wise along every transition in the path.

Strategy: Finally, to tie it all together, we implemented a `ZoomStrategy` component that schedules states based on a convex hull of the explored locations, and prioritizes locations on the perimeter, that are furthest from the start.

In addition, the strategy implements an event observer task that is responsible for reacting to urgent events such as enemy sighting or stepping in slime pits. By preempting the fuzzer’s main loop, the strategy minimizes the reaction time to increase the survivability of the player.

C.2.2 Results

With these extensions, TANGO consistently manages to finish the E1M1 level of DOOM, on difficulty 3, in 10 to 40 wall-clock minutes. The main factor contributing to this variability is perimeter exploration. As can be seen in fig. C.1b, in one recorded fuzzing session, the fuzzer encountered a big undiscovered area on the left side of the map, and dedicated a significant amount of time to exploring it. It also managed to discover the path up the stairs to the higher platform, where it found a level 1 armor pickup. In other runs, due to the stochastic nature of the fuzzing process, it may miss that area completely and continue exploring in the immediate direction of the exit door, achieving a lower overall finish time in the process.

Having found the armor pickup, TANGO maintains a record of it in its later exploration stages. As can be seen in fig. C.1c, its path to the finish line includes going up the stairs, picking up the armor boost, and returning back on another path to the exit room.

Regardless of the overall finish time, once TANGO finds the path to the exit, it consistently manages to follow it in 3 to 4 in-game minutes. While far from typical speed-runs for this level, which have hit as low as 9 seconds, it is still a formidable achievement to be able to explore the state space of a DOOM level and manage to finish it in a sensible amount of time.

Bibliography

[1] Joshua Ackerman and George Cybenko. «Large Language Models for Fuzzing Parsers (Registered Report)». In: *2nd International Fuzzing Workshop (FUZZING)*. 2023.

[2] Kayla Afanador and Cynthia Irvine. «Representativeness in the Benchmark for Vulnerability Analysis Tools (B-VAT)». In: *13th USENIX Workshop on Cyber Security Experimentation and Test (CSET)*. 2020.

[3] Karan Aggarwal, Finbarr Timbers, Tanner Rutgers, Abram Hindle, Eleni Stroulia, and Russell Greiner. «Detecting Duplicate Bug Reports with Software Engineering Domain Knowledge». In: *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2017.

[4] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. *Announcing OSS-Fuzz: Continuous Fuzzing for Open Source Software*. Accessed: 2019-09-09. 2016. URL: `https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html`.

[5] Enrique Amigó, Julio Gonzalo, Javier Artiles, and Felisa Verdejo. «A Comparison of Extrinsic Clustering Evaluation Metrics Based on Formal Constraints». In: *Information Retrieval* 12.4 (2009).

[6] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. «OPTICS: Ordering Points to Identify the Clustering Structure». In: *25th ACM SIGMOD International Conference on Management of Data (MOD)*. 1999.

[7] Apple. *Diagnosing Issues Using Crash Reports and Device Logs*. Accessed: 2023-10-18. 2010. URL: `https://developer.apple.com/library/archive/technotes/tn2004/tn2123.html`.

[8] Branden Archer and Darkkey. *radamsa: A Black-Box Mutational Fuzzer*. Accessed: 2019-09-09. URL: `https://gitlab.com/akihe/radamsa`.

[9] Brad Arkin. *Adobe Reader and Acrobat Security Initiative*. Accessed: 2019-09-09. 2009. URL: `http://blogs.adobe.com/security/2009/05/adobe_reader_and_acrobat_secur.html`.

[10] Abhishek Arya and Cris Neckar. *Fuzzing for Security*. Accessed: 2019-09-09. 2012. URL: `https://blog.chromium.org/2012/04/fuzzing-for-security.html`.

[11] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. «IJON: Exploring Deep State Spaces via Fuzzing». In: *41st IEEE Symposium on Security and Privacy (SP)*. 2020.

[12] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. «REDQUEEN: Fuzzing with Input-to-State Correspondence». In: *27th Network and Distributed System Security Symposium (NDSS)*. 2019.

[13] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. «The Nonstochastic Multiarmed Bandit Problem». In: *SIAM Journal on Computing* 32.1 (2002).

[14] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. «Stateful Greybox Fuzzing». In: *31st USENIX Security Symposium (USENIX Security)*. 2022.

[15] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. «FUDGE: Fuzz Driver Generation at Scale». In: *27th ACM SIGSOFT Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2019.

[16] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. «A Survey of Symbolic Execution Techniques». In: *ACM Computing Surveys* 51.3 (2018).

