FUZZING WITH ADVANCED PROGRAM EXPLORATION AND BUG MODELING FOR SOFTWARE SECURITY

A Dissertation
Presented to
The Academic Faculty

By

Yongheng Chen

In Partial Fulfillment
of the Requirements for the Degree
in the
College of Computing
School of Cybersecurity and Privacy

Georgia Institute of Technology

August 2024

FUZZING WITH ADVANCED PROGRAM EXPLORATION AND BUG MODELING FOR SOFTWARE SECURITY

Thesis committee:

Dr. Wenke Lee (Advisor)
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Daniel Genkin
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Saman Zonouz
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. László Szekeres
Google Research
Google

Dr. Brendan Saltaformaggio
School of Cybersecurity and Privacy
Georgia Institute of Technology

Date approved: May 6, 2024

The people who are crazy enough to think they can change the world are the ones who do.

Steve Jobs

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Wenke Lee. He is a man of intelligence, honesty, integrity, motivation and insight, and supported me through my entire graduate study. I would also like to acknowledge my thesis committee, Dr. Saman Zonouz, Dr. Brendan Saltaformaggio, Dr. Daniel Genkin and Dr. László Szekeres, for their invaluable comments and suggestions on my dissertation.

I am fortunate to work with many great collaborators, including Dr. Hong Hu, Dr. Dinghao Wu, Hanfan Zhang, Yupeng Yang, Jizhou Chen, Kyuhong Park, Evan Downing, Burak Sashin. Dr. Hong Hu taught me hand-by-hand how to conduct research and how to write papers during my early PhD career, and I am grateful for his guidance and friendship. I want to show my special thanks to Dr. Rui Zhong, a man that I can trust unconditionally. We have each other's back and have been through thick and thin together during our PhD journey.

I would like to thank my parents, Yuxing Chen and Meizhen Tao, for their endless love and support during these years. And finally, I want to thank my fiancée, Yuanling Chen, for her support, patience, understanding, encouragement and everything.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Fuzzing is a well-received software testing technique. It operates by generating random inputs and then executing these against a given target program, thus probing various program states to pinpoint anomalies. Despite its proven utility, fuzzing has its limitations. Like other dynamic testing methods, it struggles with inadequate exploration of the program state space. This limitation stems from issues such as the unstructured nature of the generated inputs and the inefficient use of computational resources across multiple cores. A more critical shortcoming of traditional fuzzing lies in its approach to bug modeling: it primarily detects bugs through program crashes, overlooking a myriad of bugs that do not crash the program execution but are equally consequential. While the development of dedicated oracles represents a stride toward refined bug modeling, this solution is often impractical due to the high costs associated with crafting oracles that are typically bug-specific or tailored to individual programs.

To address these limitations, we propose two-dimensional improvements, which scales the program exploration capability and enhances bug modeling in fuzzing. To explore more program states, we propose POLYGLOT and μFUZZ to scale the program exploration capability vertically and horizontally. Specifically, POLYGLOT utilizes a unified intermediate representation to handle diverse programming languages, effectively generating semantically valid inputs that result in deeper program exploration, finding over 170 new bugs in 21 language processors. μFUZZ, on the other hand, employs a microservice architecture to maximize the efficiency of parallel fuzzing, reducing synchronization overhead and enhancing the utilization of computational resources. More importantly, μFUZZ found 11 new bugs in well-tested popular programs. To enhance bug modeling, we introduce PROPGUARD, a framework that enables the specification and automatic detection of a wide range of bug patterns, moving beyond mere crash detection to identify subtle, non-crashing bugs. By allowing users to define bug patterns through an intuitive specification language, PROPGUARD

facilitates the development of targeted fuzzing oracles, thus significantly broadening the spectrum of detectable software vulnerabilities and finding two new non-crashing issues in open-source projects.

CHAPTER 1
INTRODUCTION

Fuzzing has shown its effectiveness in finding software bugs and vulnerabilities [1, 2, 3, 4, 5]. It usually demands minimal manual effort and little knowledge of the target software, earning widespread adoption among security researchers and developers. For example, OSS- Fuzz, an open-source fuzzing service provided by Google that integrates various state-of-the-art fuzzers, has found over 46,000 bugs and vulnerabilities across 1,000 of open-source projects [4].

However, existing fuzzing techniques have limitations which can be categorized in two dimensions: **limited program state space exploration** and **lack of bug modeling**. Modern software systems, characterized by their intricacy and demanding input requirements, often hinder fuzzers due to their inability to produce appropriately structured inputs to efficiently explore vast state spaces [6, 7, 8, 9]. For instance, compilers necessitate inputs that adhere to precise syntactical and semantic standards. A syntactical or semantic error in the input source program may cause the compilation process to end at an early stage. Randomly generated inputs by traditional fuzzing techniques often fail to pass the checks and thus cannot explore the deep program states effectively. Various approaches have been proposed to use input grammars to guide the input generation process and produce syntactically correct inputs, but they fail to generate semantically correct inputs. Recent work tries to further integrate the semantics modeling of the inputs so that the generated inputs are both syntactically and semantically correct [9, 8]. However, these approaches are often limited to specific programs and cannot be easily extended to other programs at a low cost. Moreover, the expansive state spaces characteristic of contemporary programs exceed the exploration capabilities of a single fuzzer within a reasonable timeframe. Practitioners propose using multiple cores to parallelize the fuzzing process, but existing parallel fuzzing techniques simply aggregate the results from multiple single-core fuzzing instances [2, 10, 11]. These instances collaborate by periodic synchronization on the fuzzing seed corpus to share the fuzzing progress. However, these fuzzers are designed to run synchronously, such synchronization can be either expensive or untimely, resulting in under-utilization of computation power and limited improvement in program state space exploration.

Critically, traditional fuzzing methods primarily detect anomalies through the occurrence of crashes, neglecting a wide array of non-crashing yet significant bugs such as command injection and path traversal [12, 2, 13]. Without modeling these bugs, even if a fuzzer successfully explores a program state that contains such bugs, it cannot identify them. Developing dedicated oracles to identify non-crashing bugs helps, but these oracles are often bug-specific or even program-specific. Considering the cost of developing oracles and the number of different types of bugs, it is impractical to develop oracles for every bug type.

1.1 Problem Statement and Our Approach

This thesis research studies the problem of:

How to improve the bug detection performance of fuzzing in a systematic and scalable way?

We seek a systematic approach so that we can address the limitations in two dimensions, namely bug searching (i.e., program state space exploration) and bug modeling. We pursue a scalable way so that the proposed solutions can be universal and easily applied to a wide range of situations. For bug searching, we improve it both vertically and horizontally. Vertically, we propose a generic approach to model the syntax and semantics of the input format of a program with a few lines of user annotations as the guidance of the input generation process, which allows the fuzzer to generate semantically valid inputs to reach deep program states. Horizontally, we redesign traditional parallel fuzzing with microservice architecture that allows multiple fuzzing instances to run asynchronously and share the fuzzing progress timely and efficiently, achieving a better collaborative fuzzing process.

Users can easily integrate with the new framework by porting their algorithms as services. For bug modeling, we design a specification language that allows users to describe the code pattern for different types of bugs with minimal effort. Such bug specifications are then automatically translated into fuzzing oracles and thus fuzzers can detect bugs other than crashes.

1.2 Thesis Contributions

This thesis substantiates the feasibility and efficiency of our introduced methodologies through the development and assessment of three advanced fuzzing frameworks. These frameworks overcome existing limitations in bug discovery and modeling, as detailed below:

We first present POLYGLOT, which can generate semantically valid test cases to extensively test processors of different programming languages. To achieve generic applicability, we design a uniform IR to neutralize the difference in the syntax and semantics of programming languages. Given the BNF (Backus-Naur form) grammar [14] of a language, POLYGLOT can generate a frontend that translates source programs into this IR. At the same time, users can provide semantic annotations to describe the specific semantics about the scopes and types of definitions of the language. To achieve high language validity, we develop two techniques, the constrained mutation and the semantic validation, for test case generation. These techniques retain the syntactic correctness during mutation and fix semantic errors afterwards, resulting in a high language validity of the generated test cases. More importantly, we evaluated POLYGLOT on 21 popular language processors (*e.g.*, v8 javascript engine, the php interpreter) of nine programming languages and have successfully identified 173 previously unknown bugs, 18 of which are assigned CVEs. Compared with state-of-the-art fuzzers, POLYGLOT achieves up to 30 times more code coverage in 24 hours.

Next, we present a parallel fuzzing framework in microservice architecture called μFUZZ, enhancing multicore utilization through concurrent execution and fuzzing state partitioning.

By deconstructing the synchronous fuzzing process into asynchronous services, μFUZZ minimizes blocking I/O's impact, thus efficiently utilizing CPU cycles. With a dual-level state partitioning design, μFUZZ minimizes synchronization delays, allowing workers to access up-to-date information for better decision-making. Comparative analysis showed μFUZZ achieving a 24% increase in new coverage and a 33% boost in bug discovery over top parallel fuzzers. Furthermore, μFUZZ identified 11 new bugs in extensively fuzzed applications, demonstrating its superior efficiency.

Finally, we present a fuzzing framework called PROPGUARD, which simplifies the development of effective fuzzing oracles. It streamlines bug specification, tracking, and oriented searching, enabling users to describe and locate specific bug patterns with minimal effort. More specifically, we create an expressive specification language for users to describe the bug patterns, with which users can specify the bug pattern in just a few lines of specification. We then convert the bug specification into a fuzzing searchable form by constructing a finite state machine (FSM) that models the specification and then instrumenting the target program to track the state transition of the state machine. Finally, we perform directed fuzzing to search for the specified bug patterns by guiding the fuzzer with the bug specification. PROPGUARD effectively identifies both crashing and non-crashing bugs. Our evaluation confirmed the capability to detect seven out of eight bug types, including two previously unknown non-crashing bugs in real-world applications.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter §2 reviews the related work in the field of fuzzing and discusses the limitations of existing techniques. Chapter §3 discusses how we improve input generation of fuzzing and presents POLYGLOT, a fuzzing framework that can generate valid test cases for language processors of different programming languages. Chapter §4 presents μFUZZ, a novel parallel fuzzing architecture that can effectively utilize multiple cores to improve the program state space exploration. Chapter §5 presents

PROPGUARD, a framework that allows users to develop fuzzing oracles with minimal effort in a high-level specification language. Finally, Chapter §6 concludes the thesis and discusses future work.

CHAPTER 2

RELATED WORK

In this section, we delve into existing works related to fuzzing enhancement in terms of program state space exploration and bug modeling. Section §2.1 discusses vertical scaling, which focuses on improving the components of a single fuzzer. Section §2.2 explores horizontal scaling, which aims to enhance the fuzzing process by parallelizing multiple fuzzers. Section §2.3 reviews bug modeling, which focuses on improving the bug identification capability of fuzzers.

2.1 Vertical Fuzzing Improvement

Vertical fuzzing improvement focuses on enhancing the fuzzing process itself, including input generation, feedback guidance, seed selection, and test case execution.

2.1.1 Input Generation

Existing improvements on the input generation can be mainly categorized into generation-based [15, 16, 8, 17] and mutation-based [12, 18, 19].

Generation-based. Generation-based fuzzing can effectively test software that require structural inputs, such as compilers and document viewers [17, 7, 8, 20, 21, 22]. They usually generate test cases from a model or grammar and thus guarantee the syntactic correctness [7, 23, 24]. MoWF [21] shows how to use file format information as the model to fuzz the code beyond the parser. SQLsmith [7] generates SQL queries utilizing SQL grammar and database schemas. It collects the schemas from the initial databases and generates limited types of queries, like `SELECT`, to ensure that the database is unchanged. Bikash Chandra *et al*. [25] proposed a way to generate initial databases that can cover most types of SQL queries. More advanced generation-based fuzzers [8, 26, 27] further model the semantics of the inputs to ensure the semantic correctness. CodeAlchemist [27] proposes semantics-aware assembly to synthesize semantics-correct `JavaScript` test cases. CSmith [8] specializes its analysis for C semantics and produces completely correct test cases. Vikram *et al.* [28] proposes Bonsai fuzzing, a coverage-guided grammar-based technique that synthesizes concise test inputs for educational compilers by growing them iteratively, offering a more efficient alternative to minimizing complex inputs. Dewey *et al.* uses constraint logic programming to specify syntactic features and semantic behaviors in test case generation [26, 29], Still, ensuring the semantic correctness can be difficult. QAGen shows that ensuring perfect semantic correctness is an NP-complete problem [30]. Instead, it provides an approximate solution to improve the semantic correctness. Several works reduce the generation to the SAT problem [31, 32] and use an SAT solver (*e.g.*, Alloy [33]) to provide potential solutions. Sometimes, it can be nontrivial to get the model or grammar. For example, the tested application is closed-source and has no public documents. Recent works propose methods to infer the structures of the inputs by static analysis or machine learning on an initial seed corpus [34, 35, 36, 37]. Viide *et al.* [34] proposes a model inference approach to assist fuzzing. Osbert *et al.* [35] utilizes a set of test cases and the black-box access to the tested binary to construct a context-free grammar of the language. AUTOGRAM [36] uses dynamic taint analysis to produce readable and accurate input grammars. The learning-enabled fuzzer NEUZZ [38] leverages a surrogate neural network to smoothly approximate the branching behavior of the program in order to generate useful test cases.

Mutation-based. Mutation-based fuzzing is recognized for its depth in probing program logic, relying on an initial corpus and mutating existing cases to uncover new execution paths [12, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48]. Unlike generation-based ones, mutation-based fuzzers usually require an initial corpus to run. They perform random mutation on existing test cases to generate new ones. If a test case triggers a new execution path, it will be considered as useful and saved for further mutation. In this way, fuzzers quickly reach the deep logic and explore more program states. AFL [12] adopts coverage feedback as guidance and performs random bitflip mutation. Wu *et al.* [48] propose a way to dynamically adjust the mutation strategy by modeling it as a multi-armed bandit problem to be solved. Xu *et al.* [49] introduces cooperative mutation, a technique that concurrently alters script code and program native input to expose bugs in binding code, utilizing a two-dimensional mutation strategy and three innovative methods: clustering objects by semantic similarity, inferring script-object relationships from extensive executions, and using these relationships for targeted, effective mutations in scripting languages. FairFuzz [50] identifies seldom-exercised branches (rare branches) and employs a dynamic mutation mask creation algorithm, tailored to generate inputs that target these branches, thus uncovering bugs in previously unexplored code regions. EMS [19] leverages the Probabilistic Byte Orientation Model (PBOM) to enhance byte-level mutation strategies based on intra- and inter-trial histories, efficiently guiding mutations to stimulate unique execution paths and uncover crashes by probabilistically determining optimal mutation byte values. Rainfuzz [51] leverages reinforcement learning to create heat-maps for input mutation and formalizes the selection of mutation positions as a reinforcement learning problem, demonstrating superior effectiveness in directing mutations compared to random-position fuzzing. BeDivFuzz [52] emphasizes not just the quantity of different branches covered (richness) but also the distribution frequency (evenness), advocating for behavioral diversity by using structure-preserving and structure-changing mutations to generate syntactically valid inputs that trigger branches evenly and diversely. DARWIN [53] optimizes and adapts mutation operator probabilities without extra user-configurable parameters with an evolution strategy. As naive bitflip mutation can hardly pass complicated checks such as magic numbers, existing fuzzers [40, 54, 42, 55, 56, 57, 58, 43] adopt symbolic execution or taint analysis to overcome the problem. Chen *et al.* [59] proposes a solution to tackle challenge with deeply nested conditional statements by identifying control and taint flow-dependent conditionals, then applying three strategic methods to concurrently satisfy all identified conditional statements. Driller [43] performs selective concolic execution while QSYM [40] integrates the symbolic emulation with the native execution. Redqueen [42] avoids heavy taint tracking by inferencing the input-to-state correlation and then uses it to guide the mutation to overcome common fuzzing roadblocks. T-Fuzz [60] further proposes a way to dynamically transform the program in order to remove certain checks that are hard for the fuzzer to bypass successfully. SGXFUZZ [61] detects vulnerabilities in SGX enclaves via a novel input synthesis method and an enclave runner, allowing effective fuzzing without source-code access by dynamically constructing expected input formats and simulating real-world interactions. However, they are implemented as general fuzzers and are unaware of the syntactic structure of inputs. Therefore, researchers propose to combine the advantages of generation-based and mutation-based fuzzing [62, 63, 64, 65, 66, 67, 68]. Tim Blazytko *et al.* [63] proposed a way to utilize grammar-like combinations to synthesize highly structured inputs, but most of its generated test cases in SQL are still syntactically incorrect. Salls *et al.* proposes Token-Level Fuzzing that operates mutations at the token level, enabling evolutionary fuzzers to produce both grammatically conforming and non-conforming inputs, thereby enhancing the ability to parse and assess target applications effectively. Gramatron [69] leverages grammar automatons and aggressive mutation operators and addresses sampling bias by restructuring grammars for unbiased input generation. Mathis *et al.* [70] proposes a test generation technique that targets input parsers by systematically producing inputs, monitoring comparisons, and adapting after rejections to satisfy the conditions that led to rejection, effectively ensuring comprehensive coverage of the input space. ProFuzzer [71] dynamically identifies and understands crucial input fields during fuzzing, adapting mutations intelligently without needing input specifications. By analyzing mutation outcomes to correlate and type-define related bytes, it applies type-specific strategies for focused mutation and thus improves fuzzing effectiveness. Hardik Bati *et al.* [64] proposed to mutate SQL statements by adding or removing grammar components. DIFUZE [62] leverages static analysis to identify the input structures of kernel drivers. SLF [67] infers the relation between input validity checks and input fields to generate valid seed inputs. Nautilus [65] adopts code coverage as feedback to decide whether to keep its generated test cases for mutation. These fuzzers can likely preserve the syntactical correctness, but cannot guarantee the semantic correctness of the inputs. Recent works further model the semantics of the inputs and try to preserve them during the mutation [6, 9, 72, 73, 74]. LangFuzz [73] and Superion [75] accept a grammar to translate test cases to AST and then mutate the AST. Unicorn [76] combines hybrid input synthesis for generating valid, feature-rich queries and proactive exception detection to uncover subtle bugs concealed by implicit exception handling mechanisms in time-series databases. Fuzzilli [72] and Squirrel [6] design their own IRs for mutation and semantic analysis in `JavaScript` and SQL respectively. SoFi [77] ensures test case validity through detailed program analysis to identify and infer variable types for mutations and introduces an automatic repair strategy to correct any syntax or semantic errors in generated Javascript test cases. Favocado [78] efficiently generates valid test cases and reduces input space by classifying DOM objects into equivalence classes, enhancing the likelihood of exposing binding code vulnerabilities in JavaScript engines. Montage [79] utilizes neural network language models trained by AST transformation to guide the input generation. DIE [9] advocates a new technique, called an aspect-preserving mutation, that stochastically preserves the desirable properties to generate semantically correct test cases. GrayC [80] develops a new set of mutations to target common C constructs to produce test cases that compile.

2.1.2 Feedback Guidance

Feedback guidance plays a crucial role in the effectiveness of fuzzing as it directs the fuzzing process toward more fruitful areas of the target software, considering the huge input space of real world programs. Code coverage is the most widely used feedback guidance in fuzzing [12, 2, 6, 9]. The fuzzer instruments the target program to collect the code coverage by randomly assigning a unique identifier to each basic block and favors the test cases that can reach new code regions. However, the way to collect the coverage information suffers from path collision, where different program paths are merged into the same coverage info. InstruGuard [81] addresses common instrumentation errors by using static analysis to detect inaccuracies in target binaries and applying binary rewriting for corrections. CollAFL [82] provides more accurate coverage information by mitigating path collisions in AFL. Wang *et al*. [83] evaluates several coverage metrics, including "branch coverage", "context-sensitive coverage", and "n-gram branch coverage", and "memory-access-aware branch coverage", and shows that more sensitive coverage metrics can guide the fuzzer to deeper bugs but also increase the overhead. Some fuzzers [58, 84, 55, 54, 42, 46] use taint analysis to incorporate data flow information into their coverage metrics. PATA [85] further proposes a path-aware taint analysis by distinguishing between multiple occurrences of the same constraint. To fully utilize computation power, researchers try to find a better feedback guidance other than naive code coverage [86, 87, 47, 50]. Apollo [**apollo**] measures the difference in execution time to favor generated SQL queries. AFLGo [47] introduces directed greybox fuzzing with the objective of reaching a given set of target program locations efficiently. TortoiseFuzz [87] considers the security impacts indicated by code coverage to prioritize inputs. TaintScope [88] uses checksum as feedback guidance to help fuzz file segments. SAVIOR [44] prioritizes its concolic execution towards the locations with potential vulnerabilities. Ijon [89] annotates the data that represent the internal program states to guide the fuzzer.

2.1.3 Seed Selection & Scheduling

State-of-the-art mutation-based fuzzing starts with a seed corpus, whose quality can have great impacts on the fuzzing performance [90, 91]. MoonShine [90] distills seeds for OS fuzzers from system call traces of real-world programs while still maintaining the dependencies across the system calls by leveraging light-weight static analysis for efficiently detecting dependencies across different system calls. Drifuzz [91] introduces a concolic-augmented fuzzing approach for WiFi and Ethernet drivers, designed to enhance OS-peripheral security by generating 'golden seeds', which are seeds that overcome complex driver initialization code. Seed scheduling prioritizes which test cases to use for further mutations or analysis, aiming to maximize the effectiveness of program exploration and vulnerability detection with limited resources such as CPU power and fuzzing time [92, 93, 86, 94, 95]. AFLFast [86], MOPT [94], DigFuzz [96] collect information about the test cases and prioritize those with higher potential to reach new code regions. StateFuzz [95] models program states using state-variable values, employs static analysis for variable recognition, and uses state information with a three-dimensional feedback mechanism to enhance test case prioritization. She *et al*. [92] proposes K-Scheduler, a seed scheduling strategy for fuzzing that utilizes centrality measures from graph analysis to estimate the potential of a seed to reach unvisited edges in a program's control flow graph. This approach, embodied in an edge horizon graph connecting seeds to nearby unvisited nodes, assesses seed value based on reachability, mutation history, and proximity to unexplored areas, effectively improving edge coverage.

2.1.4 Test Case Execution

Fuzzing is a process of trying different test cases on the target program. Improving speed allows fuzzers to run more executions in the same amount of time with the same fuzzing strategy [97, 98, 99, 100, 101, 102, 103, 104]. Various techniques [98, 99, 105, 106] have been proposed to improve the instrumentation of the target program to reduce its overhead. Advanced generic fuzzers like `AFLplusplus` and `libfuzzer` [2, 39] utilizes persistent loop, which keeps executing a certain amount of test cases before restarting fuzzing process. Alternatively, SnapFuzz and SNPSFuzzer [107, 108] take snapshots of the process and utilize fast snapshot restore instead of slow forking. This reduces the overhead of forking a new process at each test case execution. INSTRIM [109] demonstrates that strategic instrumentation of a minimal number of basic blocks, leveraging common program structures and compiler optimizations, can accurately capture execution path coverage. INSTRCR [110] focuses on indirect jumps and frequently executed paths and optimizes instrumentation, thereby increasing coverage accuracy and reducing overhead. Nagy et al. [99, 105] proposes coverage-guided tracing to trace code coverage only when new ones are discovered. Zeror [111] enhances fuzzer efficiency by combining a self-modifying tracing mechanism for zero-overhead coverage collection with a real-time scheduling mechanism that adaptively toggles between zero and full instrumentation. RIFF [112] precomputes certain analyses at instrumentation-time via static analysis, reducing runtime instrumentation. Odin [98] adopts dynamic recompilation to prune necessary instrumentation on the fly. RetroWrite [106] uses static binary rewriting to support high-speed coverage-guided binary-only fuzzing with an efficient binary-only Address Sanitizer. Researchers have also explored hardware-assisted feedback-collecting mechanisms. kAFL [113], Honggfuzz [114], and PTrix [115] utilize Intel's *Processor Trace* technology, which enables them to efficiently collect coverage feedback with minimum overhead. Another well-explored topic is to improve the symbolic execution speed for hybrid fuzzing. Qsym [40] implements a symbolic execution engine tailored for fuzzing. Instead of translating the instructions to the intermediate representation and then executing them symbolically, Qsym tightly integrates the symbolic emulation with the native execution. SymCC [116] generalizes the idea of Qsym and presents a compiler that builds concolic execution right into the binary. In this way, the symbolic execution engine can run natively without any interpretation. Furthermore, utilizing QEMU, SymQEMU [117] modifies the IR of the target program before it gets translated into the host architecture, which enables compiling symbolic execution capabilities into the binary without access to its source code. On the other hand, sometimes it is not even trivial to execute the test case without a specialized environment, so researchers propose various approaches to provide such environments. Unicorefuzz [118] explores the use of CPU emulation to fuzz arbitrary parsers in kernel space with coverage-based feedback. Razzer [119] employs a deterministic thread interleaving techniques at the hypervisor level to make test case execution in kernel deterministic to find race condition bugs. V-Shuttle [120], a novel framework for hypervisor fuzzing, offers scalable and semantics-aware testing by introducing a DMA redirection mechanism for automated and efficient virtual device protocol reconstruction.

EQUAFL [121] proposes user-mode emulation for fuzzing embedded applications. It meticulously monitors the application in full-system mode to identify critical transition points and then replicates the necessary execution environment, allowing for user-mode emulation that accurately mimics system calls and resource management, thus optimizing the fuzzing process.

2.2 Horizontal Fuzzing Improvement: Parallel Fuzzing

Parallel fuzzing significantly enhances the efficiency and coverage of the fuzzing process by distributing the workload across multiple computing resources, thereby accelerating bug discovery and improving the utilization of available hardware in limited time budget. Existing works improve the performance of parallel fuzzing by either improving the fuzzing strategy [11, 122, 123, 10, 124, 125] or improving the fuzzing speed [126]. One popular way to improve the fuzzing strategy is task partitioning. PAFL [122] proposes an efficient guiding information synchronization method and statically divides fuzzing tasks based on branching information to reduce the overlap between instances. AFLEdge [11] further utilizes static analysis to dynamically create mutually exclusive and evenly weighted fuzzing tasks. AFLTeam [10] proposes a dynamic, graph-based task allocation approach for parallel fuzzing by optimally partitioning and prioritizing program exploration spaces. ClusterFuzz [127] is the distributed fuzzing infrastructure behind OSS-Fuzz, and automatically executes different fuzzers on scale. ClusterFuzzlite [128] is a lightweight version of ClusterFuzz, mainly used for continuous integration and continuous deployment (CI/CD). MicroFuzz [129], a specialized fuzzing framework for Microservices, proposes Mocking-Assisted Seed Execution, Distributed Tracing, and Pipeline Parallelism to navigate the unique challenges in Microservices environments. glibFuzzer [130], a parallel fuzzing tool, employs a group-based corpus scheduling strategy to avoid the conflicts in task division and corpus allocation. UltraFuzz [131], a distributed fuzzer focusing on resource efficiency, leverages centralized dynamic scheduling to optimally dispatch tasks and allocate computing power, minimizing resource waste and preventing seed evaluation bottlenecks in the fuzzing process. Another way to improve the fuzzing strategy is to combine the capabilities of different fuzzers, which is also called ensemble fuzzing [125] or collaborative fuzzing [132, 133]. The main idea is that different fuzzers might have different strengths on different targets. We can fuzz the same target with different fuzzers and share their fuzzing progress to let them help each other and achieve an overall better performance. EnFuzz [125] designs three heuristics for evaluating the diversity of existing fuzzers and choosing the most diverse subset to perform ensemble fuzzing through efficient seed synchronization. Collabfuzz [133] allows users to express different test case scheduling policies and use that to guidance the collaborative fuzzing process. Cupid [132] further proposes a collaborative fuzzing framework that can automatically discover the best combination of fuzzers for a target. One well-known problem of parallel fuzzing is the bottleneck of the underlying operating system. Xu et al. [126] found that the fuzzing performance can significantly degrade when running with multiple cores due to the file system contention and the scalability of the `fork` system call. Thus, they proposed three new operating primitives that allow much higher scalability and performance for parallel fuzzing. The current state-of-the-art fuzzers [2, 39, 114] support persistent fuzzing mode, which reuses the same process for multiple test cases to reduce the overhead of forking. Moreover, in-memory test cases [2] are also adopted to reduce the I/O overhead and file system contention.

2.3 Bug Modeling With Fuzzing Oracle

Bug modeling is the process of representing the characteristics of a bug. It is important for fuzzing to detect bugs other than crashes, because without bug modeling, fuzzing cannot decide whether a program state is buggy even if it has reached that state.

Differential Fuzzing. One popular technique to model bugs is differential fuzzing is a technique. It compares the behavior of different versions of a program. If the outputs of two versions of a program are different, it indicates that there is a bug in one of the versions. Chen *et al*. [134] propose using predefined mutation operators and Markov Chain Monte Carlo sampling to efficiently generate and select representative classfiles for differential testing of JVM startup processes, aiming to identify defects across various JVM implementations. Following up, Classming [135] introduces live bytecode mutation to generate valid and executable bytecode mutants for deep, differential testing of JVMs, focusing on producing diverse, semantically varied mutants effectively. NEZHA [136], an input-format-agnostic differential testing framework, tackles the inefficiencies of existing tools by exploiting behavioral asymmetries between programs, focusing on inputs that are likely to reveal semantic bugs, and thereby enhancing the effectiveness of differential testing. Cao *et al*. [137] propose differential replay to detect uninitialized variable usage by recording and replaying program executions with variations in memory initialization, comparing states to identify discrepancies, and using symbolic taint analysis to pinpoint the allocation locations, aiding in root cause analysis and exploit development. DifFuzz [138] identifies time and space side-channel vulnerabilities by comparing resource usage between two program versions and employing resource-guided heuristics to discover inputs that accentuate resource consumption disparities on secret-dependent execution paths, applicable across various programming languages. DPIFuzz [139] systematically identifies elusion strategies against stateful Deep Packet Inspection (DPI) systems for QUIC by generating and mutating Quick UDP Internet Connections (QUIC) streams to reveal discrepancies in server-side interpretations. DIFUZZRTL [140] models RTL characteristics such as cycle-sensitive guidance and asynchronous event handling to perform robust testing across diverse CPU register-transfer level (RTL) configurations. Duo [141] utilizes mutation-based fuzzing for tensor input generation and differential testing to assess the correctness of outputs across various operator instances of deep learning models. SEDiff [142] introduces a novel algorithm for mapping functionalities to implementation code, utilizing scope-aware grey-box differential fuzzing along with a specialized input generator and bug checker to efficiently and accurately identify erroneous inconsistencies. Bernhard *et al*. [143] a novel fuzzing technique that leverages the intrinsic relationship between a JavaScript engine's interpreter and JIT compiler as a domain-specific oracle, enabling highly sensitive detection of subtle logic bugs and miscomputations in JIT optimization processes. SQLRight [144] design a set of general APIs to decouple the logic of fuzzers and oracles, so that developers can easily port fuzzing tools to test database management systems and write new oracles for existing fuzzers.

Specialized Fuzzing Oracles. To further extend the types of bugs detectable by fuzzing, researchers have proposed various bug modeling techniques (*i.e.*, the fuzzing oracles) to detect non-crashing bugs [145, 146, 147, 148], especially for the popular and security-critical types of bugs. Notable examples include `Address Sanitizer`, which detects memory corruption bugs, and UndefinedBehavior Sanitizer, which detects undefined behavior [145]. Witcher [146] implements the concept of fault escalation to detect both SQL and command injection vulnerabilities. Additionally, Witcher captures coverage information and creates output-derived input guidance to focus the input generation. Rigger *et al.* [149] presents Pivoted Query Synthesis, a novel approach for identifying logic bugs in DBMSs, which generates queries guaranteed to fetch a designated 'pivot row'; failure to retrieve this row indicates potential logic errors within database management system. FuzzJIT [150] uncovers JIT compiler bugs in JavaScript engines by ensuring test cases activate the JIT compiler and detect any result discrepancies, with a unique input wrapping template that triggers JIT compilation and integrates self-verifying oracles to identify inconsistencies in execution outcomes. RoboFuzz [151], a feedback-driven fuzzing framework for Robot Operating System, enhances testing of robotic systems by incorporating an oracle handler that identifies correctness bugs by checking the execution states against predefined correctness oracles. ContractFuzzer [152] assesses security vulnerabilities by generating inputs from ABI specifications, employing test oracles for vulnerability detection, instrumenting the Ethereum virtual machine for behavioral logging, and analyzing these logs to identify and report security flaws. CorbFuzz [153], a web browser security policy fuzzer, evaluates CORB and related policies by leveraging a policy oracle to compare expected web application behaviors against actual browser responses, using symbolic evaluation for enhanced coverage and type information from database queries to ensure data consistency during fuzzing. Recently, researchers have also proposed to use specification for bug modeling in fuzzing [147, 148]. Kim et al. [148] propose PGFuzz, a policy-guided fuzzing framework, which validates whether a robotic vehicle adheres to identified safety and functional policies that cover user commands, configuration parameters, and physical states. LTL-fuzzer [147] utilizes linear temporal logic (LTL) to witness arbitrarily complex event orderings and finds violations, or counterexample traces, of the LTL property in stateful software systems.

CHAPTER 3
POLYGLOT: A GENERIC AND EFFECTIVE LANGUAGE FUZZING FRAMEWORK

In this chapter, we present POLYGLOT, a language fuzzing framework that generates semantically valid test cases and can be easily extendable to support multiple programming languages, scaling the program exploration capability vertically.

3.1 Introduction

Language processors [154], such as compilers and interpreters, are indispensable for building modern software. They translate programs written in high-level languages to low-level machine code that can be understood and executed by hardware. The correctness of language processors guarantees the consistency between the semantics of the source program and the compiled target code. Buggy language processors can translate even correct programs to malfunctional codes, which might lead to security holes. For example, miscompilation of memory-safe programs produces memory-unsafe binaries [155, 156]; vulnerabilities in interpreters enable attackers to achieve denial-of-service (DoS), sandbox escape, or remote code execution (RCE) [9, 157, 158]. Even worse, these defects affect all translated programs, including other translated language processors [159].

However, it is nontrivial for traditional software testing techniques to automatically detect bugs in language processors, as the processors pose strict requirements on their inputs regarding the syntactic and semantic validity. Any error in the program can terminate the execution of the language processor and hinder the tester from reaching deep translation logic.

Recent works on software testing, such as grey-box fuzzing, try to meet these requirements to effectively test language processors [12, 4, 160, 40, 114, 161]. Originally, structure-unaware mutation [12, 40, 114] can hardly generate syntax-correct test cases; advanced fuzzers [73, 72] adopt higher-level mutation in the abstract syntax tree (AST) or the intermediate representation (IR) to preserve the input structures. Alternatively, generation-based fuzzers leverage a precise model to describe the input structure [7, 17, 162], and thus can produce syntax-correct test cases from scratch. To further improve the semantic correctness, recent fuzzers adopt highly specialized analyses for specific languages [9, 6, 8].

However, a fuzzer will lose its generic applicability when it is highly customized for one specific language.Users cannot easily utilize the specialized fuzzer to test a different programming language, but have to develop another one from scratch. Considering the complexity of language-specific fuzzers (*e.g.*, `CSmith` [8] consists of 80k lines of code) and the large number of programming languages (over 700 currently [163]), it is impractical to implement a specific fuzzer for each language. This puts current fuzzers in a dilemma: pursuing high semantic validity sacrifices their generic applicability, while retaining generic applicability cannot guarantee the quality of test cases.

In this chapter, we propose POLYGLOT, a fuzzing framework that can generate *semantically valid* test cases to extensively test processors of *different* programming languages. To achieve generic applicability, we design a uniform IR to neutralize the difference in the syntax and semantics of programming languages. Given the BNF (Backus–Naur form) grammar [14] of a language, POLYGLOT can generate a frontend that translates source programs into this IR. At the same time, users can provide semantic annotations to describe the specific semantics about the scopes and types of definitions of the language. The definitions include the defined variables, functions, and composite types. In this chapter, we use variables and definitions interchangeably. These annotations will produce semantic properties in IR during translation. For example, the BNF grammar of functions in C is `<func := ret-type func-name arg-list func-body>`. We can give annotations such as "`func` defines a new function" and "`func-body` creates a new scope". In this way, the language differences of programming languages are unified in the IR.

To achieve high language validity, we develop two techniques, the *constrained mutation* and the *semantic validation*, for test case generation. The constrained mutation retains the grammar structure of the mutated test cases, which helps preserve their syntactic correctness. Further, it tries to maintain the semantic correctness of the unmutated part of the test case. For example, it avoids mutating the statement "`int x = 1;`" in a C program in case that the rest of the program uses `x`, which otherwise introduces the error of using undefined variables. Since the syntactic correctness of a test case is preserved, and the semantic correctness of the unmutated part is still valid, the only thing left is to fix the potential semantic errors in the mutated part. The mutated part could introduce semantic errors because it might bring in invalid variables. To fix the errors, we replace these invalid variables according to the rules of scopes and types. For example, our mutation may insert a statement "`unknownVar + 1;`" into the mutated program P which only defines two variables, `num` of type integer and `arr` of type array. We should replace `unknownVar` with `num` because addition by 1 requires the variable to have an integer type. Our semantic validation utilizes the semantic properties of IR to collect type and scope information of variables in the test case and integrates them in the symbol tables. These symbol tables describe the types, scopes and the names of every variable. The semantic validation then utilizes them to replace invalid variables with valid ones in the mutated code, which greatly improves the semantic correctness (up to 6.4× improvement in our evaluation §3.4.3).

We implement POLYGLOT with 7,016 lines of C++ and Python codes, which focus on IR generation, constrained mutation and semantic validation. POLYGLOT currently supports 9 programming languages and we can easily adopt it to others.

We evaluate POLYGLOT on 21 popular compilers and interpreters of 9 different programming languages and successfully find 173 new bugs. 113 of the bugs have been fixed with 18 CVEs assigned. Our experiments show that POLYGLOT is more effective in generating high-quality test cases (up to 100× improvement in language validity), exploring program states (up to 30× more new paths) and detecting vulnerabilities (8× more unique bugs)

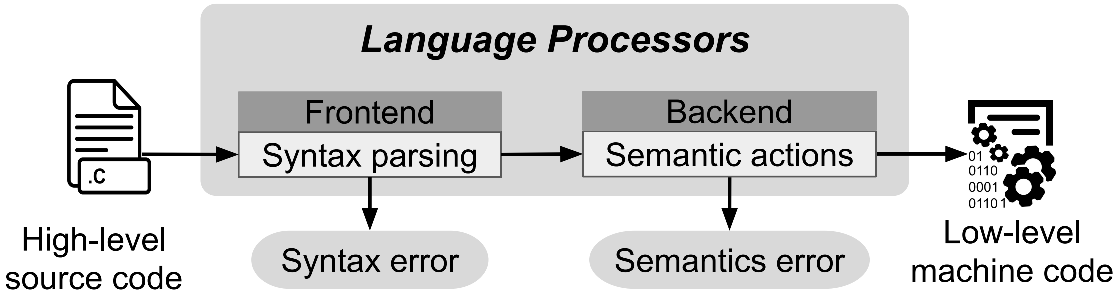


Fig.3.1: Workflow of language processors. Given a high-level source-code program, a language processor checks it for syntactic and semantic errors. If none, the processor translates the program into low-level machine code.

```c
1   struct S { int d; } s;
2   int a, c;
3   int main() {
4     short *e, b[3] = {1, 2, 0};
5     e = b;
6   + e = b     // missing ';'
7   + e = s;    // mismatch type
8   + e = res; // undef var
9     do{c += *(e++);} while(*e);
10    int res = c;
11    return res;
12  }
```

(a) An example C program

```javascript
function opt(x){return x[1];}
let arr = [1, 2];

if(arr[0]) {let arr2=[1, 2]};
// ig is a wrong keyword
+ ig(arr[0]) {let arr2=[1, 2]};
arr[1] += "1234";

for(let idx=0; idx<100; idx++)
  opt(arr);
+ for(let idx=0; idx<100; idx++)
+   opt(arr2); // undef var
```

(b) An example `JavaScript` program

Fig.3.2: Running examples. Fig.3.2(a) shows a program written in C, a statically typed programming language. If we replace line 5 with one of line 6–8, we get different errors as stated in the comments. Similarly, Fig.3.2(b) shows a program written in `JavaScript`, a dynamically typed language, which allows more type conversion.

than state-of-the-art general-purpose fuzzers, including the mutation-based fuzzer `AFL`, the hybrid fuzzer `QSYM` and the grammar fuzzer `Nautilus`. We also compare POLYGLOT with language-specific testing tools `CSmith` for C and `DIE` for `JavaScript`, and the results show that POLYGLOT can explore more program states.

3.2 Problem

In this section, we first briefly describe how language processors handle input programs, and how syntax errors and semantic errors terminate this process. Next, we illustrate the challenges and limitations of existing fuzzers in testing language processors. Then, we summarize the common semantic errors in test cases generated by fuzzing tools. Finally, we present our insights to solve this problem.

3.2.1 Language Processors

Language processors convert programs written in high-level languages into low-level machine codes. For example, compilers translate the whole program into machine codes, while interpreters translate one statement at a time.

Language processors check the input program for both syntactic and semantic errors. Any error can terminate the execution of processors. We show the workflow of language processors in Fig. 3.1. The frontend checks for syntax errors at the early stage of processing. Afterward, the backend checks for semantic errors, which cannot be detected by the parser. Only semantically correct programs can be successfully processed.

We show a C program in Fig. 3.2(a) and a `JavaScript` program in Fig. 3.2(b). If we add the statements that start with "+", we introduce errors in the program. For example, line 6 in Fig. 3.2(a) and line 6 in Fig. 3.2(b) introduce syntax errors, and the parser detects these errors and bails out. Line 7–8 in Fig. 3.2(a) and line 11–12 in Fig. 3.2(b) contain semantic errors which will be caught by the backend optimizer or translator.

3.2.2 Limitations of Current Fuzzers

Fuzzing is a well-received technique of discovering bugs and vulnerabilities in software [12, 164, 4]. However, current fuzzers have limitations in testing language processors. General-purpose mutation-based fuzzers [12, 114, 40, 54] are unaware of input structures and randomly flip the bits or bytes of the inputs, so they can hardly generate syntax-correct inputs. Recent works adopt higher level mutation in AST or IR to guarantee the syntactic correctness [73, 75]. Alternatively, generation-based fuzzers [17, 7] utilize a model or grammar to generate structural inputs effectively. These fuzzers have shown their advantages in passing the syntactic checks over random bitflip mutation-based fuzzers. Yet they ignore the semantic correctness of generated test cases and fail to find deep bugs in the optimization or execution code of language processors. We did a quick test to understand how semantic correctness helps fuzzers reach deeper logic: compiling the code in Fig. 3.2(a) with "-O3" covers 56,725 branches in gcc-10, while those invalid variants starting with "+" only trigger less than 27,000 branches as they are rejected during or right after the parsing.

Researchers try to specialize their fuzzers for higher semantic correctness [8, 9, 6, 72]. CSmith [8] performs heavy analyses to generate valid C programs without undefined behaviors. `JavaScript` fuzzers [9, 72] consider the types of expressions to avoid semantic errors in generated test cases. SQUIRREL [6] tackles the data dependency of SQL to generate valid queries to test DBMSs. Unfortunately, these approaches are highly specialized for one programming language. Users need to put huge development efforts to adopt them on new programming languages, which is time-consuming and impractical considering the large number of real-world languages [163].

Recent language-based fuzzers [73, 65] try to generate correct test cases for different languages. LangFuzz [73] replaces every variable in the mutated code randomly while `Nautilus` [65] uses a small set of predefined variable names and relies on feedback guidance to improve semantic correctness. However, these strategies are only effective in testing languages which allow more implicit type conversion, such as `JavaScript` and PHP.

3.2.3 Common Semantic Errors

We manually investigate 1,500 invalid test cases generated by existing language fuzzers [17, 9, 65] and summarize four common types of semantic errors. Two of them are related to the scope of variables and functions, and the rest two are related to the types of variables and expressions. These errors violate the common rules of types and scopes on definitions and are language-independent.

Undefined Variables or Functions. Variables or functions should be defined before they can be used. Otherwise, the behaviors of the program can be undefined or illegitimate. For example, line 8 of Fig. 3.2(a) uses an undefined variable `res` and C compilers refuse to compile the code.

Out-of-scope Variables or Functions. In a program, variables or functions have their scopes, which determine their visibility. We cannot use an out-of-scope invisible variable or function. For example, `arr` is visible at line 10 of Fig. 3.2(b), while `arr2` is not since its scope is within the `if` statement at line 4.

Undefined Types. Many programming languages allow users to define custom types, such as `class` in `JavaScript` and `struct` in C. Like variables, such types should be defined before their instances can be used.

Unmatched Types. Usually, assigning a value to a variable of incompatible type or comparing incompatible types introduces semantic errors. In some cases, programming languages allow type conversions, which convert mismatched types to compatible ones explicitly or implicitly. For example, `e` of type `pointer of short` and `s` of type `S` are not compatible in C, so line 7 of Fig. 3.2(a) introduces an error. Line 7 of Fig. 3.2(b) is correct because in `JavaScript` numbers can convert to strings.

3.2.4 Our Approach

The goal of this chapter is to build a generic fuzzing framework that generates semantically correct inputs to test different language processors. We achieve the goal in two steps. First, we neutralize the difference in syntax and semantics of programming languages by embedding them into a uniform IR, so we can perform uniform mutation or analysis regardless of the underlying languages. Second, we constrain our mutation to generate new test cases, which might contain semantic errors, and then we perform semantic validation to fix these errors.

Neutralizing Difference in Programming Languages. Different programming languages have unique syntax and semantics. To neutralize their differences, we design a new intermediate representation to map the language-specific features, both syntactic and semantic, into a uniform format. Given the BNF grammar of a language, we can generate a frontend to translate a source program into an IR program. The IR program consists of a list of IR statements or IRs as we call them in this chapter. This IR program keeps the syntactic structures of source programs so we can easily translate it back into the original source. Regarding semantics, we design a simple annotation format for users to describe the scopes and types of a language. These descriptions will be encoded into the semantic properties of the IR and guide our system to fix semantic errors. After the language-specific grammar and semantics are captured by the IR, we can perform mutation or analyses regardless of the underlying language.

Improving Language Validity. We improve the language validity with *constrained mutation* and *semantic validation*. Our constrained mutation tries to preserve two aspects of the program: the syntactic correctness of the whole program, and the semantic correctness of the unmutated part. First, we mutate the IRs based on their IR types that reflect the underlying grammar structures. This preserves the syntactic correctness of the test case. For example, we replace an `IF` statement (in the form of IR) with another `IF` statement instead of a function call expression. Second, we only mutate IRs with *local effects* to preserve the semantic validity of the unmutated code. Such IRs contain no definitions or create new local scopes. For example, in Fig. 3.2(b), line 7 has local effects because it only uses the variable `arr` and contains no new definition. Without line 7, the rest of the program is still valid. Therefore, assuming the initial test case is correct, a mutated variant produced by constrained mutation only has potential semantic errors in the mutated part, which might use invalid variables. To fix these errors in a systematic way, our semantic validation first utilizes IR's semantic properties to collect type and scope information of the mutated test case. We integrate the collected information into the symbol tables, which contain the types, scopes and names of every definition. These symbol tables guide POLYGLOT to replace the invalid use of variables properly. Afterward, the validated test cases should be correct and are helpful for thoroughly fuzzing language processors.