[17] Nils Bars, Moritz Schloegel, Tobias Scharnowski, Nico Schiller, and Thorsten Holz. «Fuztruction: Using Fault Injection-based Fuzzing to Leverage Implicit Domain Knowledge». In: *32nd USENIX Security Symposium (USENIX Security)*. 2023.

[18] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. «The Dacapo Benchmarks: Java Benchmarking Development and Analysis». In: *21st ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. 2006.

[19] 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)*. 2020.

[20] Marcel Böhme and Brandon Falk. «Fuzzing: On the Exponential Cost of Vulnerability Discovery». In: *28th ACM SIGSOFT Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2020.

[21] Marcel Böhme, Valentin J. M. Manès, and Sang Kil Cha. «Boosting Fuzzer Efficiency: An Information Theoretic Perspective». In: *28th ACM SIGSOFT Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2020.

[22] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. «Coverage-Based Greybox Fuzzing as Markov Chain». In: *23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016.

[23] Ingwer Borg and Patrick J. F. Groenen. «Modern Multidimensional Scaling: Theory and Applications». In: Springer New York, 2005. Chap. Classical Scaling, pp. 261–267.

[24] Mark Brodie, Sheng Ma, Leonid Rachevsky, and Jon Champlin. «Automated Problem Determination Using Call-Stack Matching». In: *Journal of Network and Systems Management* 13.2 (2005).

[25] Ricardo J. G. B. Campello, Davoud Moulavi, and Jörg Sander. «Density-Based Clustering Based on Hierarchical Density Estimates». In: *17th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. 2013.

[26] Brian Caswell. *Cyber Grand Challenge Corpus*. Accessed: 2023-10-18. 2015. URL: `http://www.lungetech.com/cgc-corpus`.

[27] Brian Chan, Ying Zou, Ahmed E. Hassan, and Anand Sinha. «Visualizing the Results of Field Testing». In: *18th IEEE International Conference on Program Comprehension (ICPC)*. 2010.

[28] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. «IoTFuzzer: Discovering Memory Corruptions in Iot through App-Based Fuzzing.» In: *26th Network and Distributed System Security Symposium (NDSS)*. 2018.

[29] Peng Chen and Hao Chen. «Angora: Efficient Fuzzing by Principled Search». In: *39th IEEE Symposium on Security and Privacy (SP)*. 2018.

[30] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. «Taming Compiler Fuzzers». In: *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2013.

[31] Arpit Christi, Matthew Lyle Olson, Mohammad Amin Alipour, and Alex Groce. «Reduce before You Localize: Delta-Debugging and Spectrum-Based Fault Localization». In: *29th IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2018.

[32] James Clause and Alessandro Orso. «Penumbra: Automatically Identifying Failure-Relevant Inputs Using Dynamic Tainting». In: *18th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2009.

[33] Nicolas Coppik, Oliver Schwahn, and N. Suri. «MemFuzz: Using Memory Accesses to Guide Fuzzing». In: *12th IEEE International Conference on Software Testing, Validation and Verification (ICST)*. 2019.

[34] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. «REPT: Reverse Debugging of Failures in Deployed Software». In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2018.

[35] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P. Kemerlis. «Retracer: Triaging Crashes by Reverse Execution from Partial Memory Dumps». In: *38th IEEE/ACM International Conference on Software Engineering (ICSE)*. 2016.

[36] Yingnong Dang, Rongxin Wu, Hongyu Zhang, Dongmei Zhang, and Peter Nobel. «ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity». In: *34th IEEE/ACM International Conference on Software Engineering (ICSE)*. 2012.

[37] Sanjeev Das, Kedrian James, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. «A Flexible Framework for Expediting Bug Finding by Leveraging Past (mis-) Behavior to Discover New Bugs». In: *37th Annual Computer Security Applications Conference (ACSAC)*. 2020.

[38] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. «A fast and elitist multiobjective genetic algorithm: NSGA-II». In: *IEEE Transactions on Evolutionary Computation* 6.2 (2002).

[39] Tejinder Dhaliwal, Foutse Khomh, and Ying Zou. «Classifying Field Crash Reports for Fixing Bugs: A Case Study of Mozilla Firefox». In: *27th IEEE International Conference on Software Maintenance (ICSM)*. 2011.

[40] Brendan Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, Andrea Mambretti, William K. Robertson, F. Ulrich, and Ryan Whelan. «LAVA: Large-Scale Automated Vulnerability Addition». In: *37th IEEE Symposium on Security and Privacy (SP)*. 2016.

[41] William Drozd and Michael D. Wagner. «FuzzerGym: A Competitive Framework for Fuzzing and Learning». In: *Computing Research Repository* (2018).