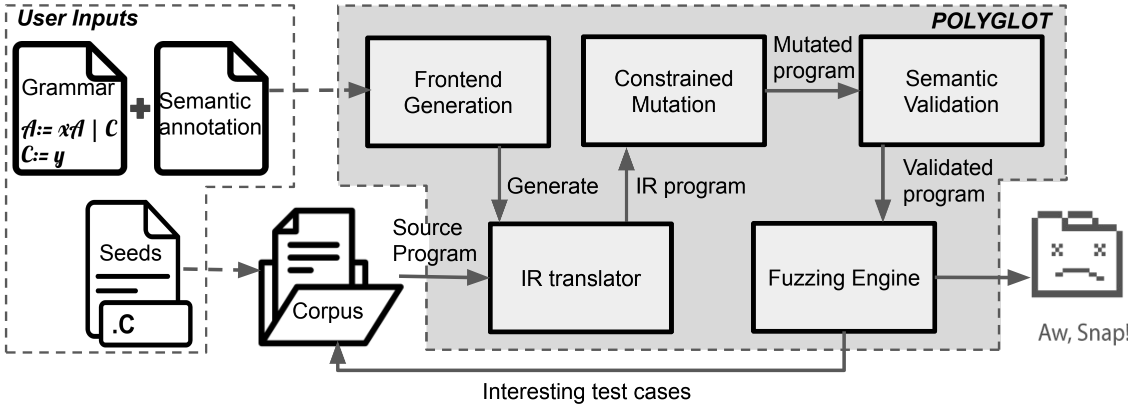


Fig.3.3: Overview of POLYGLOT. POLYGLOT aims to discover bugs that crash language processors. POLYGLOT accepts the BNF grammar, semantic annotations, and seeds from users as input. First, the frontend generator generates an IR translator that converts a source program to an IR program. Second, the constrained mutator mutates the IR program to get new ones, which might contain semantic errors. Next, the semantic validator fixes the semantic errors. Finally, the fuzzer runs validated programs to detect bugs.

3.3 Design & Implementation

Fig. 5.1 shows an overview of POLYGLOT. Given the BNF grammar, semantic annotations and initial test cases of the targeted programming language, POLYGLOT aims to find inputs that trigger crashes in the language processor. First, the frontend generator generates an IR translator using the BNF grammar and the semantic annotations (§3.3.1). Then, for each round of fuzzing, we pick one input from the corpus. The IR translator lifts this input into an IR program. Next, the constrained mutator mutates the IR program to produce new syntax-correct ones, which might contain semantic errors (§3.3.2). Afterward, the semantic validator tries to fix the semantic errors in the new IR programs (§3.3.3). Finally, the IR program is converted back to the form of source code and fed into the fuzzing engine. If the test case triggers a crash, we successfully find a bug. Otherwise, we save the test case to the corpus for further mutation if it triggers a new execution path.

3.3.1 Frontend Generation

To achieve generic applicability, our frontend generation generates a translator that transforms a source program into an IR program. This lowers the level of mutation and analysis

```
//IR<type,       left, right, op,         val
//        [, semantic_property]>
ir0<Type,        NIL,  NIL,   NIL,        "int">
ir1<RetType,     ir0,  NIL,   NIL,         NIL >
ir2<FuncName,    NIL,  NIL,   NIL,      "main",
        [FunctionName]>
ir3<Literal,     NIL,  NIL,   NIL,          12 >
ir4<Literal,     NIL,  NIL,   NIL,          23 >
ir5<BinaryExpr,  ir3,  ir4,   "+",         NIL >
ir6<RetStmt,     ir5,  NIL,   "return ;",  NIL >
ir7<FuncBody,    ir6,  NIL,   "{ }",       NIL,
        [FunctionBody, NewScope]>
ir8<NIL,         ir1,  ir2,   "()",        NIL >
ir9<FuncDef,     ir8,  ir7,   NIL,         NIL,
        [FunctionDefinition]>
ir10<Program,    ir9,  NIL,   NIL,         NIL >
```

(a) IR program for "`int main() { return 12+23;}`". The format of IR is shown in the comments at the top.

```
<program> ::=
  (<global-def> | <func-def>)*
...

<func-def> ::=
  (<ret-type> <func-name>
  "(" <func-arg>? ")" <func-body>)

<ret-type> ::= <type>

<type> ::=
  ("int" | "short" | ...)

<binary-expr> ::=
  <literal> "+" <literal>
...
```

(b) Part of the BNF grammar for `C` programs.

```
{ "Comment1": "Scopes and composite types",
  "func-def":["FunctionDefinition"],
  "func-name":["FunctionName"],
  "func-body":["FunctionBody", "NewScope"],

  "Comment2": "Types and conversion rules",
  "BasicType": ["int", "short", "..."],
  "ConversionRule": [
    {"short": ["int", "..."]}
  ],

  "Comment3": "Type inference rules",
  "TypeInference":{
    "+": { "left": "long", "right": "long",
      "output":"long"} }
}
```

(c) Part of semantic annotations for the grammar in Fig.3.4(b). It is in `JSON` format.

Fig.3.4: An example IR program with its corresponding BNF grammar and semantic annotations. The IRs in Fig.3.4(a) are in a uniform format. The number suffix of the IR is the IR order. The IR types have corresponding symbols in the BNF grammar in Fig.3.4(b). `left` and `right` are the two operands. As `func-def` has four components and the IR can have no more than two operands, `ir8` is an intermediate IR for `func-def` and does not have a type. The semantic annotations in Fig.3.4(c) describe what semantic properties the symbols in the BNF should have, which will be reflected in the `semantic_property` in IRs. We predefine properties about types and scopes for users to use.

from language-specific source code to a uniform IR.

In Fig. 3.4, we show the IR (Fig. 3.4(a)) of a simple C program to demonstrate how the BNF grammar (Fig. 3.4(b)) and semantic annotations (Fig. 3.4(c)) help construct the IR statements. Each symbol in the BNF grammar generates IRs of a unique type (*e.g.*, symbol `<func-def>` generates `ir9` of type `FuncDef`). The original source code is stored in the `op` or `val` of the IRs (*e.g.*, the `val` of `ir2` stores the name `main`). We predefine semantic properties about types and scopes of variables for users to use. The generated IRs will carry these properties as described in the annotations (*e.g.*, `ir9` has property `FunctionDefinition`). Users can easily use the BNF grammar and semantic annotations to describe the specific syntax and semantics of a programming language.

Intermediate Representation

Our IR is in a uniform format and captures the syntax and semantics of the source program. It includes an order, a type, an operator, no more than two operands, a value, and a list of semantic properties. The IR order and the type correspond to the statement order in source code and the symbol in the BNF grammar respectively. The IR operator and the IR value store the original source code. All the IRs are connected by the IR operands, which are also IRs. These parts carry the syntactic structures, while the semantic properties describe the semantics of the source program, as discussed below.

Syntactic Structures. Syntactic structures keep all the grammar information of the source program. As we see earlier, some IRs store a small piece of the source code (*e.g.*, a function name is stored in an IR of type `FuncName`). Also, the IRs are connected in a directed way that forms a tree view of the source program. If we perform inorder traversal on the IR program, we can reconstruct the original source program.

Semantic Properties. Semantic properties capture the semantics about the scopes and types of definitions. They tell us which IRs belong to variable definitions. Additionally, for scopes, they tell which IRs create new scopes so that we can decide the visibility of the variables within the scopes. For types, they describe the predefined and user-defined types in a language and their type conversion rules. They also describe the expected operand types and the output type of operators, which perform mathematical, relational, or logical operations and produce a result.

For example, `ir9` in Fig. 3.4(a) has the semantic property `FunctionDefinition`, indicating that it relates to a function definition. Line 14-15 in Fig. 3.4(c) describe the inference rule for the operator "+", which accepts two operands of type `long` and outputs a result of type `long`. Assuming number literals are of type `long`, we know `11 + 12` produces a result of `long`.

Generating IR Translator

To generate a translator, users should provide the BNF grammar, which describes the unique syntax, and semantic annotations, which capture the specific semantics of a language. The language information of these two files is embedded into the syntactic structures and semantic properties of IR respectively. First, the frontend generator treats every different symbol in the grammar as a different object. Then it analyzes the semantic annotations to decide which symbols should have what semantic properties. Finally, it generates for each object unique parsing and translation methods, which parse the source code and generate IRs with required semantic properties. These generated methods composite an IR translator.

3.3.2 Constrained Mutation

As the first step towards language validity, we apply two rules to constrain our mutation on an initially correct test case to preserve its syntactic correctness (§3.3.2) and the semantic correctness of its unmutated part (§3.3.2). The former is the base for semantic correctness, and the latter makes it possible to gain language validity by fixing the semantic errors in the mutated part.

Rule 1: Type-Based Mutation

This rule performs three different mutation strategies based on the IR types. *Insertion* inserts a new IR (*e.g.*, IRs from another program) to the IR program. This includes inserting an IR that represents an element to a list and inserting an IR to where it is optional but currently absent. *Deletion* performs the opposite operation of insertion. For example, in C, a statement block is a list of statements, and we can insert a statement to the block. Also, we can delete an optional `ELSE` statement after an `IF` statement. *Replacement* replaces an IR with a new one of the same type. For example, we can replace an addition expression with a division expression as they are both of type `EXPRESSION`.

Since the IR type reflects the grammar structures of underlying source codes, this rule helps preserve the syntactic correctness of the mutated test cases.

Rule 2: Local Mutation

This rule requires us to only mutate IRs with local effects. Changes in these IRs will not invalidate the semantic correctness of the rest of the program. POLYGLOT handles two types of IRs with local effects as follows.

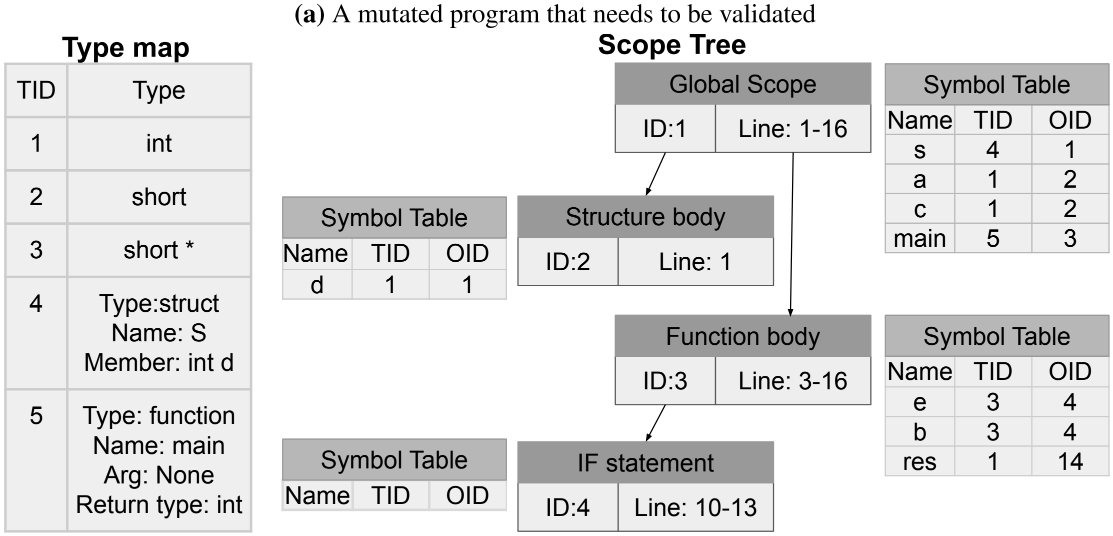
IRs that Contain No New Definitions. These IRs do not define any variables, functions, or types, so the rest of the program will not use anything defined by them. Even if these IRs get deleted, the rest of the program will not be affected. For example, line 7 in Fig. 3.2(b) only uses the variable `arr` and does not define anything. If we delete this line, the program can still be executed.

IRs that Create Scopes. These IRs can contain new definitions, but these definitions are only visible within the scope created by the IRs. For example, the `for` statement at line 9 of Fig. 3.2(b) creates a new scope. `idx` is defined and only valid within this new local scope. Therefore, mutating the `for` statement as a whole will not affect the rest of the program.

With these two rules, our constrained mutation produces syntax-correct test cases. These test cases might contain semantic errors. According to local mutation, the semantic errors are introduced by the mutated part, which might use invalid variables from other test cases. Next, we will fix all these errors to get a semantically correct test case.

```c
1   struct S { int d; } s;
2   int a, c;
3   int main() {
4   short *e, b[3] = {1, 2, 0};
5   e = b;
6   // Originally: do{ c += *(e++); } while(*e);
7   /* Replaced by: if(x >= y){
8                       struct X z;
9                       x += y; } */
10  if(FIXME >= FIXME){
11  //struct X z;
12  FIXME += FIXME;
13  }
14  int res = c;
15  return res;
16  }
```

(a) A mutated program that needs to be validated



(b) The type map, scope tree and its the symbol tables of the program in Fig. 3.5(a)

Fig. 3.5: A mutated variant of Fig. 3.2(a) and its semantic information collected by the semantic validator. The mutated program is generated by replacing the `DO-WHILE` statement (line 6) with `IF` statement (line 10-13) in Fig. 3.5(a). Every invalid variable in the mutated part is replaced with a `FIXME`. Line 11 is removed because it uses a composite type (`struct X`) without a detailed definition. As shown in Fig. 3.5(b), the type map contains the types used in the program. The program has four different scopes created by different symbols. Each scope has a symbol table with the information of definitions within the scope. `TID` in the symbol table refers to the `TID` in the type map. `OID` corresponds to the statement order in the IR program and we currently use line number for easy demonstration.

3.3.3 Semantic Validation

As the second step towards language validity, we perform semantic validation to fix the semantic errors in the mutated part of the test case. We do so by replacing the invalid variables with the valid ones. To figure out the proper variables for replacement, we first need to know the scopes of the variables, which tell us the available variables to use. This avoids using undefined or out-of-scope variables. Further, we need to know the types of these variables so that we can use them appropriately, which avoids using variables of undefined or unmatched types.

Therefore, our semantic validation relies on two components: *type system* that collects type information of variables (§3.3.3) and *scope system* that collects scope information (§3.3.3). We then integrate all the information into the symbol tables, which contain the types, scopes, and names of every definition in the test case. With the symbol tables, the semantic validation generates correct expressions to replace the invalid ones and produces a semantically correct test case (§3.3.3).

Type System

In programming languages, types include predefined ones such as `int` in C, and user-defined ones such as `class` in `JavaScript` [165]. We call the former *basic types* and the latter *composite types*. Basic types are limited so they can be completely described with semantic annotations, but composite types cannot as they are specific to test cases. To collect precise type information, we need to handle both basic types and composite types. Therefore, POLYGLOT utilizes the type system to construct composite types on demand and infer types of variables or expressions.

Type Map. As the collected type information will be used frequently, we maintain them in a type map for easy and fast access. The key of the map is a unique id for the type, and the value is the structure of the type. This map stores all the basic types of a language and the composite types used in the current test case. For example, Fig. 3.5(a) and Fig. 3.5(b) show a mutated program and its type map. We can see that type id 5 refers to a function type whose name is `main` and return type is `int`. As composite types are specific to a test case, we remove them from the map each time we finish processing a test case to avoid using types defined in other programs.

Composite Type Construction. Currently, the type system supports the construction of three composite types: structures, functions, and pointers. These types consist of several components. For example, a function consists of a function name, function arguments, and return value.

To construct a composite type, the type system walks through the IR program to find IRs related to composite type definitions by checking their semantic properties. When it finds one, it searches for the required components for this definition. Then type system creates a new type with the collected components and stores it in the type map.

Type Inference. The type system infers the types of variables so that we know how to use them correctly. We handle both the variable definition and variable use. For a variable definition, we check whether it has an explicit type. If so, the type system searches the name of the type in the type map. Then it returns the corresponding type id when the names match. Otherwise, we infer the type of a variable from its assigned expression, which will be discussed in the next paragraph. For example, in `C`, "`int y;`" explicitly states that the variable `y` is of type `int`. In `JavaScript`, "`let z = 1.0;`" does not state a type for `z`, but we can infer from expression `1.0` that `z` is of floating-point number type. For variable use, we just look for the variable name in the symbol tables (§3.3.3), which contains the type information of variables, and return its type.

To infer the type of an expression, we first check whether it consists of a simple variable or literal. If it is a variable, it must be a variable use, which has been handled above. If it is a literal, we return its type as described in the semantic annotations. For an expression with operands and operator, we first recursively infer the types of the operands as they are also expressions. As discussed in §3.3.1, the semantic properties describe the expected operand types and the output type of the operator. If the inferred operand types can match or convert to the expected types, we return the corresponding result type.

Our type inference has limitations in dynamically typed programming languages, where the types might be undecidable statically. For example, the type of `x` in line 1 of Fig.3.2(b) is not determined because `JavaScript` can call the function with arguments of any type. If we simply skip the variables whose types cannot be inferred, we might miss useful variables. Therefore, we define a special type called `AnyType` for these variables. Variables of `AnyType` can be used as variables of any specific type. Using `AnyType` might introduce some type mismatching, but it can improve the effectiveness of POLYGLOT in dynamically typed programming languages.

Scope System

A program can have different scopes that decide the visibility of variables within them. The scope system partitions the program into different scopes so that variables automatically gain their visibility according to the scope they are inside. Afterward, we integrate the type information collected by the type system and the scope information into symbol tables. The symbol tables contain all the necessary information of variables for fixing the semantic errors.

Partitioning IR Program With A Scope Tree. A program has a global scope where variables are visible across the program. Other scopes should be inside existing ones. This forms directed relations between scopes: variables in the outer scope are visible to the inner scope, but not vice versa. Therefore, we build a directed scope tree to describe such relations. In the scope tree, the global scope is the root node, and other scopes are the child nodes of the scopes that they are inside. As the semantic properties of IR tell which IRs create new scopes, we create a new node of scope when we find such an IR. We assign each node a unique id and label the IR, along with their children IRs (*i.e.*, their operands), with this id to indicate that they belong to this scope. In this way, we partition the IR program into different scopes in the tree. A variable is visible to a node if the variable is in any node along the path from the root node to the given node.

Fig.3.5(b) shows the constructed scope tree of Fig.3.5(a). "Line" means the IRs translated from these lines belong to the scope or the children of the scope. Scope 1 is the global scope, which is the root node. Scope 2 and 3 are created by the structure body of `S` and function body of `main` respectively and they are child nodes of scope 1. Variables in scope 1 and 3 are visible to scope 4 as they are in the same path.

Symbol Table. We integrate the collected information of types and scopes by building symbol tables which contain the names, scopes, defined orders, and types of variables. They describe what variables (names) are available at any program location (scopes and defined orders) and how they can be used (types).

Fig. 3.5(b) shows symbol tables of each scope for Fig. 3.5(a). Variables `s, a, c` and function `main` are defined in scope 1, the global scope, and `d` is defined in scope 2 which is the scope for structure body. There is no definition in the `IF` statement so its symbol table is empty. `TID` is the type id of the variable, which corresponds to the `TID` in the type map. `OID` is the defined order of variables and we currently use the line number as `OID` for easy demonstration. A variable is visible at a given location (*i.e.*, line number) if its scope is the ancestor node of the scope of the location and if it is defined before the location.

Validation

With the symbol tables, we can fix the semantic errors in the mutated test case. We call this process *validation*. We replace every invalid variable with the special string `FIXME` to indicate that this is an error to be fixed, as shown in Fig. 3.5(a).

Specifically, we first remove any IRs that use user-defined types in the mutated part in case we cannot find the definition of these types. Then, for each `FIXME` in the mutated code, we replace it with a correct expression. We generate the expressions with the variables in the symbol tables according to their types and scopes. For example, in Fig. 3.5(a), the original `for` statement is replaced by an `if` statement during mutation (line 6-10). The `if` statement contains a user-defined structure without definition (line 8), so we remove line 8. Finally, we replace the `FIXME`s with generated expressions.

Generating Valid Expressions. POLYGLOT generates four types of expressions: a simple variable, a function call, an element indexed from an array, and a member accessed from a structure. In Fig. 3.5(a), "`a`", "`main()`", "`b[1]`", "`s.d`" are all examples of generated expressions.

First, POLYGLOT infers the type of expressions containing `FIXME` and tries to figure out what type of expressions should be used for replacement. It adopts a bottom-up approach: it assigns `AnyType` to each `FIXME`, and converts `AnyType` to a more specific type when it goes up and encounters concrete operators. For example, we want to fix the two `FIXME`s in the expression "`FIXME >= FIXME`" in line 10 of Fig. 3.5(a). We assign `AnyType` to both of them. Then we go up the expression and encounter the operator "`>=`", which accepts numeric types as operands, such as `int` and `short`, and outputs a result of type `bool`. As the operands are `FIXME` of `AnyType`, which can be used as any other specific type, we convert the type of `FIXME` to numeric types. Now POLYGLOT needs to generate two expressions of numeric types to replace the two `FIXME`s.

Second, POLYGLOT checks the symbol tables to collect all the available variables. It walks through the symbol tables of all the visible scopes in scope tree, from the global scope to the scope of the expression with `FIXME`, and collect the variables defined before the to-be-validated expression.

Third, we enumerate the possible expressions we can generate from these variables and categorize them by types. For example, from the definition `s` in line 1 in Fig. 3.5(a), we can generate the expressions `s` and `s.d`. They are of different types so they belong to different categories.

Finally, POLYGLOT randomly picks some expressions of the required type to replace `FIXME`s. If every `FIXME` of a test case can be replaced by a proper expression, the validation succeeds. The validated test case should be semantically correct and we feed it to the fuzzer for execution. If the validation fails (*e.g.*, there is no definition for a specific type), we treat the test case as semantically invalid and discard it.

Table 3.1: Line of codes of different components of POLYGLOT, which sum up to 7,016 lines. As we build our fuzzer on AFL, we only calculate the code that we add into AFL, which is 285 lines in the fuzzer component.

Module	Language	LOC
Frontend Generator	C++	367
	Python	1,473
Constrained Mutator	C++	1,273
Semantic Validator	C++	3,313
Fuzzer	C++	285
Others	C++/Bash	305
Total	C++/Python/Bash	**7,016**

One possible solution to fix `FIXME >= FIXME` at line 11 in Fig.3.5(a) is "`b[1] >= s.d `", where we replace the `FIXME`s with "`b[1]`" of type `short` and "`s.d`" of type `int`. `short` and `int` are of different numeric types, but `short` can be converted to `int`. Therefore, `b[1]` and `s.d` can be compared by >= though they are of different types.

3.3.4 Implementation

We implement POLYGLOT with 7,016 lines of code. Table 3.1 shows the breakdown.

Frontend Generation. We extend the IR format proposed in [6] by adding semantic properties. Users provide semantic annotations to help generate these properties. The frontend generator generates a parsing and a translation method from code templates written in C++ for each symbol in the BNF grammar. Then we use Bison [166] and Flex [167] to generate a parser with the parsing methods. The parser and the translation methods are compiled together to be an IR translator.

Scope Tree Construction. The scope system maintains a stack of scopes to construct the scope tree. The scope in the stack top indicates the current scope. First, it generates the global scope as the root and pushes it in the stack. Next, it walks through IRs in the IR program, labeling each IR with the id of the scope in the stack top. Meanwhile, it checks the semantic properties of the IR. If the scope system meets an IR that creates a new scope, it creates one. It sets the new scope as the child node of the scope at the stack top and pushes it to the stack. After the children of the IR are recursively processed, the scope system pops the scope out of the stack. In this way, we construct the scope tree and partition the IRs.

Builtin Variables and Functions. To improve the diversity of the generated expressions, POLYGLOT allows users to optionally add predefined builtin variables and functions of the tested programming language. These builtin variables and functions are written in the source format and added along with the initial seed corpus. POLYGLOT then analyzes these test cases and collects them as definitions. These definitions will be added into the symbol table of the global scope of every generated test case and thus used for expression generation.

Complex Expression Generation. To introduce more code structures in the test cases, we allow semantic validation to generate complex expressions. Since we have the symbol tables and the inference rule of operators, we can chain simple expressions with operators. For example, with the symbol tables in Fig. 3.5(b), we can generate complex expressions such as `(a + b[1]) » c`, which is chaining three simple expressions (`a`, `b[1]`, `c`) with three operators (`+`, `()`, `»`). We first randomly pick an operator and then recursively generate expressions of the types of its operands. Afterward, we simply concatenate them to get a complex expression.

Fuzzer. We build POLYGLOT on top of AFL 2.56b. We keep the fork-server mechanism and the queue schedule algorithm of AFL and replace its test case generation module with POLYGLOT's. POLYGLOT also makes use of AFL's QEMU mode, which can test binary without instrumentation. Since many programming languages are bootstrapping, which means their language processors are written in themselves, it is difficult or time-consuming to instrument these processors. Using AFL QEMU mode can greatly save time and effort.