[42] T. Dullien. «Weird Machines, Exploitability, and Provable Unexploitability». In: *IEEE Transactions on Emerging Topics in Computing* 8.2 (2020).

[43] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. «FormatFuzzer: Effective Fuzzing of Binary File Formats». In: *Computing Research Repository* (2021).

[44] Michael Eddington. *Peach Fuzzer Platform*. Accessed: 2023-10-18. 2004. URL: `https://peachtech.gitlab.io/peach-fuzzer-community`.

[45] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. «Graph Drawing Software». In: Springer New York, 2004. Chap. Graphviz and Dynagraph—Static and Dynamic Graph Drawing Tools, pp. 127–148.

[46] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. «A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise». In: *2nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 1996.

[47] Rong Fan and Yaoyao Chang. «Machine Learning for Black-Box Fuzzing of Network Protocols». In: *19th International Conference on Information and Communications Security (ICICS)*. 2017.

[48] 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)*. 2021.

[49] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. «AFL++: Combining Incremental Steps of Fuzzing Research». In: *14th USENIX Workshop on Offensive Technologies (WOOT)*. 2020.

[50] Andrea Fioraldi, Alessandro Mantovani, Dominik Maier, and Davide Balzarotti. «Dissecting American Fuzzy Lop: A FuzzBench Evaluation». In: *ACM Transactions on Software Engineering and Methodology* 32.2 (2023).

[51] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. «CollAFL: Path Sensitive Fuzzing». In: *39th IEEE Symposium on Security and Privacy (SP)*. 2018.

[52] Vijay Ganesh, Tim Leek, and Martin C. Rinard. «Taint-Based Directed Whitebox Fuzzing». In: *31st IEEE/ACM International Conference on Software Engineering (ICSE)*. 2009.

[53] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. «Debugging in the (very) Large: Ten Years of Implementation and Experience». In: *22nd ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 2009.

[54] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. «Grammar-Based Whitebox Fuzzing». In: *29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2008.

[55] Google. *AI-Powered Fuzzing: Breaking the Bug Hunting Barrier*. Accessed: 2023-10-09. 2023. URL: `https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html`.

[56] Google. *ClusterFuzz*. Accessed: 2023-10-18. 2021. URL: `https://google.github.io/clusterfuzz`.

[57] Google. *FuzzBench*. Accessed: 2023-10-18. 2020. URL: `https://google.github.io/fuzzbench`.

[58] Google. *Fuzzer Test Suite*. Accessed: 2023-10-18. 2017. URL: `https://github.com/google/fuzzer-test-suite`.

[59] Google. *Syzbot Dashboard*. Accessed: 2023-10-18. 2015. URL: `https://syzkaller.appspot.com`.

[60] Rahul Gopinath, Björn Mathis, and Andreas Zeller. «Mining Input Grammars from Dynamic Control Flow». In: *28th ACM SIGSOFT Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2020.

[61] Gustavo Grieco, Martín Ceresa, and Pablo Buiras. «QuickFuzz: An Automatic Random Fuzzer for Common File Formats». In: *9th ACM SIGPLAN International Symposium on Haskell*. 2016.

[62] Sumit Gulwani. «Dimensions in Program Synthesis». In: *12th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP)*. 2010.

[63] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. «Exploring Network Structure, Dynamics, and Function Using Networkx». In: *7th Python in Science Conference (SciPy)*. 2008.

[64] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. «Magma: A Ground-Truth Fuzzing Benchmark». In: *47th ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (METRICS)*. 2021.

[65] Adrian Herrera, Hendra Gunadi, Liam Hayes, Shane Magrath, Felix Friedlander, Maggi Sebastian, Michael Norrish, and Antony L. Hosking. «Corpus Distillation for Effective Fuzzing: A Comparative Evaluation». In: *Computing Research Repository* (2020).

[66] Adrian Herrera, Hendra Gunadi, Shane Magrath, Michael Norrish, Mathias Payer, and Antony L. Hosking. «Seed Selection for Successful Fuzzing». In: *30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2021.

[67] Daniel S. Hirschberg. «Algorithms for the Longest Common Subsequence Problem». In: *Journal of the Association for Computing Machinery* 24.4 (1977).

[68] Josie Holmes and Alex Groce. «Using Mutants to Help Developers Distinguish and Debug (compiler) Faults». In: *Software Testing, Verification and Reliability* 30.2 (2020).

[69] Allen D. Householder and Jonathan M. Foote. *Probability-Based Parameter Selection for Black-Box Fuzz Testing*. Tech. rep. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2012.

[70] Intel. *Intel Pin API Reference*. Accessed: 2023-10-18. URL: `https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-binary-instrumentation-tool-downloads.html`.

[71] laf intel. *Circumventing Fuzzing Roadblocks with Compiler Transformations*. Accessed: 2023-10-09. 2016. URL: `https://lafintel.wordpress.com/2016/08/15/circumventing-fuzzing-roadblocks-with-compiler-transformations/`.

[72] Kyriakos K. Ispoglou, D. Austin, V. Mohan, and Mathias Payer. «FuzzGen: Automatic Fuzzer Generation». In: *29th USENIX Security Symposium (USENIX Security)*. 2020.

[73] Hyeon-Gu Jeon, Seong-Kyun Mok, and Eun-Sun Cho. «Automated Crash Filtering Using Interprocedural Static Analysis for Binary Codes». In: *41st IEEE Annual Computer Software and Applications Conference (COMPSAC)*. 2017.

[74] Samuel Jero, Maria Leonor Pacheco, Dan Goldwasser, and Cristina Nita-Rotaru. «Leveraging Textual Specifications for Grammar-Based Fuzzing of Network Protocols». In: *33rd AAAI Conference on Innovative Applications of Artificial Intelligence (IAAI)*. 2019.

[75] Zhiyuan Jiang, Xiyue Jiang, Ahmad Hazimeh, Chaojing Tang, Chao Zhang, and Mathias Payer. «Igor: Crash Deduplication Through Root-Cause Clustering». In: *28th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021.

[76] Ajay Joshi, Aashish Phansalkar, L. Eeckhout, and L. K. John. «Measuring Benchmark Similarity Using Inherent Program Characteristics». In: *IEEE Transactions on Computers* 55.6 (2006).

[77] Edward L. Kaplan and Paul Meier. «Nonparametric Estimation from Incomplete Observations». In: *Journal of the American Statistical Association* 53.282 (1958).

[78] Richard M. Karp. «Reducibility among Combinatorial Problems». In: *The Symposium on the Complexity of Computer Computations*. Springer, 1972.

[79] Vineeth Kashyap, Jason Ruchti, Lucja Kot, Emma Turetsky, R. Swords, David Melski, and Eric Schulte. «Automated Customized Bug-Benchmark Generation». In: *19th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2019.

[80] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. «Failure Sketching: A Technique for Automated Root Cause Diagnosis of In-Production Failures». In: *25th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. 2015.

[81] Sunghun Kim, Thomas Zimmermann, and Nachiappan Nagappan. «Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage». In: *41st IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. 2011.

[82] Jóakim v. Kistowski, Jeremy A. Arnold, Karl Huppler, Klaus-Dieter Lange, John L. Henning, and Paul Cao. «How to Build a Benchmark». In: *6th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 2015.

[83] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. «Evaluating Fuzz Testing». In: *25th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018.

[84] Danai Koutra, Ankur Parikh, Aaditya Ramdas, and Jing Xiang. *Algorithms for Graph Similarity and Subgraph Matching*. Tech. rep. Computer Science Department, Carnegie Mellon University, 2011.

[85] Nils M. Kriege, Pierre-Louis Giscard, and Richard C. Wilson. «On Valid Optimal Assignment Kernels and Applications to Graph Classification». In: *30th International Conference on Neural Information Processing Systems (NeurIPS)*. 2016.

[86] Joseph B. Kruskal. «An Overview of Sequence Comparison: Time Warps, String Edits, and Macromolecules». In: *SIAM Review* 25.2 (1983).

[87] S. Kullback and R. A. Leibler. «On Information and Sufficiency». In: *The Annals of Mathematical Statistics* 22 (1951).

[88] Bjornar Larsen and Chinatsu Aone. «Fast and Effective Text Mining Using Linear-Time Document Clustering». In: *5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. 1999.

[89] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. «PerfFuzz: Automatically Generating Pathological Inputs». In: *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2018.

[90] Caroline Lemieux and Koushik Sen. «FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage». In: *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2018.

[91] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. «SNPSFuzzer: A Fast Greybox Fuzzer for Stateful Network Protocols Using Snapshots». In: *IEEE Transactions on Information Forensics and Security* 17 (2022).

[92] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. «Steelix: Program-State Based Binary Fuzzing». In: *11th ACM SIGSOFT Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2017.

[93] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. «UniFuzz: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers». In: *30th USENIX Security Symposium (USENIX Security)*. 2021.

[94] Chung-Yi Lin, Chun-Ying Huang, and Chia-Wei Tien. «Boosting Fuzzing Performance with Differential Seed Scheduling». In: *14th Asia Joint Conference on Information Security (AsiaJCIS)*. 2019.