User Inputs for Adoption. To apply POLYGLOT to a programming language, users need to provide: the BNF grammar, the semantic annotations and the initial corpus of test cases. The BNF grammars of most programming languages are available from either the official documents of the languages or open-source repositories [168]. The semantic annotations should describe symbols that relate to definitions, symbols that create new scopes, basic types of the languages, and the inference rules of operators. We provide a template of

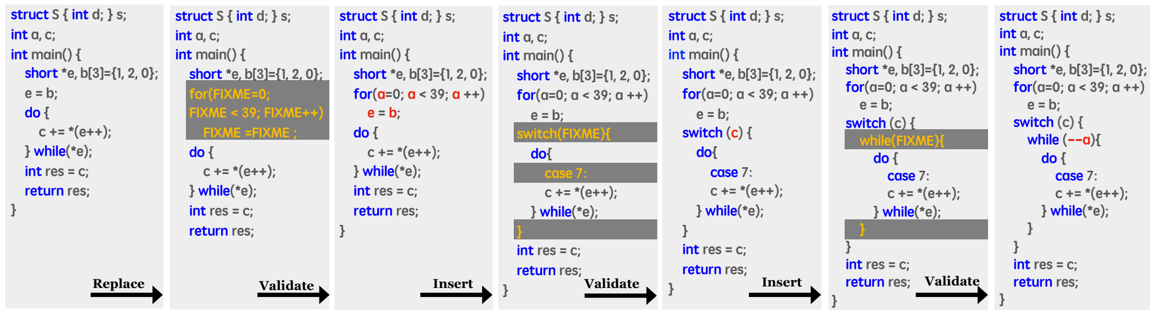


Fig. 3.6: The process of how POLYGLOT generates the PoC for Case Study 1. We perform three rounds of mutation and validation on the motivating example of Fig. 3.2(a). Each round of mutation introduces new code structures (*e.g.*, the first replacement introduces a `for` statement), and each round of validation generates correct expressions to fix the semantic errors.

semantic annotations in JSON format so users can easily adjust them according to their needs. Users are free to choose the corpus that fits the tested processor. In our case, it took one of our authors 2-3 hours to collect the amentioned inputs for one language and 3-5 hours to refine them to fit in POLYGLOT.

3.4 Evaluation

Our evaluation aims to answer the following questions:

- Can POLYGLOT generally apply to different real-world programming languages and identify new bugs in their language processors? (§3.4.2)
- Can semantic validation improve POLYGLOT's fuzzing effectiveness? (§3.4.3)
- Can POLYGLOT outperform state-of-the-art fuzzers? (§3.4.4)

3.4.1 Evaluation Setup

Benchmark. To evaluate the genericity of POLYGLOT, we test 21 popular processors of nine programming languages according to their popularity [169] and variety in domains (*e.g.*, `Solidity` for smart contracts, R for statistical computation, SQL for data management). We show the target list in Table 3.3. To understand the contributions of our semantic validation,

Table 3.2: Detected bugs. POLYGLOT found 173 bugs, including 6 from `GCC`, 24 from `Clang`, 4 from `G++`, 6 from `Clang++`, 3 from `v8`, 1 from `JSCore`, 9 from `ChakraCore`, 1 from `Hermes`, 1 from `mujs`, 4 from `njs`, 5 from `JerryScript`, 1 from `DukTape`, 1 from `QuickJs`, 4 from `R`, 3 from `pqR`, 35 from `php`, 27 from `SQLite`, 12 from `Lua`, 2 from `LuaJit`, 16 from `solc`, 8 from `freepascal`.

Bug Type: **UAF**: use-after-free. **BOF**:buffer overflow of Global (**G**), Heap (**H**), and Stack (**S**).

SEGV: potential exploitable vulnerability. **AF**: assertion failure. **LEAK**: memory leak.

Bug Status: Fixed: Fixed, Verified: Verified, ◇: Open, †: CVE

ID	Type	Function	Reference
SQLite v3.31, 300K LoC			
1	UAF	`resetAccumulator`	† 2020-13871
2	SBOF	`sqlite3_str_vappendf`	† 2020-13434
3	SEGV	`sqlite3ExprCodeTarget`	† 2020-13435
4	HBOF	`multiSelectOrderBy`	† 2020-15358
SQLite v3.32 (development), 304K LoC			
5	UAF	`sqlite3BlobCompare`	Fixed da5a09b
6	UAF	`resetAccumulator`	Fixed 7c6d876
7	SEGV	`sqlite3VdbeExec`	Fixed 0899cfu
8	SEGV	`sqlite3VdbeExec`	Fixed 1f6f353
9	SEGV	`sqlite3VdbeCursorMoveto`	Fixed e5504e9
10	SEGV	`moveToRoot`	Fixed f7d8908
11	SEGV	`sqlite3Select`	Fixed b706351
12	SEGV	`sqlite3VdbeCursorMoveto`	Fixed 5829597
13	SEGV	`isAuxiliaryVtabOperator`	Fixed 4374860
14	AF	`flattenSubquery`	Fixed e367f31
15	AF	`sqlite3VdbeExec`	Fixed 9fb26d3
16	AF	`sqlite3ExprCodeIN`	Fixed fd1bda0
17	AF	`selectExprDefer`	Fixed 7aa91ab
18	AF	`Where` clause	Fixed e12a0ae
19	AF	`sqlite3VdbeExec`	Fixed c4c5648
20	AF	`Vdbe` Memory	Fixed d165ad7
21	AF	`sqlite3WhereBegin`	Fixed 82b588d
22	AF	`clearSelect`	Fixed 618156e
23	AF	`OrderBy` Optimize	Fixed 41c1456
24	AF	`xferOptimization`	Fixed f07d71b
25	AF	`Btree` Cursor	Fixed aa43786
26	AF	`sqlite3IsLikeFunction`	Fixed b985f0b
27	AF	`sqlite3VdbeExec`	Fixed 9d36667
GCC v10.0.1, 5956K LoC			
28	SEGV	`gimplify_compound_expr`	Fixed 93576
29	AF	`Nonlocal Reference`	Verified 93572
30	AF	`force_constant_size`	Verified 93573
31	AF	`c_expr_sizeof_expr`	◇ 93574
32	AF	`extended_tree`	Verified 93577
33	AF	`gimple_call_arg`	Verified 93631
G++ v10.0.1, 5956K LoC			
34	SEGV	`dump_parameters`	Verified 93788
35	AF	`pop_local_binding`	Fixed 93752
36	AF	`Output constructor`	Verified 93753
37	AF	`cc1plus`	Fixed 93789
Clang v11.0.0, 1578K LoC			
38	SEGV	`getExprLoc`	◇ 44729
39	SEGV	`isa_impl_wrap`	◇ 44735
40	SEGV	`ActOnFinishFullExpr`	◇ 44741
41	SEGV	`usePhysReg`	Verified 44750
42	SEGV	`VisitBuiltinCallExpr`	◇ 44756
43	AF	`CheckListElementTypes`	◇ 44738
44	AF	`ParseSwitchStatement`	◇ 44740
45	AF	`ActOnForStmt`	◇ 44732
46	AF	`VisitCastExpr`	◇ 44742
47	AF	`DoMarkVarDeclReferenced`	◇ 44744
48	AF	`eliminateRegSequence`	Fixed 44749
49	AF	`getASTRecordLayout`	◇ 44734
50	AF	`cast_retty`	◇ 44755
51	AF	`ImplicitMember`	◇ 44737
52	AF	`VisitBuiltinCallExpr`	◇ 44757
53	AF	`isIncompleteDeclExternC`	◇ 44758
54	AF	`udivrem`	◇ 44831
55	AF	`removeDecl`	◇ 44832
56	AF	`getIndirectResult`	◇ 44850
57	AF	`getCommonPtr`	◇ 44867
58	AF	`TailRecursionEliminator`	Fixed 46125
59	AF	`getTypeSizeInBits`	◇ 46211
60	AF	`getAccessTagInfo`	◇ 46388
61	SEGV	`Inifite loop`	◇ 46262
Clang++ v11.0.0, 1578K LoC			
62	AF	`ParenExpression`	◇ 44924
63	AF	`HasAccess`	◇ 44925
64	AF	`LookupTemplateName`	◇ 44926
65	AF	`ImplicitConversion`	◇ 44927
66	AF	`CreateOverloadedBinOp`	◇ 44928
67	AF	`PushDeclContext`	◇ 44940
V8 v8.2.0, 811K LoC			
68	SEGV	`TimeFormatConstructor`	Fixed 1052019
69	AF	`DeclarationScope`	Fixed 1060023
70	SEGV	`monkey-patched`	◇ 1098588
JavaScriptCore v2.27.4, 497K LoC			
71	SEGV	`finishCreation`	Fixed 208040
ChakraCore v1.12.0, 690K LoC			
72	SEGV	`ReparseAsmJsModule`	Verified 6472
73	SEGV	`FlowGraph::Build`	Verified 6473
74	AF	`MapStFldHelper`	Verified 6474
75	SEGV	`CloneBlockData`	◇ 6482
76	AF	`FlowGraph` Destruct	◇ 6483
77	AF	`GetArgCount`	◇ 6485
78	AF	`CallFunction`	◇ 6486
79	SEGV	`Optimize`	Verified 6487
80	SEGV	`EmitTopLevelStatement`	◇ 6488
Hermes v0.5.0, 620K LoC			
81	AF	`operationWalk`	Fixed 279
MuJs 9f3e141, 15K LoC			
82	UAF	`jsR_run`	Fixed 136
njs v0.4.3, 78K LoC			
83	SEGV	`JSON Stringify`	† 2020-24348
84	SEGV	`njs_lvlhsh_bucket_find`	† 2020-24347
85	SEGV	`njs_value_property`	† 2020-24349
86	UAF	`JSON Parse`	† 2020-24346
JerryScript v2.4.0, 173K LoC			
87	UAF	`jerry-core`	† 2020-24344
88	AF	`Typedarray`	Fixed 3975
89	AF	`JSON parse`	Fixed 3950
90	AF	`JSON parse`	Fixed 3945
91	AF	`Statement parser`	Fixed 3944
DukTape v2.5.0, 238K LoC			
92	AF	`duk_hobject_getprop`	Fixed 2338
QuickJs 32d72d4, 89K LoC			
93	LEAK	`JS_FreeRuntime`	Fixed c389f9594
solc v0.6.3, 193K LoC			
94	SBOF	`ReferencesResolver`	Fixed 8266
95	AF	`calldataEncodedSize`	Fixed 8275
96	AF	`encodeToMemory`	Fixed 8276
97	AF	`Expression` Assignment	Fixed 8277
98	AF	`storeValue`	Fixed 8278
99	AF	`isDynamicallyEncoded`	Fixed 8279
100	AF	`interfaceType`	Fixed 8280
101	AF	`interfaceType`	Fixed 8282
102	AF	`TypeChecker::visit`	Fixed 8283
103	AF	`memoryDataSize`	Fixed 8286
104	AF	`ConstStateVar`	Fixed 8296
105	AF	`copyToStackTop`	Fixed 9272
106	SEGV	`SourceLocation`	Fixed 9404
107	SEGV	`RationalNumber`	Fixed 9434
108	SEGV	`SourceLocation`	Fixed 9441
109	SBOF	`interfaceType`	Fixed 9443
Lua v5.4.0, 31K LoC			
110	UAF	`luaD_call`	† 2020-15888
111	HBOF	`getobjname`	† 2020-15889
112	SEGV	`changedline`	† 2020-15945
113	SBOF	`luaO_pushvfstring`	† 2020-24342
114	UAF	`luaD_call`	† 2020-24371
115	SEGV	`lua_getlocal`	† 2020-24370
116	SEGV	`lua_traceexec`	† 2020-24369
117	SBOF	`luaO_pushvfstring`	Fixed eb41999
118	HBOF	`luaT_adjustvarargs`	Fixed 6298903
119	HBOF	`luaD_pretailcall`	Fixed eb41999
120	HBOF	`luaH_get`	Fixed 6298903
121	UAF	`lua_checkstack`	Fixed 34affe7
LuaJit v2.1, 88K LoC			
122	SEGV	`lj_err_run`	† 2020-15890
123	SEGV	`lj_err_run`	† 2020-24372
R v4.0.2, 851K LoC			
124	HBOF	printing system	Fixed 17867
125	HBOF	`printAttributes`	Fixed 17870
126	SEGV	`writelines`	Fixed 17876
127	SEGV	`model.matrix`	Fixed 17879
pqR 5c6058e, 845K LoC			
128	SEGV	`Primitive`	Verified issue-43
129	SEGV	`do_dotcall_e`	◇ issue-44
130	SEGV	`plot`	◇ issue-45
PHP v8.0.0, 1269K LoC			
131	SBOF	`header callback`	◇ 79207
132	UAF	`zend_hash_packed_to_hash`	Verified 79209
133	SEGV	`php_var_export_ex`	Verified 79214
134	SEGV	`zend_hash_next_index_insert`	Fixed 79258
135	SBOF	`php_array_element_dump`	Fixed 79259
136	AF	`zend_dispatch_try`	Fixed 79777
137	AF	`zend_closure_get_debug_info`	Fixed 79778
138	AF	`zend_wrong_string_offset`	Fixed 79779
139	HBOF	`zend_array_destroy`	◇ 79781
140	SEGV	`php_str_replace_common`	Fixed 79783
141	UAF	`slow_index_convert`	Fixed 79784
142	AF	`zend_array_destroy`	Verified 79788
143	AF	`zend_ast_evaluate`	Fixed 79790
144	AF	`_get_zval_ptr_cv_BP_VAR_RW`	Fixed 79791
145	AF	`zend_array_destroy`	Fixed 79792
146	UAF	fetch `imension_address`	Fixed 79793
147	AF	`zend_gc_delref`	Verified 79813
148	SEGV	`zend_string_release_ex`	Fixed 79815
149	AF	`_zend_is_inconsistent`	Verified 79816
150	SBOF	`zval_get_tmp_string`	Fixed 79817
151	UAF	`ZEND_VM_HANDLER` marco	Fixed 79818
152	SEGV	`zend_std_write_property`	◇ 79819
153	SEGV	`zend_get_properties_for`	Fixed 79821
154	SEGV	`php_pcre_replace_array`	Fixed 79829
155	LEAK	`php_var_dump`	Fixed 79830
156	AF	`increment_function`	Verified 79831
157	SEGV	`property_info_for_slot`	Fixed 79832
158	SEGV	`php_str_replace_in_subject`	◇ 79835
159	SEGV	`concat_function`	◇ 79836
160	SEGV	`zend_get_type_by_const`	◇ 79837
161	SEGV	`zend_mm_alloc_small`	◇ 79838
162	AF	`array_walk`	Fixed 79839
163	AF	`zend_get_property_offset`	Fixed 79862
164	LEAK	`zend_hash`	Fixed 79947
165	LEAK	`zend_string`	Fixed 79951
freepascal v3.3.1, 405K LoC			
166	AF	`tmoddivnode.simplify`	Fixed 37449
167	AF	`TSparcReader.BuildOperand`	◇ 37459
168	AF	`get_funcretloc`	Fixed 37460
169	AF	`tinlinenode.handle_str`	Fixed 37462
170	AF	`replaceloadnodes`	Fixed 37475
171	AF	`GetCopyAndTypeCheck`	◇ 37476
172	AF	`searchcsedomain`	Fixed 37477
173	AF	`GetCopyAndTypeCheck`	Fixed 37508

Table 3.3: 21 compliers and interpreters of 9 programming languages tested by POLYGLOT. # refers to the TIOBE index, a measurement of the popularity for programming languages [169], and - means that language is not within top 50. * in Version means the git commit hash. #Bug shows the number of reported bugs, confirmed bugs and fixed bugs from left to right.

#	Language	Target	Version	LOC(K)	#Bug
1	C	GCC	10.0	5,956	6/5/1
		Clang	11.0.0	1,578	24/3/2
4	C++	G++	10.0	5,956	4/4/2
		Clang++	11.0.0	1,578	6/0/0
7	JavaScript	V8	8.2.0	811	3/3/2
		JSCore	2.27.4	497	1/1/1
		ChakraCore	1.12.0	690	9/4/0
		Hermes	0.5.0	620	1/1/1
		mujs	9f3e141*	15	1/1/1
		njs	0.4.3	78	4/4/0
		JerryScript	2.4.0	173	5/5/4
		DukTape	2.5.0	238	1/1/1
		QuickJs	32d72d4*	89	1/1/1
8	R	R	4.0.2	851	4/4/4
		pqR	5c6058e*	845	3/1/0
9	PHP	php	8.0.0	1,269	35/27/22
10	SQL	SQLite	3.32	304	27/27/27
41	Lua	lua	5.4.0	31	12/12/12
		luajit	2.1	88	2/2/2
-	Solidity	solc	0.6.3	192	16/16/16
-	Pascal	freepascal	3.3.1	405	8/8/8
Sum	**9**	**21**			**173/136/113**

we perform an in-depth evaluation on the representative processors of four popular languages (two statically typed and two dynamically typed): `Clang` of C, solc of `Solidity`, `ChakraCore` of `JavaScript` and php of PHP. We also use these four processors to conduct the detailed evaluation to compare POLYGLOT with five state-of-the-art fuzzers, including three generic ones (the mutation-based `AFL`, the hybrid `QSYM`, the grammar-based `Nautilus`) and two language-specific ones (`CSmith` of C and `DIE` of `JavaScript`).

Seed Corpus and BNF Grammar. We collect seed corpus from the official GitHub repository of each language processor. Additionally, we collected 71 and 2,598 builtin functions or variables for `JavaScript` and PHP respectively from [170] and [171] using a crawler script. We feed the same seeds to AFL, QSYM, `DIE` and POLYGLOT as initial corpus. `Nautilus` and `CSmith` do not require seed inputs. The BNF grammar POLYGLOT uses is collected and adjusted from ANTLR [168]. The official release of `Nautilus` only

```c
1 struct S { int d; } s;
2 int a, c;
3 int main() {
4   short *e, b[3] = {1, 2, 0};
5   for (a = 0; a < 39; a++)
6     e = b;
7   switch (c){
8     while(--a){
9       do{
10        case 7:
11          c += *(e++);
12      }
13      while (*e);
14    }
15  }
16  int d = 3;
17  return 0;
18 }
```

(a) PoC for case study 1

```c
1 struct S { int d; } s;
2 int a, c;
3 int main() {
4   short *e, b[3] = {1, 2, 0};
5   for (a = 0; a < 39; a++)
6     e = b;
7   if(c == 7){
8     do{
9       do{
10          c += *(e++);
11      }
12      while (*e);
13    }
14    while(--a);
15  }
16  int d = 3;
17  return 0;
18 }
```

(b) Logically equivalent program

Fig. 3.7: PoC and its logically equivalent program for Case Study 1. The code in line 7 to 15 should not be executed because the value of `c` is `0`. However, the development branch of Clang crashes when compiling the PoC with "-O3".

supports the grammars of `JavaScript` and PHP, so we further provide the grammars of C and `Solidity` to `Nautilus`.

Environment Setup. We perform our evaluation on a machine with an AMD EPYC 7352 24-Core processor (2.3GHz), 256GB RAMs, and an Ubuntu 16.04 operating system. We adopt edge coverage as the feedback and use AFL-LLVM mode [12] or AFL-QEMU mode to instrument the tested applications. We enlarge the bitmap to 1024K-bytes to mitigate path collisions [172]. Each tested target is compiled with the default configuration and with debug assertion on. We additionally patch `ChakraCore` to ignore out-of-memory errors, which can be easily triggered by valid test cases, like large-array allocations or recursive function calls, to avoid lots of fake crashes and false invalid test cases in language correctness. For new bug detection, due to the limited computation resource, we tested the 21 targets for different duration, summing up to a total period of about three months. For other evaluations, we run each fuzzing instance (one fuzzer + one target) for 24h and repeat this process five times. Each fuzzing instance is run separately in a docker with 1 CPU and 10G RAM. We perform Mann-Whitney U test [173] to calculate the P-values for the experiments and provide the result in Table 3.5.

3.4.2 Generic Applicability and Identified Bugs

To evaluate the generic applicability of PolyGlot, we applied it on 21 representative processors of 9 programming languages to see whether PolyGlot can thoroughly test them and detect bugs. We only use about 450 lines of BNF grammar and 200 lines of semantic annotations on average for each of the 9 programming languages. As also mentioned in §3.3.4, it only took one of our authors about 5-8 hours to apply PolyGlot on each of the tested programming language.

Identified Bugs. As shown in Table 3.3, PolyGlot has successfully identified 173 bugs in the 21 tested processors of 9 programming languages, including 30 from C, 10 from C++, 26 from `JavaScript`, 35 from PHP, 16 from `Solidity`, 27 from SQL, 14 from LUA, 7 from R and 8 from Pascal. The complete and detailed information of the bugs can be found in Table 3.2 in Appendix. All the bugs have been reported to and acknowledged by the corresponding developers. 113 bugs have been fixed and 18 CVEs are assigned. Most of these bugs exist in the deep logic of the language processors and are only triggerable by semantically correct test cases. In the following case studies, we discuss some of the representative bugs to understand how PolyGlot can find these bugs and what security consequences they cause.

Case Study 1: Triggering Deep Bugs in Clang. PolyGlot identifies a bug in the loop strength reduction optimization of `Clang`. Fig. 3.7(a) shows the Proof-of-Concept (PoC), and Fig. 3.7(b) shows its logically equivalent program for understanding the semantics of the PoC easily. Fig. 3.6 in Appendix shows the process of how PolyGlot turns the benign motivating example (Fig. 3.2(a)) into the bug triggering PoC. After each round of mutation, all the definitions are intact, and new code structures are introduced. Each round of validation produces a semantically correct test case. With new code structures and semantic correctness, the mutated test case keeps discovering new execution paths, which encourages PolyGlot to keep mutating it. And we get the bug triggering PoC after 3 rounds of mutation and validation. The PoC might look uncommon to programmers, but its

```
1 function handler(key, value) {
2     new Uint32Array(this[8] = handler);
3     return 1.8457939563e-314;
4 }   //1.8457939563e-314 is the floating point
5     //representation of 0xdeadbeef
6 JSON.parse("[1, 2, 3, 4]", handler);
```

(a) PoC that hijacks control flow in njs. njs crashes with `RIP` hitting `0xdeadbeef`. The bug is assigned with `CVE-2020-24349`.

```
<?php
  $a = 'x';
  str_replace($a , array () , $GLOBALS );
?>
```

(b) PoC that triggers an invalid memory write in PHP interpreter. This kind of bug does not involve dangerous functions and can be used to escape PHP sandboxes.

Fig. 3.8: POC for case study.

Table 3.4: Distribution of bugs found by evaluated fuzzers. We perform the evaluation for 24 hours and repeat it five times. We report the bugs that appear at least once. "-" means the fuzzers are not applicable to the target. POLYGLOT-*st* refers to POLYGLOT-$syntax$.

Target	**Type** SF: segmentation fault AF: assertion failure UAF: use-after-free SBOF: stack buffer overflow	POLYGLOT	POLYGLOT-*st*	`AFL`	`QSYM`	`Nautilus`	`CSmith`	`DIE`
clang	AF in parser	✗	✗	✔	✗	✗	✗	-
clang	AF in parser	✗	✗	✗	✔	✗	✗	-
clang	AF in code generation	✔	✗	✗	✗	✗	✗	-
clang	SF in optimization	✔	✗	✗	✗	✗	✗	-
`ChakraCore`	SF in JIT compilation	✔	✗	✗	✗	✗	-	✗
`ChakraCore`	AF in JIT compilation	✔	✗	✗	✗	✗	-	✗
php	UAF in string index	✔	✗	✗	✗	✗	-	-
php	SF in setlocale	✔	✔	✗	✗	✗	-	-
php	SF in zend API	✔	✔	✗	✗	✔	-	-
php	SBOF in header callback	✔	✗	✗	✗	✗	-	-
solc	SBOF in recursive struct	✔	✗	✗	✗	✗	-	-

syntax and semantics are legitimate in C. Therefore, the PoC shows that POLYGLOT can generate high-quality inputs to trigger deep bugs in language processors.

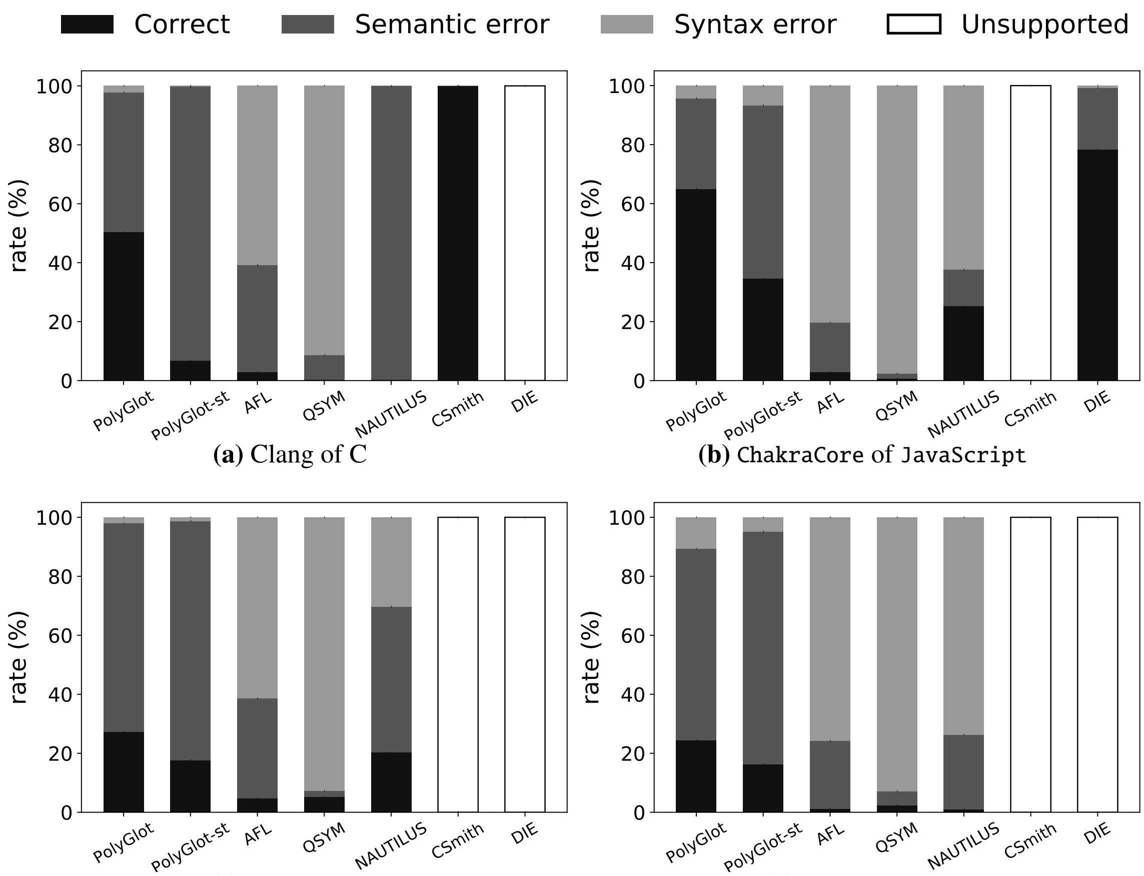
Case Study 2: Control Flow Hijacking in njs. POLYGLOT identified many exploitable bugs, including the one shown in Fig. 3.8(a), which leads to control flow hijacking in njs. In `JavaScript`, when `JSON` parses a string, it accepts a handler to transform the parsed values. In the PoC, `JSON.parse` first parses the string `"[1, 2, 3, 4]"` into an array of four integer elements, which is denoted by `arr`. Then, the handler at line 1 runs on each of the four elements and replaces them with `1.8457939563e-314`. It also modifies `arr`, which is referred by `this` in line 2. Assigning a new element of type `function` to `arr` changes the underlying memory layout of `arr`. After the handler processes the first element, the memory layout of `arr` has changed. However, it is undetected by njs and causes a type confusion. njs still uses the old layout and mistakenly treats the first processed element, which is a user-controllable number, as a function pointer. njs then calls that function and control flow hijacking happens. If an attacker controls the `JavaScript` code, he can utilize this bug to achieve RCE.

Case Study 3: Bypassing PHP Sandbox. The PHP bugs found by POLYGLOT can be used to escape PHP sandboxes. PHP sandboxes usually disable dangerous functions like `"shell_exec"` to prevent users from executing arbitrary commands. The PoCs of our PHP bugs do not involve these functions. Therefore, they are allowed to run in PHP sandboxes such as [174, 175], causing memory corruption and leading to sandbox escape [176, 177, 178].

We show the PoC of one of our bugs in Fig.3.8(b), which only uses a commonly-used and benign function `str_replace`. It triggers an out-of-boundary memory write and crashes the interpreter. With a well-crafted exploiting script, attackers can modify the benign function pointers to dangerous ones. For example, we overwrite the function pointer of `echo` to `shell_exec`. Then calling `echo("ls")`, which should simply print the string `"ls"`, becomes `shell_exec("ls")`. In this way, attackers can escape the sandbox and produce more severe damages. Actually, the security team of Google also considers bugs in PHP interpreter as highly security-related [164]. Therefore, our bugs in PHP interpreters, though not assigned with CVEs, can lead to severe security consequences.

3.4.3 Contributions of Semantic Correctness

To understand the contributions of semantic correctness in fuzzing language processors, we perform unit tests by comparing POLYGLOT and POLYGLOT-$syntax$ which is POLYGLOT without semantic validation. We compare them in three different metrics: the number of unique bugs, language validity, and edge coverage. We evaluate the number of unique bugs as it can better reflect bug finding capabilities of the fuzzers than the number of



(a) Clang of C **(b)** `ChakraCore` of `JavaScript`

(c) php of PHP **(d)** solc of Solidity

Fig.3.9: Rate of language correctness of inputs generated by evaluated fuzzers for 24h. "Correct" means the inputs can be successfully executed or compiled by the language processors. "Syntax error" means the inputs contain syntactic errors. "Semantic error" means the inputs are valid syntactically but not semantically. "Unsupported" mean the fuzzer is not applicable to the target.

unique crashes [160]. For language correctness, we consider a test case as semantically correct as long as it can be compiled (for compilers) or executed (for interpreters) without aborting errors. For example, if a C program uses an uninitialized variable, GCC might still successfully compile the program, so we treat it as a correct test case. This method will treat some semantically incorrect test cases as correct ones (*e.g.*, inputs containing undefined behaviors). We plan to mitigate this problem in future work. We should notice that POLYGLOT-$syntax$ without IR mutation is basically `AFL`, and we leave the comparison in the next section (§3.4.4).

Unique Bugs. We manually map each crash found by each fuzzer in 24 hours to its corresponding bug, and show the result in Table 3.4. POLYGLOT-$syntax$ finds only two bugs in PHP, which are covered by the nine bugs POLYGLOT finds in the four targets. We check the PoCs of the two PHP bugs and find the bugs are triggered by a single function call to a specific built-in function. While POLYGLOT generates such function calls in correct test cases, POLYGLOT-syntax generates them in incorrect ones. These function calls happen to be at the beginning of the PoCs of POLYGLOT-syntax. Since the PHP interpreter parses and executes one statement per time, the bugs are triggered before the interpreter detects errors in later statements. This shows that both POLYGLOT and POLYGLOT-$syntax$ can identify bugs triggerable by simple statements in the PHP interpreter. However, only POLYGLOT detects deeper bugs in optimization in Clang and `ChakraCore` because those bugs can only be triggered by semantically correct test cases.

Language Validity. We show the details of language correctness in Fig.3.9. Compared with POLYGLOT-$syntax$, POLYGLOT improves the language validity by 50% to 642%: 642% in `Clang`, 88% in `ChakraCore`, 54% in `php`, and 50% in `solc`. The result shows the semantic validation greatly improves the semantic correctness of the test cases. The difference in the degree of improvement results from the complexity and the accuracy of the BNF grammar of the language. In POLYGLOT, C and `JavaScript` have 364 and 462 lines of BNF grammar, while PHP and `Solidity` have 802 and 745 respectively. Also, the BNF grammar is a superset of the real grammar the processor accepts. The mutator generates more test cases that cannot be validated due to lower accuracy in the grammar of PHP and `Solidity`. For example, we can use a combination of the keywords `{"pure", "view", "payable"}` to describe a function in `Solidity` as long as the same keyword does not appear twice. However, in BNF it is described as `(pure|view|payable)*` and then `"pure pure"` is legal according to the grammar. Such errors will be treated as semantic errors and cannot be fixed by POLYGLOT currently.

Code Coverage. POLYGLOT identifies 51%, 39%, 23%, 31% more edge paths than POLYGLOT-$syntax$ in `Clang`, `ChakraCore`, `php`, and `solc` respectively. We show the trend of edge coverage in 24h in Fig.4.5. The increase is higher in `Clang` and `ChakraCore` than the

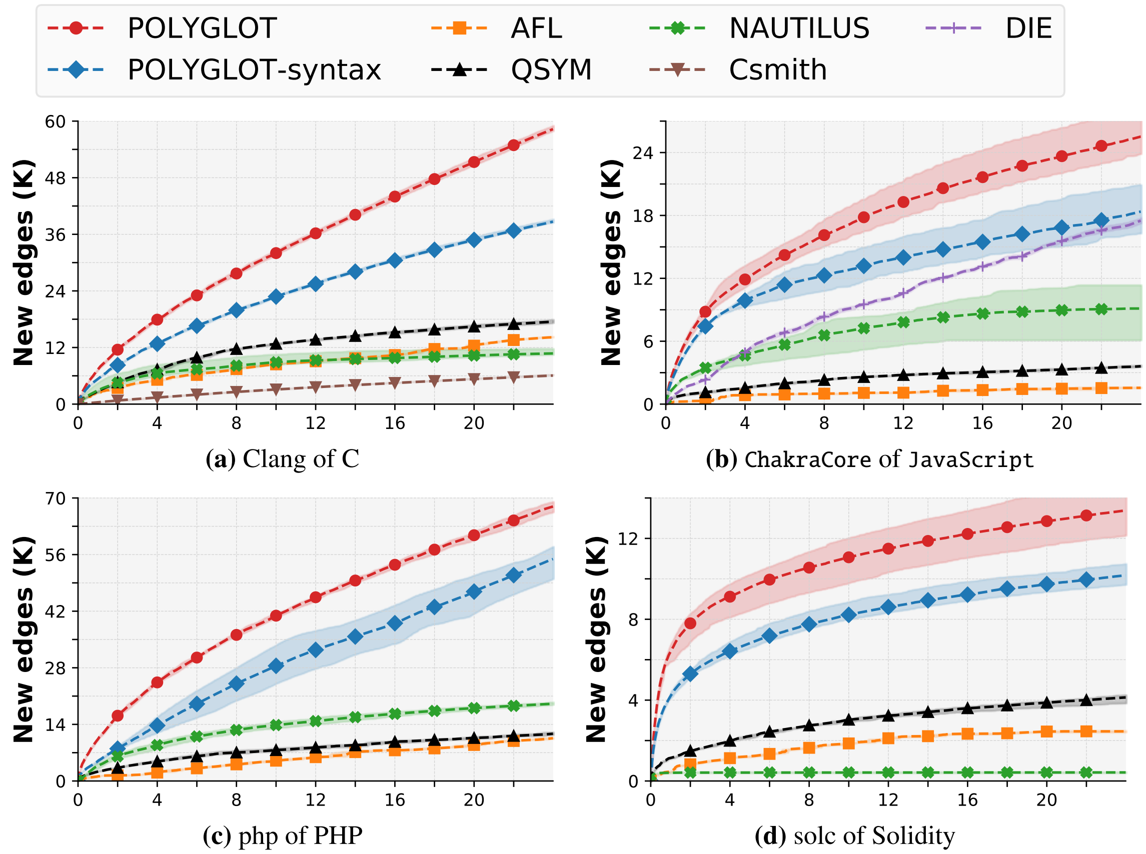


Fig.3.10: Edge coverage found by evaluated fuzzers for 24h. We repeat the experiments 5 times. The solid dot lines represent the mean of the result and the shadow around lines are confidence intervals for five runs.

rest two because `Clang` and `ChakraCore` perform heavy optimization (*e.g.*, `ChakraCore` has JIT compilation). With more semantically correct test cases, POLYGLOT can reach deeper logic to explore the optimization and compilation code. As shown in Table 3.6 in Appendix, POLYGLOT-*syntax* executes about 2× as fast as POLYGLOT, but it still achieves lower coverage. This result shows the semantically correct test cases generated by POLYGLOT are more effective in exploring the deep program states.

Overall, POLYGLOT outperforms POLYGLOT-*syntax* in the number of unique bugs, language validity, and code coverage. As they use the same mutation strategies, they generate test cases with similar code structures. Under this condition, higher language validity further improves the fuzzing performance in various dimensions, showing the importance of semantic correctness in testing deep logic.

3.4.4 Comparison with State-of-the-art Fuzzers

We also compare POLYGLOT with five state-of-the-art fuzzers to further understand its strengths and weaknesses in testing language processors, including the mutation-based fuzzer `AFL`, the hybrid fuzzer `QSYM`, the grammar-based fuzzer `Nautilus`, and two language-specific fuzzers `CSmith` and `DIE`.

Unique Bugs. POLYGLOT successfully identifies nine bugs in the four targets in 24 hours: two in `Clang`, two in `ChakraCore`, four in `php` and one in `solc`, as shown in Table 3.4. `AFL` and `QSYM` only identify one bug in clang respectively. `Nautilus` detects one in the `php` interpreter, which is also covered by POLYGLOT. `CSmith` and `DIE` find no bugs in 24 hours. The bugs found by `AFL` and `QSYM` exist in the parser of `Clang`. We check the PoCs and find them invalid in syntax: the bugs are triggered by some unprintable characters. POLYGLOT does not find such bugs because its goal is to find deeper bugs with valid test cases. In fact, it does find bugs in the optimization logic of `Clang` such as the one in Case Study 1 (Fig. 3.7(a)), proving its effectiveness in finding deep bugs.

Language Validity. Compared with the three general-purpose fuzzers (`AFL`, `QSYM`, and `Nautilus`), POLYGLOT improves the language validity by 34% to 10,000%, as shown in Fig. 3.9. Compared with the language-specific fuzzers, POLYGLOT gets 53% and 83% as much as that of `CSmith` and `DIE` respectively. We investigate the result and find the reasons as follows. `AFL` and `QSYM` do not aim to improve the language validity as POLYGLOT does. `Nautilus` uses a small number of fixed variable names and relies on name collision to generate correct input, which turns out to be less effective. `CSmith` and `DIE` perform much heavier and more specialized analyses than POLYGLOT in one specific language and thus achieve higher validity in that language.

Code Coverage against General-purpose Fuzzers. As shown in Fig. 4.5, POLYGLOT identifies 230% to 3,064% more new edges than the three compared general-purpose fuzzers: up to 442% more in `Clang`, 542% more in `php`, 1,543% more in `ChakraCore`, and 3,064%

Table 3.5: P-values of POLYGLOT *v.s.* other fuzzers. We use Mann-Whitney U test to calculate the P-values. P-values less than 0.05 mean statistical significance. The result of nearly all the experiments is statistically significant except for the language correctness compared with `CSmith` and `DIE`.

v.s. Fuzzer	Target	Coverage	Correctness	Bugs
POLYGLOT-*st*	`Clang`	0.00596	0.00545	0.00198
	`ChakraCore`	0.00609	0.00583	0.00198
	php	0.00609	0.00609	0.00520
	solc	0.00609	0.00596	0.00198
AFL	`Clang`	0.00596	0.00545	0.00279
	`ChakraCore`	0.00609	0.00583	0.00325
	php	0.00609	0.00609	0.00325
	solc	0.00609	0.00596	0.00198
QSYM	`Clang`	0.00609	0.00485	0.00325
	`ChakraCore`	0.00609	0.00609	0.00325
	php	0.00609	0.00609	0.00325
	solc	0.00609	0.00596	0.00198
Nautilus	`Clang`	0.00609	0.00558	0.00198
	`ChakraCore`	0.00609	0.00609	0.00198
	php	0.00609	0.00609	0.00485
	solc	0.00609	0.00558	0.00198
CSmith	`Clang`	0.00609	0.998	0.00198
DIE	`ChakraCore`	0.00596	0.996	0.00198

more in `solc`. If we look at the execution speed of `AFL`, `QSYM`, and `Nautilus` (Table 3.6), we can see that they all execute faster than POLYGLOT. There are several reasons that might lead to the performance gap. First, POLYGLOT puts more effort in analyzing mutated test cases and fixing semantic errors so its test case generation takes more time. Second, test cases of higher semantic correctness lead to longer processing time, because language errors cause the execution to bail out early. As we see earlier, the test cases POLYGLOT generates have a higher rate of language correctness and thus lead to slower execution. However, POLYGLOT still achieves much higher coverage, showing that POLYGLOT can generate high-quality test cases to effectively explore program states with a reasonable loss in efficiency of test case generation.

Code Coverage against Language-specific Fuzzers. As shown in Fig.4.5, POLYGLOT finds 863% more edges than `CSmith` in `Clang` and 46% more than `DIE` in `ChakraCore`. We should notice that `CSmith` and `DIE` actually have higher rate of language validity (Fig.3.9). We analyze the results and find the following reasons. First, both `CSmith` and `DIE` execute

Table 3.6: Execution speed of different fuzzers on the four tested programs within 24 hours. We repeat the experiment five times and report the average result. The number represent "Executions/Second".

Fuzzer	Clang	ChakraCore	php	solc
POLYGLOT	5.39	5.56	20.55	20.08
POLYGLOT-syntax	10.15	9.09	57.73	47.96
AFL	24.29	35.38	102.71	103.71
QSYM	10.39	10.60	37.94	52.95
Nautilus	40.59	66.37	115.07	185.24
CSmith	0.34	-	-	-
DIE	-	4.90	-	-

more slowly than POLYGLOT (Table 3.6 in Appendix). This is because CSmith and DIE adopt heavier and more complex analyses than POLYGLOT (*e.g.*, CSmith has 80k lines of code). Also, as discussed before, higher language validity may lead to slower execution. Second, CSmith generates test cases randomly without utilizing guidance, so it might generate similar test cases to keep exploring the same program logic. To confirm our speculation, we perform extra evaluations by comparing CSmith and POLYGLOT in the same conditions: we disable the feedback guidance of POLYGLOT, which is denoted by POLYGLOT-$nofeedback$, as CSmith has no guidance; we randomly collect 5,000 test cases generated by POLYGLOT-$nofeedback$ and CSmith to eliminate the effect of different execution speeds. We measure the language validity and code coverage in Clang and repeat the process five times. POLYGLOT-$nofeedback$ gets 63.8% of language validity, while CSmith keeps its 100% correctness. 5,000 test cases of POLYGLOT-$nofeedback$ and CSmith identify 672 and 1809 edges respectively. The results show our assumption that higher language validity helps explore more program states is still valid, but there are other aspects that also affect the code coverage of fuzzing (*e.g.*, feedback guidance, execution speed, code structures of test cases).

Overall, POLYGLOT outperforms the three compared general-purpose fuzzers in the evaluated metrics and also outperforms the language-specific testing tools in the number of bugs and edge coverage. The fuzzing effectiveness of POLYGLOT comes from both its constrained mutation and semantic validation. The mutation introduces various new code structures, while the validation further improves the quality of the test cases. Considering its generic applicability, we believe POLYGLOT can save huge development efforts from developers and boost the testing of language processors.

3.5 Discussion

We present several limitations of the current implementation of POLYGLOT and discuss their possible solutions.

Limitation of Scope/Type System. POLYGLOT relies on static analysis to collect type and scope information. Therefore, the scope system of POLYGLOT only handles lexical scopes but not dymanic scopes [179], and the information collected by type system on dynamically-typed programming languages might be imprecise as those types can only determined at run time. To overcome this problem, we can utilize dymanic execution to collect runtime information of the seed inputs before fuzzing to assist the analysis of POLYGLOT. Also, as POLYGLOT tries to be general, its scope and type system currently focus on common features shared by popular languages. To support some language-specific features, such as the ownership in Rust [180], we need to specialize POLYGLOT case by case. This is not our goal, but we believe that users can easily encode the extra semantic information in the IR with semantic annotations and extend the type and scope system according to their needs.

Inconsistent Grammar. POLYGLOT accepts a BNF grammar as input and generates test cases that follow the grammar. However, it still generates syntactically incorrect test cases as shown in Fig. 3.9, because a BNF grammar is usually a superset of the real grammar that language processors accept. For example, in C we can use "`(int|long|void)+ identifier`" to describe the grammar of variable definitions, where "+" means one or more. The "+" is intended for types like `long int` and `short int`, but invalid types like `long void` and `void int` are also valid according to the BNF grammar, which introduces incorrect test cases. To address this problem, we plan to adopt techniques that infer accurate grammar in runtime [181, 182]. Alternatively, we can try machine learning techniques to infer the accurate input grammar from test cases [183, 184, 185].

Supporting More Semantics. POLYGLOT improves language validity by ensuring that we use definitions of correct scopes and types. We plan to support more common semantics to further improve semantic correctness. For example, we use a variable only after it is initialized, which can reduce the frequency of undefined behaviors in programming languages such as C and Pascal. Also, we can extend our symbol tables to allow variable shadowing, which allows variables of the same name to exist in different scopes. Furthermore, instead of only using the type from variable definitions, we can track when their types change with newly assigned values.

More Relaxed Mutation. POLYGLOT restricts its mutation to preserve language correctness. However, this restriction limits the possible definitions and code structures because we hardly mutate IRs with definitions. We plan to relax the constraints in the following ways. First, we can generate and insert new definitions into the test cases. This can enrich the possible definitions of the test cases and bring more code structures. Second, we can perform test case minimization to remove definitions that are not used in the program. This also increases the possibility of mutation. For example, an IR might not be mutable because it contains a definition. If the definition is not used and removed during minimization, the IR becomes mutable.

CHAPTER 4

μFUZZ: A PARALLEL FUZZING FRAMEWORK WITH MICROSERVICE ARCHITECTURE

In this chapter, we present μFUZZ, a parallel fuzzing framework that reimagines parallel fuzzing through a microservice architecture, scaling the program exploration capability horizontally.

4.1 Introduction

In recent years, fuzzing has been widely adopted to detect security bugs [12, 47, 8, 72]. Compared with other program analysis techniques, fuzzing ensures higher throughput while requiring less manual effort and pre-knowledge of the target software. In addition, fuzzing is demonstrated to be practical for detecting security issues in complex, real-world programs [12, 4]. Thus, considerable computing resources are used for fuzzing in industry. For example, Google implemented clusterfuzz [127] in 2016, and over 36,000 bugs have been found through this project.

To improve the fuzzing efficiency, researchers propose various optimizations to enhance fuzzers' internal components [186, 187]. For instance, several projects implement grammar-based, adaptive or unified mutators to generate more valid, effective and diverse test cases [6, 188, 189, 48]. Hybrid fuzzing utilizes heavy program analysis techniques to extract useful information to help explore program state space [161, 40, 58, 43, 116]. Various algorithms are developed to adjust the input priority to make a balance between input space exploration and exploitation [86, 190, 191]. Researchers also design and implement different feedback mechanisms to promote the fuzzing speed and effectiveness [192, 83, 193]. These internal improvements have dramatically increased the performance of a single fuzzing instance.

In addition to improving internal procedures, researchers also set sights on parallel fuzzing. The goal of parallel fuzzing is to make full use of resources and detect more bugs within a shorter time than single-instance fuzzing. For example, as fuzzing shows its ability in bug detection, many companies such as Google and MicroSoft decide to invest numerous resources (*e.g.*, CPU and memory) in fuzzing [127, 194, 128, 4]. State-of-the-art parallel fuzzing approaches share a similar architecture [12, 2, 10, 11]. They launch multiple fuzzing instances in separate processes and periodically perform corpus synchronization. Each instance follows the original logic of the underlying *single-instance fuzzer*, which is designed to run as a single instance. For example, it adopts a serial fuzzing loop which first takes one test case from the input queue, then mutates the input to generate new ones, and finally runs the program with the new input while collecting feedback. Each instance maintains its own fuzzing states such as the code coverage bitmap and a corpus of interesting test cases. The advantage of this parallel-fuzzing architecture comes from state synchronization, which allows one instance to catch up on the latest progress from other instances. In this way, all instances contribute to the program state exploration and bug detection.