[95] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. «A Large-Scale Empirical Study on Vulnerability Distribution within Projects and the Lessons Learned». In: *42nd IEEE/ACM International Conference on Software Engineering (ICSE)*. 2020.

[96] Yanchi Liu, Zhongmou Li, Hui Xiong, Xuedong Gao, and Junjie Wu. «Understanding of Internal Clustering Validation Measures». In: *10th IEEE International Conference on Data Mining (ICDM)*. 2010.

[97] V. Benjamin Livshits and Monica S. Lam. «Finding Security Vulnerabilities in Java Applications with Static Analysis». In: *14th USENIX Security Symposium (USENIX Security)*. 2005.

[98] LLVM. *libFuzzer: A Library for Coverage-Guided Fuzz Testing*. Accessed: 2023-10-18. 2019. URL: `https://llvm.org/docs/LibFuzzer.html`.

[99] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. «BugBench: Benchmarks for Evaluating Bug Detection Tools». In: *Workshop on the Evaluation of Software Defect Detection Tools*. 2005.

[100] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. «Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation». In: *26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2005.

[101] Ulrike von Luxburg. «A Tutorial on Spectral Clustering». In: *Statistics and Computing* 17 (2007).

[102] 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)*. 2019.

[103] David R. MacIver and Alastair F. Donaldson. «Test-Case Reduction Via Test-Case Generation: Insights from the Hypothesis Reducer (tool Insights Paper)». In: *34th European Conference on Object-Oriented Programming (ECOOP)*. 2020.

[104] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. «The Art, Science, and Engineering of Fuzzing: A Survey». In: *IEEE Transactions on Software Engineering* 47.11 (2019).

[105] Valentin J. M. Manès, Soomin Kim, and Sang Kil Cha. «Ankou: Guiding Grey-Box Fuzzing Towards Combinatorial Difference». In: *42nd IEEE/ACM International Conference on Software Engineering (ICSE)*. 2020.

[106] William M. McKeeman. «Differential Testing for Software». In: *Digital Technical Journal* 10.1 (1998).

[107] Raveendra Kumar Medicherla, Raghavan Komondoor, and Abhik Roychoudhury. «Fitness Guided Vulnerability Detection with Greybox Fuzzing». In: *42nd IEEE/ACM International Conference on Software Engineering (ICSE)*. 2020.

[108] Ghassan Misherghi and Zhendong Su. «HDD: Hierarchical Delta Debugging». In: *28th IEEE/ACM International Conference on Software Engineering (ICSE)*. 2006.

[109] MITRE. *Common Weakness Enumeration (CWE)*. Accessed: 2023-10-18. 2007. URL: `https://cwe.mitre.org`.

[110] David Molnar, Xue Cong Li, and David A. Wagner. «Dynamic Test Generation to Find Integer Bugs in X86 Binary Linux Programs». In: *18th USENIX Security Symposium (USENIX Security)*. 2009.

[111] Mozilla. *Crash Reports*. Accessed: 2023-10-18. 2012. URL: `http://crash-stats.mozilla.com`.

[112] Roberto Natella. «StateAFL: Greybox Fuzzing for Stateful Network Servers». In: *Empirical Software Engineering* 27.7 (2022).

[113] Roberto Natella and Van-Thuan Pham. «ProFuzzBench: A Benchmark for Stateful Protocol Fuzzing». In: *30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2021.

[114] Timothy Nosco, Jared Ziegler, Zechariah Clark, Davy Marrero, Todd Finkler, Andrew Barbarello, and W. Michael Petullo. «The Industrial Age of Hacking». In: *29th USENIX Security Symposium (USENIX Security)*. 2020.

[115] Pradeep Padala. *Playing with Ptrace, Part I*. Accessed: 2023-10-18. 2002. URL: `https://www.linuxjournal.com/article/6100`.

[116] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. «FuzzFactory: Domain-specific Fuzzing with Waypoints». In: *ACM on Programming Languages* 3.OOPSLA (2019).

[117] Karl Pearson. «On Lines and Planes of Closest Fit to Systems of Points in Space». In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901).

[118] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. «Scikit-Learn: Machine Learning in Python». In: *Journal of Machine Learning Research* 12 (2011). URL: `http://jmlr.org/papers/v12/pedregosa11a.html`.

[119] Sriram V. Pemmaraju and Steven S. Skiena. «Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica». In: Cambridge University Press, 2003. Chap. "Contracting Vertices" §6.1.1, pp. 231–234.

[120] Hui Peng and Mathias Payer. «USBFuzz: A Framework for Fuzzing USB Drivers by Device Emulation». In: *29th USENIX Security Symposium (USENIX Security)*. 2020.