However, we identify two limitations in the current parallel fuzzing architecture. First, the existing architecture is built on top of single-instance fuzzers, whose fuzzing logic may not be suitable for parallel fuzzing purposes. These single-instance fuzzers adopt a serial synchronous loop, where the input generation and consumption must follow the order. Once a procedure (*e.g.*, test case execution) in this loop gets blocked (*e.g.*, by file or network I/O), the whole instance gets stuck. The CPU bound to the fuzzing instance will be idle or spinning (*e.g.*, keep looping for a lock access) until the blocking operation completes. As parallel fuzzing runs multiple instances at the same time and introduces more I/O by synchronization, the instances are more likely to get stuck, wasting more CPU cycles. We should reuse the wasted CPU cycles to fully utilize the computation power.

Second, existing approaches periodically synchronize the corpus from each other to allow instances with slow progress to catch up. The synchronization will update the local fuzzing states for all instances so that they can use the latest information to make globally beneficial fuzzing decisions. However, these state updates are not timely enough. In the time window between two consecutive synchronizations, each instance has to use the local information to make decisions. Since local information could be out-of-date, such decisions are not necessarily beneficial from the global perspective. After running fuzzing instances for a long time, the accumulated non-optimal decisions could significantly reduce the fuzzing efficacy. Increasing the frequency of synchronization could mitigate this problem. However, as demonstrated in the previous work [126], frequent synchronization brings heavy overhead, which will reduce the fuzzing efficiency.

To overcome the limitations caused by the current architecture, we need to redesign fuzzing tools to reduce the burdens of serialization and synchronization. Fortunately, we find our opportunity in *microservice architecture* [195]. Microservice architecture organizes tasks in a set of loosely coupled, self-contained services that can run concurrently with others. If no service is blocked, all services collaborate with each other according to the loose dependency. Once a running service is blocked, other services can take over the computing resources to make individual progress. Moreover, each service will maintain its own state and only needs to share minimal information with others in rare cases. Most of the time, each service can make globally optimal decisions.

In this chapter, we propose μFUZZ, a parallel fuzzing framework using the microservice architecture. To adopt this new architecture, we break the current serial fuzzing loop into four microservices, *i.e.*, corpus management, test case generation, test case execution, and feedback collection. Each microservice is self-contained and can schedule parallel workers by itself. We further design an output cache mechanism to reduce the coupling between different services (*i.e.*, decouple input generation and consumption). In this case, if one consumer service gets stuck, the producer service can still make progress and save results into the cache. Similarly, the consumer services can retrieve results from the caches even if the producer service gets stuck. This effectively addresses the CPU cycle wasting issue since each microservice is loosely coupled and can replace the blocked service for execution.

To address the challenges caused by synchronization delay, we design two levels of state partition in μFUZZ. First, μFUZZ splits the global state into different service states so that each service can use its state locally. For example, the coverage bitmap will be put into the feedback collection service as it will evaluate the code coverage and update the bitmap according to the execution status. Second, different workers in each service handle unique parts of the service states. Accumulating all worker states will obtain the service states. These partitions avoid the state synchronization among the workers and enable each service worker to use up-to-date information to make overall good decisions.

We implement μFUZZ in 7,016 lines of Rust code, consisting of the concurrent infrastructure (*i.e.*, the asynchronous runtime) and the fuzzer. The concurrent infrastructure is built on top of Tokio [196], a well-tested asynchronous runtime library. For the fuzzer, we adopt the fork-server execution, havoc mutation, and edge coverage feedback from `AFLplusplus`, and use a simple round-robin algorithm that favors test cases finding more new code for seed selection.

To understand the effectiveness of our new design, we evaluate μFUZZ on two popular benchmarks: Magma [197] and FuzzBench [198]. We compare μFUZZ with the state-of-the-art parallel fuzzers, including `AFLplusplus`, `AFLEdge` and `AFLTeam`, and find μFUZZ can explore 24% more program states and 33% more bugs than the second-best fuzzer in 24 hours. Besides, our experiments show that different aspects of the microservice architecture contribute to the improvement of μFUZZ. Moreover, μFUZZ found 11 new bugs in well-tested real-world programs.

4.2 Problem

In this section, we first briefly describe how state-of-the-art parallel fuzzing approaches work and then discuss their limitations. Next, we show the potential of using microservice architecture to mitigate the limitations of parallel fuzzing. Finally, we present our novel

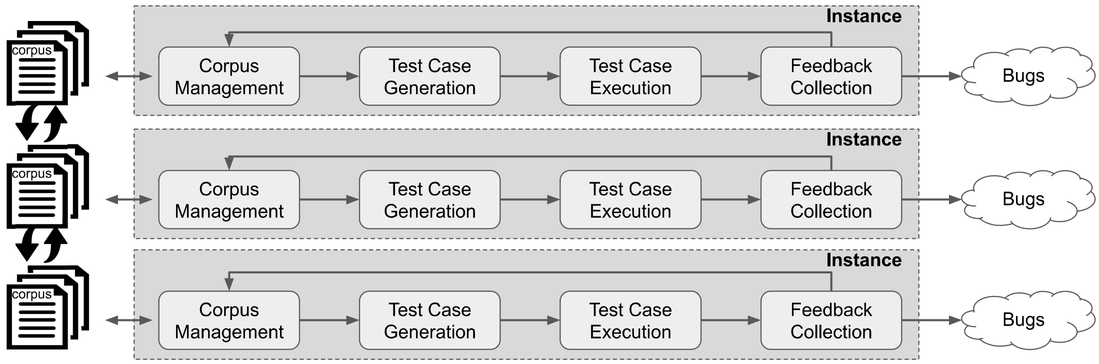


Fig.4.1: The state-of-the-art parallel fuzzing approach. The fuzzer spawns multiple instances and runs them in parallel. Each instance is self-contained and functionality-complete. They maintain their own local fuzzing states, such as the corpus and coverage bitmap. Most of the time, the instances work independently as if there were no other instances. Occasionally, the instances perform corpus synchronization with each other to share their fuzzing progress.

approach to solving the problem.

4.2.1 How Existing Parallel Fuzzing Works

To better test complex programs with time constraints [128, 199], many fuzzers [12, 2, 114, 54, 39] support parallel fuzzing mode to boost the fuzzing performance. The state-of-the-art approach is to run multiple fuzzing instances of the same fuzzer independently on multiple CPU cores. The instances perform periodic corpus synchronization with each other because the corpus represents the fuzzing progress of an instance. Synchronizing the corpus allows the latest progress made by one instance to be caught up by the others and guide their work [12]. As shown in Fig. 4.1, each instance maintains a local seed corpus. Most of the time, these instances run independently, as if there are no other instances. Occasionally, they check others' seeds and copy those that trigger new code to their own corpus. Advanced parallel fuzzing approaches either run instances of different fuzzers to combine their capability [125, 132, 133] or further optimize the corpus distribution strategy by partitioning the synchronized corpus among instances to avoid duplicated fuzzing efforts [11, 10, 122].

Fuzzing State. Corpus synchronization improves fuzzing because the corpus is part of the fuzzing state. We define the fuzzing state of a fuzzing instance to be the minimum

Table 4.1: The percentage of wasted CPU cycles by blocking I/O in single instance fuzzing.

Target	`lua`	`PHP`	tcpdump	MySQL
Wasted Cycles (%)	1.2%	0.5%	28.7%	60.3%

Table 4.2: The percentage of wasted CPU cycles by blocking I/O in parallel fuzzing. We measure the blocking cycles in the main fuzzing instance with `AFLplusplus` during corpus synchronization with 1-second, 1-minute and 30-minute synchronization intervals and different number of instances.

Target	`lua`			`PHP`		
Interval\Instance	20	40	60	20	40	60
1 sec	3.18%	7.29%	8.74%	13.54%	25.79%	29.70%
1 min	0.13%	0.30%	0.31%	1.34%	2.47%	3.34%
30 min	0.07%	0.08%	0.15%	0.58%	0.76%	1.01%

information to represent its full fuzzing progress. They might include the corpus, average running time of the test case executing, seeds of the random number generator, etc.

4.2.2 Limitation of Existing Approaches

Existing parallel fuzzers maintain a local fuzzing state in each instance and perform state synchronization periodically. Such approaches mainly have two problems: First, it aggravates the problem of wasting CPU cycles due to the serial design of the underlying fuzzer. Second, the global fuzzing state cannot be synchronized to each instance timely or efficiently, resulting in suboptimal performance.

CPU Cycles Wasted due to Blocking I/O. Existing fuzzers run their instances in a serial synchronous loop [12, 2, 114]. For example, the fuzzing pipeline of `AFLplusplus` is as follows: Select a test case, mutate it, execute it, check the execution feedback, and loop. If any of the steps are blocked by I/O, the other steps can do nothing but wait. Therefore, such a design might suffer from performance degradation in the existence of blocking I/O. I/O can come from two sources. First, the tested program can involve heavy blocking I/O (e.g., a compression application might do heavy file I/O.). During the execution phase, the fuzzing loop can get stuck, waiting for the I/O to complete. Since the fuzzing loop is synchronous, the CPU cannot perform other fuzzing tasks but wait, wasting CPU cycles. Second, when fuzzing with multiple instances, state synchronization might also bring in lots of I/O [200]. Take `AFLplusplus` as an example. When running in parallel mode, each instance periodically checks and synchronizes the corpus with other instances in a shared folder. This has been shown to bring lots of I/O, such as shared folder locking and file copying, which hurts the fuzzing performance [126].

We performed quick experiments on four popular programs to investigate these two sources of I/O. We measure the percentage of CPU cycles wasted by blocking I/O, where the CPU is either idle or spinning waiting for gaining locks or I/O completion. Table 4.1 shows the result in single instance fuzzing. As we can see, if the tested target involves lots of blocking I/O (*e.g.*, `tcpdump` calls the system call `poll` and waits), the wasted percentage can be significant. For targets without much I/O (*e.g.*, `lua` and `PHP`), we futher evaluated them for the main fuzzing instance (*i.e.*, the instance that performs synchronization with all other instances) in parallel fuzzing with respect to different synchronization intervals and number of instances. As shown in Table 4.2, even for targets with little inherent I/O operations, if we perform heavy state synchronization (with more cores or higher frequency), the introduced I/O significantly degrades the CPU utilization. Simply spawning more instances on the same CPU (*i.e.*, oversubscription) cannot solve the problem because it might introduce more resource contention such as context-switching and again hurt the performance.

Fuzzing State Not Timely Synchronized. The instances maintain their local fuzzing states and perform periodic synchronization. Before the next synchronization, they use the possibly outdated states and fuzz with strategies which are locally optimal but can be globally suboptimal. On the other hand, it is not feasible to perform synchronizations too frequently as it incurs a significant overhead [126].

We did a quick experiment to verify our hypothesis. We used `AFLplusplus` to fuzz `QuickJS` [201], a popular JavaScript engine. As a comparison, we fuzzed with one instance of `AFLplusplus` for ten hours and ten instances in parallel for one hour, respectively. The measured metrics include the code coverage and the number of interesting test cases that are further selected for fuzzing. The result shows that if we fuzz with a single instance for 10 hours, about 13,700 new program paths are found, and about 80% of the interesting test cases are further used for fuzzing. However, when fuzzing with ten instances in parallel for an hour, we only find about 6,700 new program paths, which is only 49% of that of a single instance. About only 40% of the test cases are further selected for fuzzing. We assume the fuzzing strategy of the single-instance fuzzer is optimal. That means the ten instances use suboptimal strategies and duplicate their works on similar test cases, while the globally optimal strategy is to explore test cases diversely. Similar results are also found in [11].

To further verify that the performance gap is caused by synchronization delay, we change the synchronization frequency of `AFLplusplus` and measure the change in the fuzzing performance in terms of code coverage. More specifically, we fuzz `QuickJS` with ten `AFLplusplus` instances for one hour by setting their synchronization frequency per hour from 2 (`AFLplusplus`'s default setting) to 40,000 (which performs synchronization after every test case execution). The result is shown in Fig. 4.2. As we can see, if the frequency is too low, the code coverage is also low because the instances are using suboptimal fuzzing strategies. If the synchronization frequency is high, the code coverage also drops dramatically because the overhead of synchronization is too high. However, even the best result in the curve is still much worse than that of the single instance fuzzing. This means that simply changing the synchronization frequency does not solve the problem.

From the above discussion, we want a parallel fuzzing framework that supports concurrency to better utilize CPU cycles even in the existence of I/O and can synchronize instances' states timely with little overhead so that we fuzz with up-to-date states and make good decisions. However, it is difficult to do so on top of existing fuzzers with monolithic serial architecture. We need a different architecture.

4.2.3 Microservice Architecture

We find microservice architecture [195] fits parallel fuzzing well and can potentially mitigate its current limitations. First, microservice architecture structures the application as a set of small, loosely coupled, collaborating services. These services run concurrently with others. For parallel fuzzing, we can break the different phases in the serial fuzzing loop

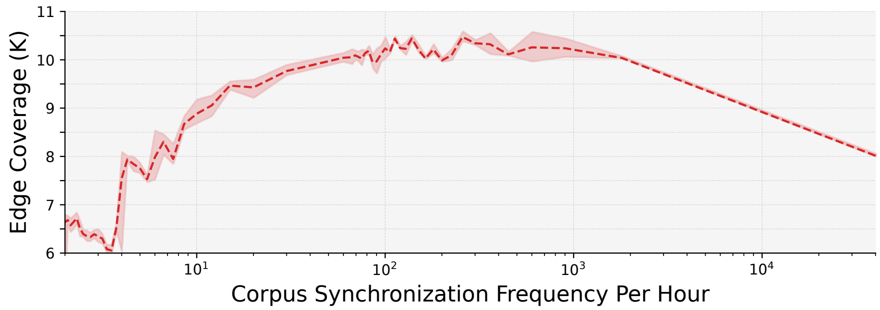


Fig. 4.2: Code coverage of 10 `AFLplusplus` instances testing `QuickJS` with different synchronization frequency in an hour.

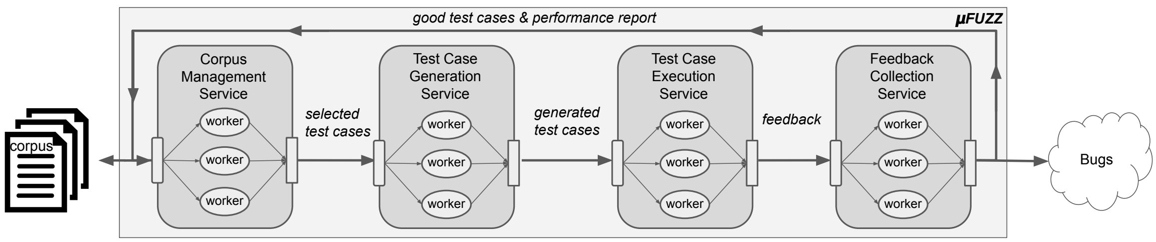


Fig. 4.3: Overview of μFUZZ. Instead of running multiple fuzzing instances and performing periodic synchronization, μFUZZ breaks the traditional monolithic architecture into microservices. The new architecture consists of four self-contained services, each maintaining a partition of the fuzzing state. The services are loosely dependent on each other using output caching. Inside each service, we run multiple workers to exploit parallelism.

into concurrent services, where we might run other services if one gets stuck. Second, the services are self-contained (*i.e.*, it does not rely on others to finish its job), which means it does not need to synchronize with others. For parallel fuzzing, the services can be self-contained if each of them focuses on a single functionality of fuzzing (*e.g.*, test case generation). And we do not need state synchronization among the services. Third, inside a service, we can easily scale the capability by creating multiple instances and partitioning the service data among the instances. For parallel fuzzing, we can create multiple workers inside a service to achieve parallelism and partition the service state maintains among the workers. And these workers do not need to synchronize with each other because their states have no overlap. We only need to ensure that the workers can work independently using their local states and still achieve a good aggregated result.

4.2.4 Our Approach

This chapter aims to design a parallel fuzzing framework that embraces concurrency to better utilize CPU power even with blocking I/O and avoids synchronization but still gets overall great performance. We achieve our goal in two steps: redesigning the fuzzing framework with microservice architecture and partitioning the fuzzing state. Microservice architecture adds concurrency to the framework and enables the fuzzer to effectively utilize the CPU power in the existence of I/O. Fuzzing state partition allows the instances to fuzz with locally optimal strategy and still achieve an overall globally great performance without synchronization.

Redesign with Microservice Architecture. We break the traditional serial fuzzing loop into four services based on functionality: Corpus management, test case generation, test case execution, feedback collection. The whole fuzzing state is also partitioned among the services in a way that each service only needs its partition to function and is thus self-contained. However, these services are still tightly coupled: every service produces output for other services to consume and vice versa. Instead of running the services synchronously, we utilize output caching to decouple production and consumption so that they can run concurrently. When one service gets stuck, other services can still make progress and cache the outputs. After the stuck service is ready to run again, it can directly consume the cached outputs without waiting for the producer to generate them. In this way, we can better utilize the CPU power even with blocking I/O.

Partition the Fuzzing State. We have performed the first level of fuzzing state partition by breaking down the monolithic structure. Now each service maintains its own service state. However, if the state is shared by the workers, we still need state synchronization among the workers. We further partition the service state among the workers to avoid synchronization. We use two rules to guide the partition. First, each partition of the state should be functionality-complete, which means a worker can finish its job without using others' states. Second, if each worker adopts its locally optimal strategy, we expect to get a globally great (*i.e.*, close to optimal) aggregated result. In this way, the workers can run independently and do not need synchronization with others. Since we only have one global and distributive fuzzing state, the state changes are directly applied to the states inside the workers. Therefore, the workers always fuzz with the update-to-date global state and make good fuzzing decisions. This avoids the problem of periodic synchronization, which suffers from either high overhead or large synchronization lagging.

4.3 Design & Implementation

Fig. 5.1 shows the overview of μFUZZ. We first break the traditional serial fuzzing loop into four services (§4.3.1). This step partitions the responsibility and the state of the fuzzer among different services so that they do not need state synchronization with each other. These self-contained services are the candidates for concurrency. Next, we utilize output caching to allow the services to run concurrently (§4.3.2) and achieve maximum parallelism with load balancing (§4.3.3). Then we further perform state partitioning among workers so that each worker maintains a self-contained partition of the fuzzing state (§4.3.4). This allows the workers to avoid synchronization with each other but still get an overall great aggregated result. Finally, we connect the services together with zero-copy communication (§4.3.5) to achieve efficient parallel fuzzing.

4.3.1 From Monolith to Microservice

As the first step to support concurrency, we break the monolithic serial fuzzing loop into multiple services, whose structures are shown in Fig. 4.4. We use the following guidelines from the microservice architecture to conduct the breakdown. First, each service should focus only on one core functionality of the fuzzing (*i.e.*, be micro). Second, the services should be self-contained, which means they should not rely on the states of other services to function. If any part of the fuzzing state is used by a service, then it should be maintained by the service. As a result, we classify four core functionalities from the fuzzing loop and

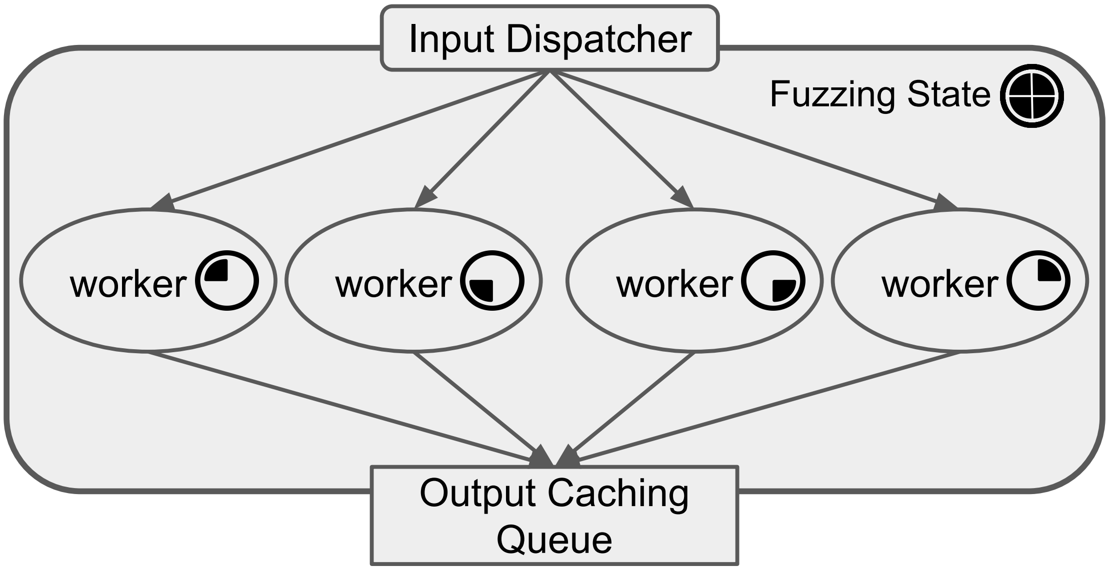


Fig. 4.4: The internal structure of a service in μFUZZ. Each service has a fuzzing state, an input queue dispatcher, an output caching queue, and some workers. The fuzzing state is partitioned among the workers. The input queue dispatcher accepts requests from other services and dispatches them to the workers. The workers handle the inputs in parallel and send the results to the output caching queue. The consumer services consume these results when they are ready.

break them into four services, which are listed below:

Corpus Management Service. It is responsible for performing test case scheduling and maintaining a corpus of interesting test cases and their associated metadata (*e.g.*, performance scores). The corpus can include test cases finding new code coverage or triggering new bugs, etc. Based on the metadata of the test cases, the scheduling algorithm prioritizes those can better explore the program for test case generation.

Test Case Generation Service. It generates new test cases to fuzz the tested program either from scratch or by mutating existing ones. For example, it can utilize the BNF grammar to generate structured inputs or bit-flip existing test cases to generate new variants.

Execution Service. It executes the tested target with the generated test cases and generates necessary feedback such as the code coverage information and whether the tested program crashes or timeouts during the execution.

Feedback Collection Service. It collects the feedback from the execution service and classifies whether the feedback is interesting or not. This information can be used to decide whether a test case should be added to the corpus. It can also generate fuzzing statistics in different metrics for other services to improve their strategies. For example, it can calculate how many interesting test cases are generated from a specific seed and report that to the corpus management service. The corpus management service can then utilize the statistics to update the performance scores of the corresponding test cases and fine-tune its scheduling algorithm.

We see these services are dependent on each other and form a loop: each service consumes some outputs from other services and also produces some for them. Without further changes, these services still need to run in a serial way that one getting stuck blocks the overall progress. We need to loosen the coupling between the services so that they can run concurrently and mitigate CPU cycle wasting, as described in the next section.

4.3.2 Concurrency by Output Caching

Each service is both a producer (produces inputs for other services) and a consumer (consumes outputs from other services). We decouple the production and consumption of each service by output caching so that the services can run concurrently. More specifically, we connect the services with an output caching queue, as shown in Fig. 4.4. When a service produces some results, it first sends them to the output queue instead of to the consumer service directly. If the consumer service is busy temporarily, the results just stay in the queue, and the producer service is free to produce more results. Once the consumer service is ready to process new inputs, it can directly fetch the cached ones from the output queue. In this way, services can run concurrently. When one gets blocked, others can still run and make progress.

Congestion Control. If we allow unlimited output caching, one potential problem is that one service might keep generating outputs and fully occupy all the CPU cores. Under this situation, other services have no chance to run and consume these cached outputs. And the fuzzing cannot make overall progress. For example, the corpus management service can keep selecting test cases for mutation and send them to the queue. And the test case generation service can not consume them as all the CPU cores are busy running the corpus management service. Therefore, we adopt congestion control by limiting the maximum number of cached results in the queue. When the producer service finds that the output queue is full, it knows that the consumer service needs more time to process the cached outputs. Then it will yield to the scheduler so that other services can run. In this way, the rate of production and consumption can reach a dynamic balance, and the fuzzing can make smooth progress continuously.

4.3.3 Parallelism by Load Balancing

To fully utilize the computation power of multiple cores, each service of μFUZZ can run multiple workers in parallel. To achieve maximum parallelism, we set the number of workers to be the number of cores, and we perform load balancing with an input dispatcher to keep all workers busy.

The input dispatcher maintains a first-in-first-out queue of idle workers and adopts two strategies of load balancing: "first come, first served" and dynamic input resizing. We define a worker as idle if it is ready to process but not currently processing inputs. Such workers notify the input dispatcher, which puts them into the back of the queue in order. Whenever an input arrives, the input dispatcher tries to pop an idle worker from the front of the queue and dispatch the input to it, which is "first come, first served." If the queue is empty, which means all workers are busy, the input dispatcher will wait for a worker to become idle. This strategy works well for most cases. However, the sizes of the incoming inputs are not fixed, and sometimes they can be very large. If we simply dispatch an input to one worker, it might result in one worker processing a large input while other workers are idle. To avoid this situation, we further perform dynamic input resizing before dispatching. If the size of the arrived input is larger than a threshold value and there are more than one idle worker, we partition the input evenly based on the number of idle workers and dispatch one partition to one idle worker. With the two strategies, we can achieve maximum parallelism by keeping

Algorithm 1: State Partition & Update

```
Procedure StatePartition(state)
    numWorkers ← The number of workers in the service;
    if state is fixed-sized then                              // Static parittion
        partitions ← Evenly partition the state based on numWorkers ;
    else                                                      // Dynamic partition
        state ← RandomlyShuffle(state) ;
        partitions ← Evenly partition the state based on numWorkers ;
    Distribute one partition to one worker ;
    The worker tags the partitioned state with its ID and store it
Procedure StateUpdate(stateUpdateRequest)
    id2WorkerMap ← Map(ID, worker);
    id ← ExtractID(stateUpdateRequest) ;
    worker ← id2WorkerMap.Get(id) ;
    worker.ProcessRequest(stateUpdateRequest)
```

the workers' workload balanced dynamically.

4.3.4 Avoid Synchronization by State Partition

As discussed, if the fuzzing instances maintain their local states and rely on periodic synchronization, they suffer from either synchronization lagging or high overhead. Therefore, we adopt state partition to maintain only one global state without synchronization.

μFUZZ performs two level of state partition: In the first level μFUZZ partitions the fuzzing states among its services so that the services can work independently. In the second level μFUZZ further partitions each service state among the workers so that each worker can work independently. We already perform the first level by breaking the fuzzing loop into services. μFUZZ partitions its two fuzzing states in two of its services: The corpus management service maintains the interesting testcases with their metadata (e.g., the execution time) and the feedback collection service maintains the code coverage bitmap. The test case generation service and the execution service are stateless. We further perform the second level state partition inside each service. The goal is that each worker maintains a unique partition of the service state and can run independently without synchronizating with others. Meanwhile, accumulating the results from all workers still gives a good overall result.

State Partition. We perform two types of partition based on the size of the fuzzing states: static partition for fixed-sized states and dynamic partition for variable-sized states, as described in Algorithm 1. For fixed-sized states, we partition them evenly based on the number of workers. Each worker gets a partition of the same size to achieve workload balance. Since we know the size of the state, we can do it statically. For example, suppose that we have a 1000-byte bitmap and 10 workers in the feedback collection service. By static partition, each worker should maintain 100 bytes (e.g., the first worker maintains byte 0 to byte 99, and the second maintains byte 100 to 199). For variable-sized states, we partition them dynamically whenever new states are found, and distribute them evenly and uniformly at random to the workers. The intuition is that each partition maintained by the workers follows the same or similar data distribution as the whole service state. In this way, each worker can work independently but still get a good accumulated overall result with high probability. For example, if we find 100 new interesting test cases and distribute them randomly to 10 workers. Suppose that the global optimal strategy is to pick 10 test cases with the best performance scores for fuzzing, such strategy can be approximated by asking each worker to pick its best test case and combining them. μFUZZ performs static partition on the coverage bitmap in the feedback collection service and dynamic partition on the corpus in the corpus management service.

Result Accumulation. We need to accumulate outputs for some services to generate an overall result. For the feedback collection service, we simply aggregate the count for the "insteretingness" for the test cases: All the workers will check its partition of bitmap and output whether a test case triggers new bits. If all the workers say no, then the test case will be discarded. Otherwise, it will be sent to the corpus management service and added to the corpus. For the corpus management service, we forward the outputs without any changes since they already approximate a good result by simple aggregation as discussed above. In μFUZZ, the output caching queue performs result accumulation since all outputs go through it.

State Updates. In μFUZZ, the fuzzing state is updated by its maintaining serivce worker, and the state update requests come from other services. For example, a worker in the corpus management service will update a test case's performance score upon receiving feedbacks from the feedback collection service. However, the service state has been partitioned across the workers. When a service receives state update requests, we need to tell the input dispatcher which worker should process them. For statically partitioned states like the coverage bitmap, we use the partition boundary as the workers' unique identifier (ID). We then check the range of the bit offsets and figure out the corresponding worker. For dynamically partitioned states, we assign each worker with a unique number as the identifier, and all the states within the worker are tagged with the ID. Such IDs will be carried along the fuzzing loop and guide the input dispatcher. For example, every test case in the corpus management service will be tagged with its maintaining worker's ID. Therefore, the test cases sent to the mutation service will carry the ID and pass through to the feedback collection service. In this way, when the feedback collection service sends the performance score update requests to the corpus management service, the input dispatcher can check the IDs and dispatch the requests to the maintaining worker. The overall workflow in shown in Algorithm 1.

4.3.5 Zero-Copy Communication

As mentioned before, we break the fuzzing loop into different services, and each service consumes the outputs from other services and produces some for them. Considering the fast speed of fuzzing, the amount of passing data can be huge and thus potentially introduce high communication overhead. Therefore, we design a safe zero-copy mechanism to reduce communication overhead. Specifically, we utilize pointer passing with shared memory to pass only a constant size of data (*i.e.*, a pointer and a data size) regardless of the amount of generated outputs and unique ownership to enable safe access to data across services. For example, suppose the average size of the generated test cases is 1,000 bytes long, and the test case generation service generates 1,000 new test cases per second. Assuming we always copy the data from one service to another, the required data copying from the test case generation service to the execution service will be 1,000,000 bytes per second. The number will keep going up if we fuzz with more cores. However, if we can pass a pointer to the data, we pass only eight bytes on a 64-bit system.

Pointer Passing with Shared Memory. To avoid unnecessary memory allocation and data copying between the producer and consumer services, we create shared memory between the services and pass the pointers to the shared memory instead. After the shared memory is set up (*e.g.*, using `mmap`), the producer service writes its outputs directly to the shared memory. To "pass" the data to the consumer, the producer simply passes a pointer to the data and the size of the data to the output queue. Afterward, the consumer can fetch the pointer and the size to perform accurate data access. In this way, regardless of the output size, we only need to pass the small constant-size pointers and integers.

Unique Ownership for Safe Access. Using shared memory poses a safety risk. Since the shared memory is accessible from multiple services, if we allow the services to access the memory at the same time, race conditions could happen. To address this, we wrap the pointers in unique ownership to ensure safe memory access. This unique ownership ensures that only one service can access the shared memory at any given moment. This makes sense because the consumer should only access the output after the producer finishes generating it, and the producer does not need to access it afterward.

4.3.6 Implementation

Table 4.3: Line of codes of different components of μFUZZ, which sum up to 7,016 lines.

Module	Language	LOC
Concurrent Runtime	Rust	1,980
Corpus Management	Rust	759
Testcase Mutation	Rust	1,604
Fork-Server Execution	Rust	1,453
Feedback Collection	Rust	1,169
Others	Rust/Protobuf	2,569
Total	Rust/Protobuf	**9,534**

We implement μFUZZ in 7,016 lines of code. Table 5.1 shows the breakdown.

Concurrent Runtime. We use `Tokio` [196] as the concurrent runtime of μFUZZ. The runtime is responsible for efficient task scheduling. Each worker in the services of μFUZZ is run as a task in the runtime. The number of workers per service is empirically set to be the number of cores to achieve maximum parallelism. Users can adjust the number accroding to their use cases (*e.g.*, perform a short time dry run to try different values and pick the best one). We maintain a queue of unfinished tasks to execute. If the runtime is looking for a task to run, it pops one from the front of the queue. When a service receives inputs, its workers will get notified, and μFUZZ will try to put the workers in front of the queue, which allows them to be picked up for execution sooner. After a worker finishes its work, we put it at the back of the queue so that workers from other services have a chance to run. If all the inputs are processed or the service gets stuck, μFUZZ will move to the next service with inputs to be processed.

Corpus Management. The corpus management service maintains a corpus of test cases and their performance scores used in the test case selection algorithm. The performance score of a test case reflects how many interesting variants it has generated. When a test case is added to the corpus, we assign it an initial score and adjust it according to the feedback. For example, if a mutated variant of a test case triggers a new code path, the score of the test case is increased. For test case selection, we sort the test cases by scores and select them in descending order with random skipping.

Test Case Generation. μFUZZ uses `AFLplusplus`'s havoc mutation as its test case generation, which performs unstructured bit flip and byte modification on existing test cases. Since test case generation and execution are in separate services, sending the mutated test cases one by one to the execution service will result in too much service switching. Instead, we send the new variants in bulk to the execution service to reduce the overhead.

Execution. The execution service adopts the popular fork-server executor in its worker [12, 2]. Each worker maintains a fork-server. When a worker receives an input to execute, it feeds the input into the fork server and requests a process fork. The forked process executes the target binary with the test case as input and generates the code coverage and execution status (*e.g.*, crash, timeout).

Zero-Copy Communication. We run all services of μFUZZ in the same process to share the memory address space. To support multiprocess run, we can use `mmap` to create shared memory. In this way, zero-copy communication can be achieved by simple pointer passing. We use Rust's `std::sync::Arc`, a thread-safe reference-counting pointer, to wrap our data. We achieve unique ownership by ensuring that the reference counter of the pointer is always one so that there is at most only one owner for the underlying memory at any moment.

4.4 Evaluation

Our evaluation aims to answer the following questions.

- Can μFUZZ outperform state-of-the-art parallel fuzzers? (§4.4.2)
- What is the contributions of μFUZZ's components in the fuzzing performance improvement? (§4.4.3)
- Can μFUZZ find new bugs in real-world programs? (§4.4.4)

4.4.1 Evaluation Setup

Benchmark. We use the state-of-the-art benchmark `Magma` [197] to evaluate μFUZZ in bug detection capability and code coverage. We use the corpus from `Magma` for all the targets and run them through `AFLplusplus`'s test case minimizers to remove redundant ones beforehand. We compare μFUZZ with three state-of-the-art fuzzers: `AFLplusplus` [2], `AFLTeam` [10], and `AFLEdge` [11]. `AFLplusplus` is the most popular fork of `AFL` with various improvements and is actively maintained. `AFLTeam` and `AFLEdge` are the most recent and the open-source advanced parallel fuzzers, which focus on partitioning fuzzing tasks to different instances and are thus good comparison for μFUZZ's state partition. `AFLEdge` and `AFLTeam` work by integrating with existing single-instance fuzzers. Therefore, we run `AFLTeam` and `AFLEdge`

Table 4.4: Bug Detection Results in 24 Hours. We measure the bug detection capability in the number of identified bugs and their average *survival time*, which indicates the time a fuzzer needs to trigger the bugs. If the fuzzer cannot find the bug in 24 hours, we mark the survival time as ∞. The bug IDs are the unique identifiers for the inserted bugs in `Magma`. The time highlighted in green means the corresponding fuzzer is the fastest to find the corresponding bug. We sum up the number of the bugs found during any of the five runs in "Total Bugs Found". We exclude results for `openssl` and `PHP` because no fuzzers find any bugs in these targets in 24 hours.

Targets	Bug ID	μFUZZ	`AFLplusplus`	`AFLEdge`	`AFLTeam`	AFLpp-FS	μFUZZ-S	μFUZZ-P	μFUZZ-C
`Poppler`	PDF010	12h56m	19h18m	22h13m	05h31m	13h13m	∞	15h02m	17h40m
	PDF016	01m40s	16m40s	01m40s	16m40s	01m40s	03h21m	01m40s	09m10s
	PDF021	05h22m	∞	∞	∞	∞	∞	∞	13h51m
`sndfile`	SND017	01h24m	03h34m	02h51m	02h46m	16m45s	02h46m	03h54m	03h40m
	SND020	01h24m	04h15m	02h51m	02h46m	02h50m	02h46m	04h10m	02h54m
`libxml2`	XML002	12h57m	∞	∞	∞	∞	∞	14h31m	15h58m
	XML003	02h46m	∞	∞	∞	∞	∞	13h53m	06h10m
	XML009	16m40s	02h46m	02h46m	02h46m	02h46m	03h06m	02h46m	23m53s
	XML012	15h13m	∞	∞	∞	20h45m	20h07m	17h02m	19h49m
	XML017	01m40s	16m40s	01m40s	01m40s	01m40s	02h48m	06m40s	01h53m
`SQLite`	SQL002	02h46m	10h34m	07h28m	∞	03h16m	06h19m	17h23m	03h12m
	SQL018	02h46m	02h52m	05h14m	∞	03h40m	03h33m	05h16m	05h27m
`lua`	LUA004	02h47m	09h47m	19h11m	06h20m	16h40m	19h48m	02h48m	02h55m
`libpng`	PNG001	19h37m	∞	∞	∞	∞	∞	∞	∞
	PNG003	01m40s	03m25s	01m40s	02m20s	01m55s	17m20s	17m30s	01m40s
	PNG007	02h28m	02h50m	21h32m	23h05m	18h36m	06h03m	02h46m	03h03m
`libtiff`	TIF002	05h56m	∞	18h17m	∞	∞	∞	12h32m	∞
	TIF007	01m40s	16m40s	05h55m	01m40s	02h47m	05h34m	02h46m	02h51m
	TIF012	02h35m	09h26m	06h23m	02h46m	10h35m	14h29m	06h39m	02h55m
	TIF014	05h47m	03h11m	06h06m	04h22m	02h50m	12h06m	04h44m	03h01m
Total Bugs Found		**20**	**14**	**15**	**12**	**15**	**14**	**18**	**18**

on top of `AFLplusplus` for a fair comparison. For new bug detection, we evaluate μFUZZ with programs from `FuzzBench` [198].

Environment Setup. We perform our evaluation on five machines, each with an Ubuntu 18.04 operating system, an Intel Xeon CPU E5-2680 v3 processor (48 virtual cores) and 256 GB RAM. We instrument the tested programs to test edge coverage. For the code coverage and bug detection experiments, we run the fuzzers with 40 fuzzing instances on 40 cores for 24 hours. For μFUZZ, we run 40 workers for each of the four services but still use only totally 40 cores, the same number as the other fuzzers. We apply `Magma`'s survival analysis to convert the recorded bug triggering time to bug survival time, which is the expected time a bug remains undiscovered [197]. A smaller survival time indicates a fuzzer can find the bug in shorter time. We run each set of experiment in a new docker container to reduce environment interference, repeat the process five times and report the average results to reduce the random noise.

4.4.2 Comparison against existing fuzzers

We compare μFUZZ against three state-of-the-art fuzzers, including the *de facto* `AFLplusplus` and the two most recent parallel fuzzers, `AFLEdge` and `AFLTeam`. To understand whether increasing the synchronization frequency of existing fuzzers can be a solution, we compare with AFLpp-FS, which is `AFLplusplus` performing synchronization every 30 seconds instead of 30 minutes. We use 30 seconds because it works as well as shorter intervals in code coverage, but with less overhead according to our experiments. The evaluated metrics include bug detection capability (the number of triggered bugs and their survival time) and edge coverage.

Bug Detection. As shown in Table 4.4, μFUZZ finds 20 bugs in 24 hours, while `AFLplusplus`, `AFLEdge`, `AFLTeam`, and AFLpp-FS find only 14, 15, 12, and 15 bugs, respectively. All the 17 bugs found by other fuzzers are also covered by μFUZZ, and μFUZZ found 12 of them using the shortest time. Three of the bugs (`PDF021`, `XML002`, `XML003`) are only found by μFUZZ. Additionally, AFLpp-FS finds all the 14 bugs found by `AFLplusplus` and one more, but it still finds five less than μFUZZ. This shows that more frequent synchronization can help improve the bug detection capability of `AFLplusplus` because the fuzzing instances can catch up with the latest progress earlier, but the improvement is limited.

Code Coverage. As shown in Fig. 4.5, on average μFUZZ identifies 24%, 41%, 80%, and 31% more new edges than `AFLplusplus`, `AFLEdge`, `AFLTeam`, and AFLpp-FS respectively. If the programs have a larger program state space to explore, μFUZZ can achieve higher code coverage improvement (*e.g.*, 40% more in `Poppler` and 37% more in `PHP` than `AFLplusplus`). Otherwise, the improvement of μFUZZ is smaller (8% more in `sndfile` than `AFLplusplus` and 2% less than AFLpp-FS). μFUZZ uses almost the same fuzzing strategies as `AFLplusplus` but has a higher code coverage. This is because `AFLplusplus` does not perform state partition and relies on corpus synchronization at long intervals (*i.e.*, 30 min). Between two synchronization, the fuzzing instances are not aware of the progress made by others. Interestingly, the coverage of AFLpp-FS is higher than `AFLplusplus` at the beginning (*i.e.*, in the first few hours) but lower in the end. We investigate the results and find the following reasons. When the fuzzing starts, the corpus is small and the program space is not well explored. An instance quickly finds a bunch of interesting test cases, but cannot explore all of them timely. Under this situation, more frequent synchronization allows other instances to catch up with the progress and help explore the interesting test cases. And a small corpus can be synchronized with low overhead. However, as the fuzzing goes, new code becomes harder to trigger and there are not as frequent progress updates as in the beginning, but AFLpp-FS still synchronizes the corpus frequently. Since the corpus has grown bigger, synchronization becomes more expensive, resulting in a slower increase in code coverage.

On the other hand, `AFLEdge` and `AFLTeam` perform state partition but still have worse performance than μFUZZ. We investigate their algorithms and execution status and find that they both follow a period "gather and partition" approach: Once per hour, they aggregate the corpus of all their instances and then perform partitioning based on heavy analysis. However, since we are running the experiment with 40 instances and the size of the aggregated corpus is large, the analysis takes a long time to finish. For example, we find that it takes `AFLEdge` more than three hours to finish one round of partitioning on `PHP`. By the time it finishes, the fuzzing has made three more hours' progress. The partition results might be obsolete and not necessarily beneficial to the latest fuzzing state. In comparison, μFUZZ updates its state continously and timely, and thus avoids using obsolete states. Although μFUZZ's partition is not as comphrehensive as that of `AFLTeam` and `AFLEdge`, the timely state updates allow μFUZZ to make fuzzing decisions that fit the current state.

Overall, μFUZZ outperforms the four compared fuzzers in both bug detection and code coverage under parallel fuzzing. The fuzzing effectiveness of μFUZZ comes from both its architecture and state partition.

4.4.3 Contribution of Components

To better understand the contributions of μFUZZ's components, we compare μFUZZ with μFUZZ-S (μFUZZ without state partition), μFUZZ-C (μFUZZ without concurrency), and μFUZZ-P (μFUZZ without zero copy communication). More specifically, every worker of the corpus management service and the feedback collection service in μFUZZ-S maintains a copy of the global service state. Whenever there are state updates, μFUZZ-S will propagate the updates to all its workers. μFUZZ-C handles all the requests and generates output synchronously. μFUZZ-P passes full copies of data for all service communication instead of pointers.

Bug Detection. As shown in Table 4.4, μFUZZ successfully identifies 20 bugs in the targets, while μFUZZ-S, μFUZZ-C and μFUZZ-P finds only 14, 18 and 18 respectively. μFUZZ finds all the bugs that are found by the other three fuzzers with shorter time, and both μFUZZ-C and μFUZZ-P cover the bugs found by μFUZZ-S. Therefore, all the three different aspects of μFUZZ help improve the bug detection capability, and the state partition contributes the most.

Code Coverage. As shown in Fig.4.5, μFUZZ finds 23%, 7%, 6% more edge than μFUZZ-S, μFUZZ-C and μFUZZ-P respectively on average. We check μFUZZ-S's execution status and find that workers in the corpus management service of μFUZZ-S tend to select duplicated test cases for mutation. This is because without state partition the workers have the exact same state as each other and use the same scheduling algorithm. Such duplication can slow down fuzzers' exploration. Both μFUZZ-C and μFUZZ-P have a smaller fuzzing speed than μFUZZ (on different targets, 4% to 8% less), but for different reasons. For μFUZZ-C, it happens occasionally that some workers in a service have finished processing the requests and others have not. Even though we have available cores to run workers in other services, μFUZZ-C cannot do so without concurrency. For μFUZZ-P, it has to spend more computation power on copying data than μFUZZ.

Overall, μFUZZ outperforms μFUZZ-S, μFUZZ-C and μFUZZ-P in bug detection and

Table 4.5: **New bugs detecte by μFUZZ.**

Target	Type	Status	Ref
PHP (8.3.0-dev)	Use-After-Free	Fixed	https://github.com/php/php-src/issues/10582
	Use-After-Free	Fixed	https://github.com/php/php-src/issues/10169
	Use-After-Free	Acknowledged	https://github.com/php/php-src/issues/10581
	Use-After-Free	Acknowledged	https://github.com/php/php-src/issues/10571
	Null Pointer Deref	Fixed	https://github.com/php/php-src/issues/9543
	Memory Leak	Fixed	https://github.com/php/php-src/issues/10582
	Assertion Failure	Fixed	https://github.com/php/php-src/issues/10570
	Assertion Failure	Fixed	https://github.com/php/php-src/issues/10085
lua(5.4.4)	Assertion Failure	Fixed	http://lua-users.org/lists/lua-l/2023-02/msg00020.html
freetype(2.12.1)	Segmentation Fault	Fixed	https://gitlab.freedesktop.org/freetype/freetype/-/issues/1197
FFmpeg (9903ba)	Assertion Failure	Fixed	https://trac.ffmpeg.org/ticket/9908

code coverage, meaning that state partition, concurrency and fast communication all contribute to μFUZZ's improvement. State partition allows all the workers to work independently and still achieves a great aggregated result. The concurrent design allows μFUZZ to run differnet services concurrently without blocking. And the fast communication allows μFUZZ to reduce the overhead of data copying.

4.4.4 Identified New Bugs

μFUZZ find 11 new bugs in four well-tested programs from `FuzzBench`, showing that μFUZZ is applicable in real-world fuzzing. We do not use `Magma` for new bug detection because `Magma` uses the fixed old version of the programs to insert bugs stably. The details for the bugs are shown in Table 4.5. The identified bugs include four logical errors (*i.e.*, assertion failure), six memory corruption errors(*e.g.*, heap use-after-free), and a memory leak error, of which nine have been fixed and the remaining two acknowledged by the developers.

4.5 Discussion

In this section, we present some limitations of the current implementation of μFUZZ and discuss their possible solutions.

4.5.1 Distributed Fuzzing

Currently, μFUZZ is implemented as a multithreaded program running in a single machine. μFUZZ can be extended to support distributed fuzzing in two ways. One way is to run one μFUZZ on each machine and perform state synchronization by connecting services with remote procedure calls (RPC). For example, we can run μFUZZ on different machines and connect their corpus management services to synchronize the corpus, which is the state-of-the-art approach. This will transfer the same amount of data across machines as existing approaches. However, existing serial fuzzers will pause to perform the slow network I/O, while μFUZZ allows other services to progress concurrently. If one service pauses, the other services can still run and make individual progress. Another way is to run different services on different machines and communicate over the network. This will greatly increase the amount of network data because the zero copy communication is not applicable. Although the data copy is inevitable, we can mitigate the waiting for network I/O. We can warm up the service by input caching: before a service starts, it fetches enough inputs from other services into its cache. Afterward, each service keeps fetching more inputs from other services into the cache and running the workers to consume the inputs concurrently. In this way, the service can keep running without waiting for inputs. Since each service will fetch inputs at demand, one straightforward way to achieve dynanmic load balancing is always picking the producer serivice with the most cached outputs. Suppose we have two test case generation services and two execution services and each of them runs on a different server with the same computation power. Each of the execution service should connect to both test case generation services. When the execution service needs to fetch more test cases, it picks the test case generation service with the more cached test cases to fetch from. We should make sure the amount of each fetch is not too large, which depends on the network capability, to avoid that one service fetches all the data and the other service is starved. Therefore, to support distributed fuzzing, μFUZZ might suffer from the same overahead as or more overhead than existing fuzzers due to data copying, but it can mitigate the blocking of network I/O by its concurrency.