[121] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. «T-Fuzz: Fuzzing by Program Transformation». In: *39th IEEE Symposium on Security and Privacy (SP)*. 2018.

[122] Theofilos Petsios, Adrian Tang, S. Stolfo, A. Keromytis, and S. Jana. «NEZHA: Efficient Domain-Independent Differential Testing». In: *38th IEEE Symposium on Security and Privacy (SP)*. 2017.

[123] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. «SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities». In: *24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017.

[124] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. «AFLNet: A Greybox Fuzzer for Network Protocols». In: *13th IEEE International Conference on Software Testing, Validation and Verification (ICST)*. 2020.

[125] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. «Model-Based Whitebox Fuzzing for Program Binaries». In: *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2016.

[126] Van-Thuan Pham, Sakaar Khurana, Subhajit Roy, and Abhik Roychoudhury. «Bucketing Failing Tests Via Symbolic Analysis». In: *20th International Conference on Fundamental Approaches to Software Engineering (FASE)*. 2017.

[127] Aashish Phansalkar, A. Joshi, L. Eeckhout, and L. John. «Measuring Program Similarity: Experiments with SPEC CPU Benchmark Suites». In: *5th IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2005.

[128] Sebastian Poeplau and Aurélien Francillon. «Symbolic Execution with SymCC: Don't Interpret, Compile!» In: *29th USENIX Security Symposium (USENIX Security)*. 2020.

[129] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. «Renaissance: Benchmarking Suite for Parallel Applications on the Jvm». In: *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2019.

[130] Ivan Pustogarov, Thomas Ristenpart, and Vitaly Shmatikov. «Using Program Analysis to Synthesize Sensor Spoofing Attacks». In: *12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. 2017.

[131] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. «NSFuzz: Towards Efficient and State-Aware Network Service Fuzzing». In: *1st International Fuzzing Workshop (FUZZING)*. 2022.

[132] Tim Rains. *Security Development Lifecycle: A Living Process*. Accessed: 2019-09-09. 2012. URL: `https://www.microsoft.com/security/blog/2012/02/01/security-development-lifecycle-a-living-process`.

[133] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. «VUzzer: Application-Aware Evolutionary Fuzzing.» In: *25th Network and Distributed System Security Symposium (NDSS)*. 2017.

[134] Peter J. Rousseeuw. «Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis». In: *Journal of Computational and Applied Mathematics* 20 (1987).

[135] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. «Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults». In: *26th ACM SIGSOFT Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2018.

[136] Korosh Koochekian Sabor, Abdelwahab Hamou-Lhadj, and Alf Larsson. «Durfex: A Feature Extraction Technique for Efficient Detection of Duplicate Bug Reports». In: *17th IEEE International Conference on Software Quality, Reliability, and Security (QRS)*. 2017.

[137] Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. «Density-Based Clustering in Spatial Databases: The Algorithm GDBSCAN and Its Applications». In: *Data Mining and Knowledge Discovery* 2 (1998).

[138] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. «Do Stack Traces Help Developers Fix Bugs?» In: *7th IEEE Working Conference on Mining Software Repositories (MSR)*. 2010.

[139] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. «DBSCAN Revisited, Revisited: Why and How You Should (still) Use DBSCAN». In: *ACM Transactions on Database Systems* 42.3 (2017).

[140] 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)*. 2021.

[141] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. «Nyx-Net: Network Fuzzing with Incremental Snapshots». In: *17th European Conference on Computer Systems (EuroSys)*. 2022.

[142] Kostya Serebryany. *OSS-Fuzz: Google's Continuous Fuzzing Service for Open Source Software*. 2017.

[143] Kostya Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. «AddressSanitizer: A Fast Address Sanity Checker». In: *USENIX Annual Technical Conference (ATC)*. 2012.

[144] Claude E. Shannon. «A Mathematical Theory of Communication». In: *Bell System Technical Journal* 27 (1948).

[145] Dongdong She, Abhishek Shah, and Suman Jana. «Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis». In: *43rd IEEE Symposium on Security and Privacy (SP)*. 2022.

[146] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. «SoK: (State of) the Art of War: Offensive Techniques in Binary Analysis». In: *37th IEEE Symposium on Security and Privacy (SP)*. 2016.

[147] Giannis Siglidis, Giannis Nikolentzos, Stratis Limnios, Christos Giatsidis, Konstantinos Skianis, and Michalis Vazirgiannis. «GraKeL: A Graph Kernel Library in Python». In: *Journal of Machine Learning Research* 21.1 (2020).

[148] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and M. Franz. «SoK: Sanitizing for Security». In: *40th IEEE Symposium on Security and Privacy (SP)*. 2019.

[149] Daan Sprenkels. *LLVM Provides No Side-Channel Resistance*. Accessed: 2020-02-13. 2019. URL: `https://dsprenkels.com/cmov-conversion.html`.

[150] Standard Performance Evaluation Corporation. *SPEC Benchmark Suite*. Accessed: 2020-02-12. URL: `https://www.spec.org`.

[151] Michael Steinbach, George Karypis, and Vipin Kumar. *A Comparison of Document Clustering Techniques*. Tech. rep. Department of Computer Scienceand Engineering, University of Minnesota, 2000.

[152] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. «Driller: Augmenting Fuzzing through Selective Symbolic Execution». In: *23rd Network and Distributed System Security Symposium (NDSS)*. 2016.

[153] Robert Swiecki. *Honggfuzz*. Accessed: 2023-10-18. 2016. URL: `http://honggfuzz.com/`.

[154] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. «Semantic Crash Bucketing». In: *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2018.

[155] Trail of Bits. *DARPA Challenge Binaries on Linux, Os X, and Windows*. Accessed: 2020-10-04. 2016. URL: `https://github.com/trailofbits/cb-multios`.

[156] Vipindeep Vangala, Jacek Czerwonka, and Phani Talluri. «Test Case Comparison and Clustering Using Program Profiles and Static Execution». In: *7th ACM SIGSOFT Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 2009.

[157] Jonas Benedict Wagner. «Elastic Program Transformations Automatically Optimizing the Reliability/performance Trade-Off in Systems Software». PhD thesis. EPFL, 2017.

[158] Jinghan Wang, Chengyu Song, and Heng Yin. «Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing». In: *28th Network and Distributed System Security Symposium (NDSS)*. 2021.

[159] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. «Skyfire: Data-Driven Seed Generation for Fuzzing». In: *38th IEEE Symposium on Security and Privacy (SP)*. 2017.

[160] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. «Superion: Grammar-Aware Greybox Fuzzing». In: *41st IEEE/ACM International Conference on Software Engineering (ICSE)*. 2019.

[161] Pengfei Wang, Xu Zhou, Kai Lu, Tai Yue, and Yingying Liu. «SoK: The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing». In: *Computing Research Repository* (2020).

[162] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. «Taintscope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection». In: *31st IEEE Symposium on Security and Privacy (SP)*. 2010.

[163] Zhiqiang Wang, Yuqing Zhang, and Qixu Liu. «RPFuzzer: A Framework for Discovering Router Protocols Vulnerabilities Based on Fuzzing». In: *KSII Transactions on Internet and Information Systems* 7.8 (2013).

[164] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. «MemLock: memory usage guided fuzzing». In: *42nd IEEE/ACM International Conference on Software Engineering (ICSE)*. 2020.

[165] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. «Scheduling Black-Box Mutational Fuzzing». In: *20th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2013.

[166] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. «CrashLocator: Locating Crashing Faults Based on Crash Stacks». In: *23rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 2014.

[167] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. «Postmortem Program Analysis with Hardware-Enhanced Post-Crash Artifacts». In: *26th USENIX Security Symposium (USENIX Security)*. 2017.

[168] 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)*. 2020.

[169] Li Yujian and Liu Bi. «A Normalized Levenshtein Distance Metric». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 29.6 (2007).

[170] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. «QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing». In: *27th USENIX Security Symposium (USENIX Security)*. 2018.

[171] Michał Zalewski. *American Fuzzy Lop (AFL)*. Accessed: 2023-10-18. 2015. URL: `http://lcamtuf.coredump.cx/afl`.

[172] Michał Zalewski. *Pulling JPEGs out of thin air*. Accessed: 2023-10-09. 2014. URL: `https://lcamtuf.blogspot.com/2014/11/pulling-jpegs-out-of-thin-air.html`.

[173] Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, and George Candea. «Automated Debugging for Arbitrarily Long Executions». In: *14th Workshop on Hot Topics in Operating Systems (HotOS)*. 2013.

[174] Andreas Zeller and Ralf Hildebrandt. «Simplifying and Isolating Failure-Inducing Input». In: *IEEE Transactions on Software Engineering* 28.2 (2002).

[175] Gen Zhang, Pengfei Wang, Tai Yue, Xiangdong Kong, Shan Huang, Xu Zhou, and Kai Lu. «MobFuzz: Adaptive Multi-objective Optimization in Gray-box Fuzzing». In: *29th Network and Distributed System Security Symposium (NDSS)*. 2022.

[176] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. «SeqFuzzer: An Industrial Protocol Fuzzing Framework from a Deep Learning Perspective». In: *12th IEEE International Conference on Software Testing, Validation and Verification (ICST)*. 2019.

Ahmad Hazimeh

ahmad@hazimeh.dev
+41 78 961 16 99
Lausanne, Vaud
Switzerland

PROFILE

- Systems security professional with 10+ years of programming experience and a strong background in software security and development
- Expertise in vulnerability research, penetration testing, reverse engineering, and firmware development
- Effective educator with experience teaching classes on Operating Systems and Software Security
- Proven track record of problem-solving, elegant software design, and reusable tooling
- Self-starter, committed to perfection, and skilled in learning complex concepts quickly

AREAS OF EXPERTISE

Software Development:	Design and architecture, Low-level systems programming, Firmware development
Cybersecurity Research:	Fuzzing, Symbolic execution, Vulnerability assessment, Reverse engineering
Computer Engineering:	Hardware design, Embedded systems
Other Technical Skills:	Networking, Tooling and automation, Containerization, RFID security

EDUCATION

École polytechnique fédérale de Lausanne — Lausanne, CH
Ph.D. in Systems Security; supervisor: Prof. Mathias Payer — *Feb 2019 – Mar 2024*

American University of Beirut — Beirut, LB
Bachelor of Engineering in Computer and Communications — *Sep 2014 – Jun 2018*

EXPERIENCE

BugScale — Lonay, CH
Vulnerability Researcher — *Dec 2023 – Present*

École polytechnique fédérale de Lausanne — Lausanne, CH
Doctoral Assistant — *Feb 2019 – Dec 2023*

- Developed the MAGMA benchmark [1] and tool suite for evaluating fuzzers
- Authored research on IGOR [2], a system for crash clustering and deduplication
- Designed and implemented TANGO [3], a custom fuzzing framework for stateful systems

Ubilite — Beirut, LB
Firmware Engineer — *Aug 2018 – Jan 2019*

- Developed firmware state machines for parsing and processing host commands using the QM™ framework
- Created a C library for SPI communication and a Java GUI to facilitate testing the SoC in different settings
- Identified and reported a stack buffer overflow vulnerability in beacon-parsing logic during on-boarding

University of Illinois at Urbana-Champaign — Illinois, US
Data Science Intern — *Summer 2017*

- Performed data mining and pattern recognition on behavioral patterns for authenticating mobile users

American University of Beirut — Beirut, LB
Research Assistant — *Spring 2017*

- Reverse engineered the lower layers of the WhatsApp security protocol, on top of the Noise Protocol Framework, in search for logical loopholes in state management and certificate validation

PUBLICATIONS

[1] **Ahmad Hazimeh**, Adrian Herrera, Mathias Payer.
MAGMA*: A Ground-Truth Fuzzing Benchmark.*
In: ACM SIGMETRICS. 2021. DOI: 10.1145/3428334
https://hexhive.epfl.ch/magma

[2] Zhiyuan Jiang, Xiyue Jiang, **Ahmad Hazimeh**, Chaojing Tang, Chao Zhang, Mathias Payer.
IGOR*: Crash Deduplication Through Root-Cause Clustering.*
In: ACM Conference on Computer and Communication Security. 2021. DOI: 10.1145/3460120.3485364.

[3] **Ahmad Hazimeh**, Duo Xu, Qiang Liu, Yan Wang, Mathias Payer.
TANGO*: Extracting Higher-Order Feedback through State Inference*
(Under submission)

Selected Projects

UltraBroken: A MITM attack on the one-way counter in Mifare Ultralight C cards with a Proxmark and a ChameleonTiny
PrimeFactory: CFRAC-based factorization with CUDA support in C++
LiveStonks: Fetching live stock tickers and financial information via Azure Functions into Google Sheets
Vulnerability Assessments: Identified and reported vulnerabilities in local online shops, education platforms, and SQL databases

Notable Accomplishments

Graduated with Distinction Beirut, LB
GPA: 4.0 *2018*

Google Hash Code - Online Quals Beirut, LB
Ranked in 2nd place *2017*

Highschool National Qualification Exams Beirut, LB
Ranked in 1st place, with an unprecedented score *2011*

Languages and Frameworks

Proficient: C, C++, Python, Bash, LaTeX
Familiar: Scala, Java, Rust, PineScript, JavaScript, VB.NET, C#, VHDL, Verilog, HTML, CSS
Frameworks: Python Standard Libraries, LLVM/Clang, Linux Kernel, QM™ Framework
Technical Toolset: GDB-gef, Ghidra, IDA, AFL, libFuzzer, angr, KLEE, SymCC