4.5.2 Support More Mutation Strategy

Currently, μFUZZ implements some basic state-of-the-art fuzzing strategies such as `AFLplusplus`'s havoc mutation and edge coverage guidance. We can integrate advanced strategies into μFUZZ to futher improve μFUZZ's applicability. Stateless fuzzing strategies can be easily integrated thanks to μFUZZ's modularized design. For example, we can use advanced mutation strategies such as AST mutation [65, 6] as the workers in the mutation service. We can also use persistent executors [2, 39] or binary-only executors [202] in the execution service. For stateful ones, we should apply dynamtic or static partitioning to avoid as much synchronization as possible. To support CmpLog [42], μFUZZ can log the compared values in the execution service and pass the log to the feedback collection service. Then it categorizes the log and passes it to the corpus management service, which maintains the log as metadata of the test cases and passes them to guide the mutation service. Since the test cases are partitioned in the corpus management service, the compared values log is also partitioned and maintained independently. To support MOPT [189] mutators in μFUZZ, we partition different mutators into different workers in the mutation service. The feedback service can send back the information about the mutators' performance (e.g., the number of interesting inputs found by each mutator). Then mutation worker can adjust its local distribution and perform period synchronization with other workers to update the global optimal distribution. We see that some synchronization might be necessary. In this situation, the service might be blocked by locking, but the concurrency still allows other services to make progress. Therefore, one best-effort strategy is to adopt μFUZZ's architecture as much as possible, and fall back to state synchronization when necessary. We plan to integrate μFUZZ with `LibAFL` [203], an open-sourced fuzzing development kit that implements many advanced fuzzing strategies, including all the discussed ones, as self-contained reusable modules. For the stateless strategies, we can directly reuse modules in the workers as both μFUZZ and `LibAFL` are written in Rust. For the stateful ones, we need to first separate the data (*i.e.*, the fuzzing state) and operations in the module and then design the partitioning strategy, which requires nonnegligible engineering efforts. Luckily, μFUZZ takes care of data interaction and communication between the services, so the users can treat the data as local and focus on the partitioning strategy.

4.5.3 Support Collaborative Fuzzing

Currently μFUZZ has not implemented collaborative fuzzing [125, 133], which combines all kinds of different fuzzers to get a higher overall fuzzing performance. We can support collaborative fuzzing with μFUZZ by increasing the vairety of the workers (*i.e.*, using workers with different fuzzing strategies). For example, we can use both grammar-based mutation workers and bitflip mutation workers in the mutation service, and we can use execution workers with different type of instrumentation in the execution service. However, different fuzzing strategies might have different fuzzing states for the same functionality, which means that we should distinguish workers of different types of fuzzing states and dispatch the inputs accordingly. For example, grammar fuzzers might maintain a corpus in the form of abstract syntax trees (AST) instead of a binary stream. Suppose we use both grammar-based mutators and bit-level mutators in the mutation service. In that case, the input dispatcher of the test case generation service should dispatch inputs of AST test cases to a worker of grammar-based mutation instead of a worker of bitflip mutation. μFUZZ can support this by tagging both workers and data. Each worker has a tag to indicate the type of the inputs it can consume. Every result generated by a worker will be tagged to show which workers can consume it. With that, the input dispatcher can dispatch inputs to matching the inputs and workers. In this way, μFUZZ can combine fuzzing strategies with different types of fuzzing states.

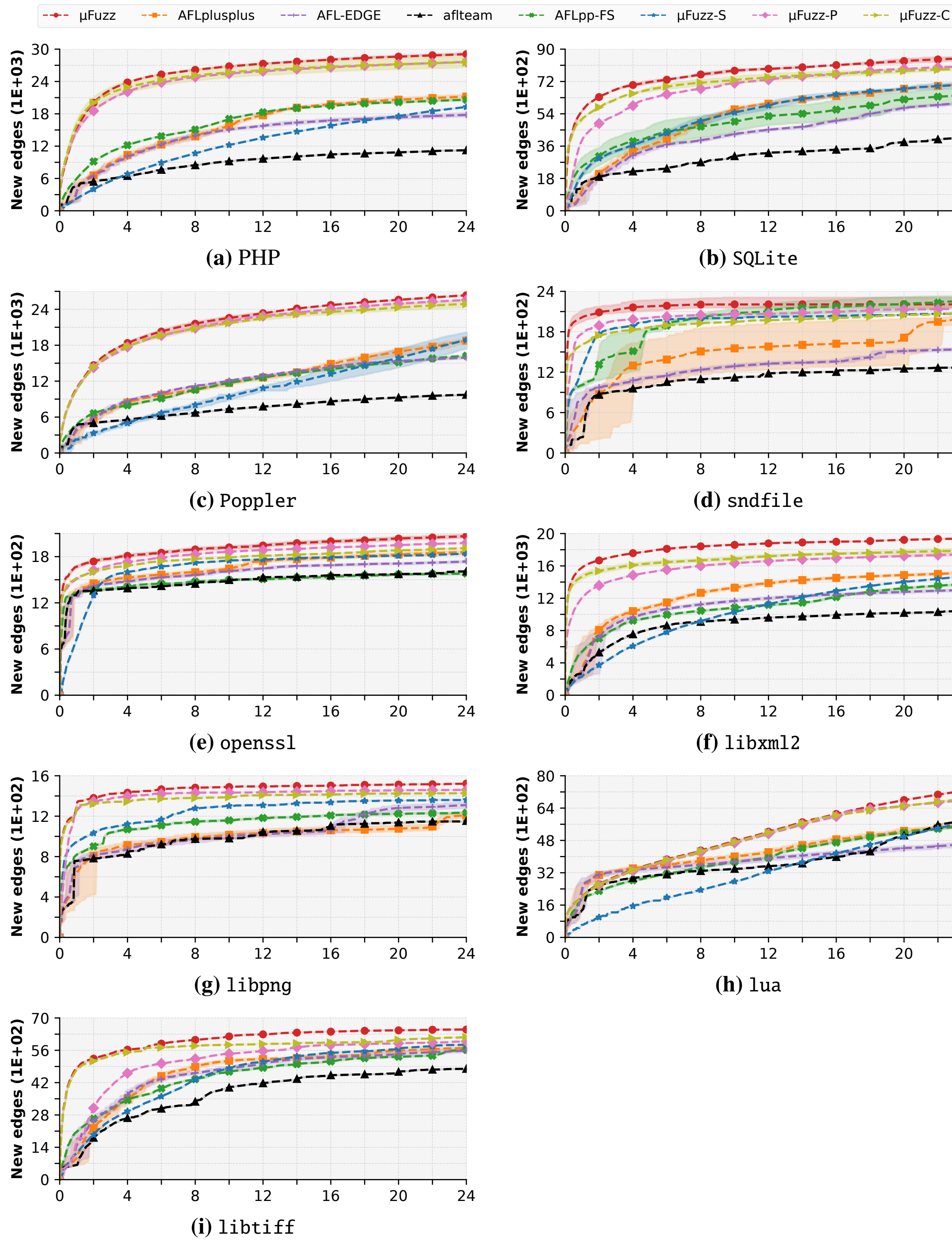


Fig. 4.5: Edge coverage found by evaluated fuzzers with 40 cores in 24h. AFLpp-FS is `AFLplusplus` with 30-second synchronization interval. μFUZZ-S is μFUZZ without state partitioning. μFUZZ-P is μFUZZ without zero copy communication. μFUZZ-C is μFUZZ without concurrency.

CHAPTER 5
PROPGUARD

In this chapter, we present PROPGUARD, a fuzzing framework that enhances the bug modeling capability of fuzzing tools by enabling users to define diverse bug patterns with a novel specification language. PROPGUARD then leverages these patterns to perform targeted bug searching, effectively identifying a wide array of vulnerabilities beyond mere crashes.

5.1 Introduction

Fuzzing has shown its effectiveness in finding software bugs and vulnerabilities [2, 4, 12, 47]. The most common types of bugs that fuzzing finds are crashing bugs, which are software defects that might cause the program to crash, halt, or terminate unexpectedly during its execution. Examples include out of memory, memory corruption and invalid memory access (*e.g.*, null pointer dereference, buffer overflow, use after free). These bugs are easily noticeable because they disrupt the normal operation of the application. The crashing symptoms are general across different programs and can be easily detected by the fuzzing engine without the need for explicit modeling.

Non-crashing bugs, on the other hand, do not cause the program to terminate abruptly but can still affect its functionality or performance. These bugs may lead to incorrect results, data corruption, or security vulnerabilities without necessarily stopping the program. Example includes bugs that cause the program to return incorrect data or leak sensitive information. These non-crashing bugs are as harmful as crashing bugs but even more dangerous in the security context because they may remain undetected for a longer time as they do not expose easily observable behaviors. To detect such bugs with fuzzing, researchers have proposed various approaches to model the bug patterns by developing oracles [204, 205], which can notify the fuzzers when such bugs are triggered. One widely used approach is to write assertions in the code to assert that some properties always hold during the program execution. An assertion failure is considered as a bug. Assertions are simple and easy to write, but they cannot describe complex properties such as those with inter-procedural flow dependencies. For example, we cannot detect whether a buffer is already freed in an assertion because the free might happen in another function. To detect more complex bugs, researchers seek to develop more dedicated oracles [204, 149, 143, 144]. For example, `System Sanitizer` detects command injection bugs by monitoring the data flow to sensitive functions. Differential testing oracles [143, 144] compares the execution of two different versions of the same functionality to detect bugs such as misoptimization. These oracles are powerful in detecting a specific type of bugs. An alternative approach is to use static analysis tools. They can analyze the source code or the binary of a program without actually executing it dynamically and thus can explore multiple possible code paths. To detect various types of bugs, state-of-the-art static analysis tools like `CodeQL` [206] provide expressive and high level ways to describe the bug patterns. Users can write a few lines of specification to describe the complex bug patterns and then search for such patterns statically.

However, these approaches have limitations. As mentioned, assertions, though simple to write, are not expressive enough to describe complex properties. Developing specific oracles is effective in detecting a specific type of bugs, but it is time-consuming and cannot be generalized to other types of bugs. For example, `Address Sanitizer` can detect memory corruption bugs but cannot detect any command injection bugs. Considering the large number of bug types that require modeling, it is impractical to develop dedicated oracles for each of them. Static analysis tools like `CodeQL` [206] allow users to describe complex properties easily in an expressive specification language, and thus greatly reduces the effort of developing oracles. However, as they match patterns statically, they may suffer from high false positive rates, and they cannot produce concrete proof of concepts. Moreover, static analysis tools are known to not scale well with large programs.

To fully understand these limitations, it is essential to consider two fundamental steps in bug discovery: **bug modeling** and **bug searching**. Bug modeling defines what a bug is (*i.e.*, the pattern of the bug). Bug searching tries to search for program states that match the bug pattern. Through these definitions, it becomes apparent that existing fuzzing approaches mainly focus on the bug searching of crashing bugs but not the bug modeling. Furthermore, it requires a huge development effort to model specific types of non-crashing bugs by developing dedicated fuzzing oracles. In contrast, static analysis tools present excellent capabilities in bug modeling. However, their static approach to bug searching is plagued by high false positive rates and faces scalability challenges in large programs. We need a new approach that combines the expressiveness of static analysis in bug modeling and the scalability of fuzzing in bug searching.

In this chapter, we propose PROPGUARD, a fuzzing framework that allows users to find non-crashing vulnerabilities effectively. The idea of PROPGUARD is simple: we integrate bug modeling capability of the static tool but still keep fuzzing as the bug searching engine. We achieve our goal in three steps: **Bug specification**, where we provide an expressive specification language for users to describe the complex bug patterns (*e.g.*, those involve multiple steps in different functions with dedicated control or data flow dependencies) easily; **Bug revelation**, where we concert the bug specification into a fuzzing oracle that notifies the fuzzer when the bug is triggered without relying on program crashes; and **Bug-oriented searching**, which directs the fuzzer to focus on searching for the specified bug patterns instead of random exploration. Bugs without explicit source code patterns (*e.g.*, hardware defects or mis-compilation, where the bugs are introduced by the compilers instead of the program itself) are out of our scope.

We implement PROPGUARD with around 7,016 lines of C++ and Python code, which consists of a frontend that serves as the language specification compiler and a backend that performs the code instrumentation. We integrate `AFLplusplus` as the underlying fuzzing engine in PROPGUARD.

To understand the effectiveness of PROPGUARD, we evaluate it on eight different types of bugs from top Common Weakness Enumeration (CWE) based on their popularity, severity and complexity, including 5 non-crashing types (Command injection, path traversal, information exposure, server-side request forgery) and 3 crashing types (use after free, out-of-bound memory access, null pointer dereference). The result shows that PropGuard can detect all eight of them while the compared dynamic tools, including assertions, `Address Sanitizer`, `System Sanitizer` and `FuzzFactory`, can detect at most four because they lack of the corresponding bug modeling. We further evaluate PropGuard on ten public non-crashing vulnerabilities from popular open-source projects and show that PropGuard successfully finds all of them. Last but not least, PropGuard finds two new non-crashing bugs in a popular gltf parsing library.

5.2 Problem

In this section, we first discuss how the existing dynamic (§5.2.1) and static (§5.2.2) approaches implement bug modeling and bug searching and their limitations in detecting non-crashing vulnerabilities. Finally, we explore how we enhance fuzzing by integrating strengths from static analysis tools and propose our approach (§5.2.3).

5.2.1 Limitations in Existing Dynamic Approaches

Popular dynamic approaches in bug detection mainly include unit testing and fuzzing [12, 145, 47, 2, 205, 207, 208]. Unit testing [207, 208] specify a set of inputs and expected outputs and any deviation from the expected output is considered as a bug. As it only checks a few specific cases, it has limited bug modeling and bug searching.

On the other hand, fuzzing, an automated testing technique that keeps trying various inputs to detect program defects, mainly focuses on bug searching [12, 2]. State-of-the-art fuzzing engines usually utilize code coverage guidance or input structure to better explore the program state space and detect more bugs. Without explicit modeling, fuzzing uses program crashing as the oracle, because program crashing has easily observable external symptoms (such as killing the process, generating the coredump) and is considered as a bug in almost all programs. However, there are many other bugs without such external symptoms [209, 210]. For example, a logical error in the program may cause the program to return incorrect data, which is not understandable by the fuzzing engine without explicit modeling. And the program can keep running without crashing. For example, state-of-the-art fuzzers like `AFL` [12, 2] do not have bug modeling capability and thus can only find crashing bugs.

To detect such bugs during fuzzing, existing approaches develop oracles to model the bug patterns [145, 205, 143, 149]. A common approach is to put assertions in the program to check the invariants of the program. However, assertions are easy to write but not expressive enough to model complex bug patterns. They only provide simple local property checking because they can only access intra-procedural information but not inter-procedural control flow related or data flow related information. For example, if we want to check whether the value of a variable comes from user input or we want to check whether the current function is called from a specific function, assertions fail to do so. To model complex bug patterns, researchers have developed dedicated oracles for specific types of bugs. A popular and widely used oracle is address sanitizer (ASAN) [145], which detect invalid memory access such as out-of-bound read/write and use after free, etc. It keeps track of the memory allocation and deallocation. Instead of relying on program crashing, which might not happen after memory corruption happens, `Address Sanitizer` will report a bug and exit the program as soon as an invalid memory access is detected. Similar oracles include thread sanitizer (TSAN), which detects data races and deadlocks and memory sanitizer (MSAN), which detects uninitialized memory access [145]. These oracles greatly enrich the types of program defects that can be detected by fuzzing.

Although dedicated oracles like `Address Sanitizer` are effective in detecting specific types of bugs, they require huge engineering efforts to develop and cannot be generalized to other types of bugs. The potential number of different bug types is unlimited, many of which are non-crashing such as information disclosure and command injection and requires bug modeling. To make things worse, the bug types might be program specific, which means the oracle cannot even be reused to other programs [206]. Therefore, it is impractical to develop dedicated oracles for each type of bug we want to find. Additionally, even with these oracles, the existing approaches do not utilize them to guide the fuzzing in a directed way. For example, if we use `AFL` and thread sanitizer together to find concurrency bugs, the fuzzing process is still undirected, trying to cover all the code area. But actually, we should focus our testing on the code that is related to concurrency instead of randomly exploring unrelated code area without any concurrency.

5.2.2 Limitations in Existing Static Approaches

Alternatively, static analysis tools [**wang2009intscope**, 206, 211] put more effort into bug modeling. To model a bug, static analysis usually equips with various information, such as control flow graph, data flow graph and call graph, and thus is more expressive in describing the bug pattern. Moreover, to further improve the bug modeling capability, recent tools like `CodeQL` [206] and Semgrep [211] provide high-level and expressive specification languages, with which users describe the bug patterns easily in a few lines of specification. As they analyze the source code or the binary of a program without actually executing it, they can explore multiple possible code paths. Therefore, they put less effort into bug searching by just performing a thorough search.

However, static bug searching has several limitations. First, they may suffer from high false positive rates. For example, as they search for the bug pattern statically across functions, they need to perform inter-procedural analysis, which is known to be difficult to be precise or accurate (*e.g.*, the potential targets of indirect calls) [212]. Second, as the program is not actually run, they cannot or require heavy symbolic solving [213] to produce concrete proof of concepts for the alerts they find, making it difficult for developers to validate the findings. Third, they might suffer from scalability issues such as path explosion. Many static tools search for the bug pattern with path-sensitive analysis and multiple path exploration. In complex applications, the number of possible paths grows exponentially with the code size, drastically increasing the analysis overhead. For example, `CodeQL` compiles the program into a semantic database and then search the pattern as database query. The complexity of searching depends on the number of searching conditions, the interaction between the searching conditions, and the complexity of the program. The more complex the query is, the more join operations are needed, which can lead to a combinatorial explosion.

Therefore, the question is whether we can combine the advantages of both fuzzing and static analysis so that we can have both expressive bug modeling and effective bug searching.

5.2.3 Our Approach

The goal of this paper is to design a framework that allows fuzzing to detect non-crashing vulnerabilities easily and effectively. We achieve our goal by combining the expressiveness of static bug modeling and the effectiveness of dynamic bug searching in three steps: **bug specification** (make the bug describable), **bug revelation** (make the bug detectable), and **bug-oriented searching** (focus searching on the bug). More specifically, we design a specification language that allows users to describe the bugs they want to find in a high-level and expressive way. Next, our framework will then transform the property specification into a fuzzing searchable oracle and instrument the oracle into the program. Then, PROPGUARD tailors the fuzzing engine based on the bug specification to explore the program for the bug in a directed and targeted way.

Bug Specification Language. The specification language allows users to describe the bugs they want to find. It is expressive that it supports modeling complex bug patterns with intra-prodedural and inter-prodedural control flow and data flow dependencies. Yet it has a simple format and is easy to use so that users can specify the bug patterns in just a few lines of specification (See examples in Fig. 5.2). It provides a simple set of primitive operations for pattern matching and connectors for combining the primitive operations into complex patterns. A bug specification describes what code we should look for and what requirements such as data flow or control flow dependencies that the code should satisfy to be considered as a bug. The specification language greatly reduces the development effort of writing an oracle, especially for complex and program specific bugs.

Fuzzing Oracle Instrumentation. With the bug specification, we want to create an oracle from the specification that can alert the fuzzer when the program execution matches the pattern. The insight of the fuzzing oracle construction is that the finite-step program execution can be modeled as the transition of a state machine. As the bug pattern is actually matching a subsequence of the program execution, we can model the bug pattern as a (sub)state machine. For example, given a bug pattern that looks for a double free bug and a program execution trace, the state machine of the bug pattern involves only the code that is related to the memory allocation and deallocation. The final states of the state machine indicate either the bug is triggered or the program execution does not match the bug pattern. Therefore, we can construct a state machine that models the pattern described in the specification and instrument program to track the transition of the state machine. If the program transitions to a final state with a bug triggered, PROPGUARD will alert the fuzzer.

Directed Fuzzing. Given the bug specification and the source of the program that we want to fuzz, we can figure out which parts of the program might be related to the bug (*e.g.*, which paths might trigger the bugs). We can then shrink the program space to only include the related parts. However, even this reduced program scope can be too large for methods like symbolic execution to explore. For instance, focusing solely on double free bugs, there are still numerous paths that could lead to these issues. This is where directed fuzzing is useful, which focuses on searching the shrank program space instead of randomly exploring unrelated code areas. Therefore, we analyze the source program and utilize the results to tailor the fuzzing engine to perform directed fuzzing (*i.e.*, bug-oriented searching). More specifically, PROPGUARD improves the feedback guidance by debloating unrelated code coverage, input scheduling by collecting inputs that trigger state transition, and input

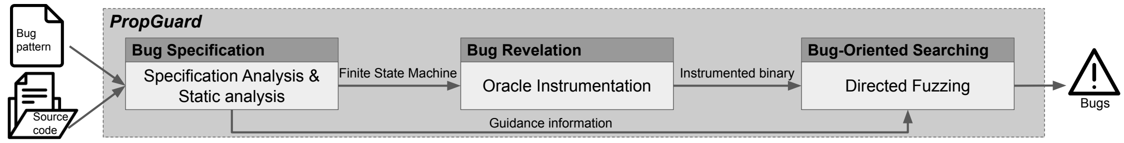


Fig.5.1: The overview of PROPGUARD. Given the specification of the bug pattern and the source code of the program under test, PROPGUARD first analyzes the specification and the source code to generate a finite state machine (FSM) that models the bug pattern and the guidance information for directed fuzzing. The FSM is then instrumented into the program as a fuzzing oracle, which tracks the state transition and report the bug if a buggy state is reached. Besides, the guidance information is used to tailor the fuzzing engine to focus on searching for the specified bug patterns instead of random exploration. Afterward, the fuzzing engine can fuzz the program for the bug.

generation by generating a dictionary of potential constants that can be used to trigger the state transition.

In short, PROPGUARD allows users to create an oracle for their bug pattern easily and search for the bug pattern effectively.

5.3 Design

Fig.5.1 shows the overview of PROPGUARD. Users first need to describe the bug pattern that they want to detect in the specification language that PROPGUARD provides (§5.3.1). This language is expressive and high-level so that users can easily express the pattern in a few lines of specification. With the bug specification and the source code of the program under test, PROPGUARD constructs a finite state machine and transforms it into a fuzzing oracle by instrumenting the program under test (§5.3.2). The fuzzing oracle decides whether the bug pattern is matched by dynamically tracking the state transition (§5.3.3). In the meantime, PROPGUARD extracts the information from the bug specification and tailors the fuzzing engine to guide the fuzzing towards the bug triggering direction (§5.3.4). Afterwards, users can perform fuzzing on the instrumented program. All users need to do is to write a few lines of bug specification, and the rest is done by PROPGUARD automatically, which greatly simplifies the process of developing effective fuzzing oracles.

```
1 vars: int $SIZE, char* $BUFFER,
2        char* $URL, bool $B1, $B2;
3
4 seq {
5  match: $SIZE = readn($BUFFER, *);
6  operation:Taint($BUFFER, $SIZE);
7  assert: true
8
9  match: SendRequest($URL)
10 operation:
11  $B1 = RegexFind($URL, "127.0.0.1")
12  $B2 = IsTainted($URL) && $B1
13 assert: $B2 == false
14 }
```

(a) Example bug specification for SSRF.

```
vars: int $RESOURCE

seq {
  match: $RESOURCE = Create(*)
  assert: true // optional

  match: Release($RESOURCE)

  not {
    match: $RESOURCE = Create(*)
  }

  match: Release($RESOURCE)
  assert: false
}
```

(b) Example bug specification for generic double free.

Fig. 5.2: Example bug specifications for server-side request forgery (SSRF) and generic double free in the specification language of PropGuard. The bug specification consists of basic units, each of which defines a code pattern and the requirements on the values. Basic units can be chained together using connectors (*e.g.*, the `seq` connector) to express the control flow dependencies or using the shared variables (*e.g.*, the variable `$RESOURCE` in Fig. 5.2(b)) to express data flow dependencies.

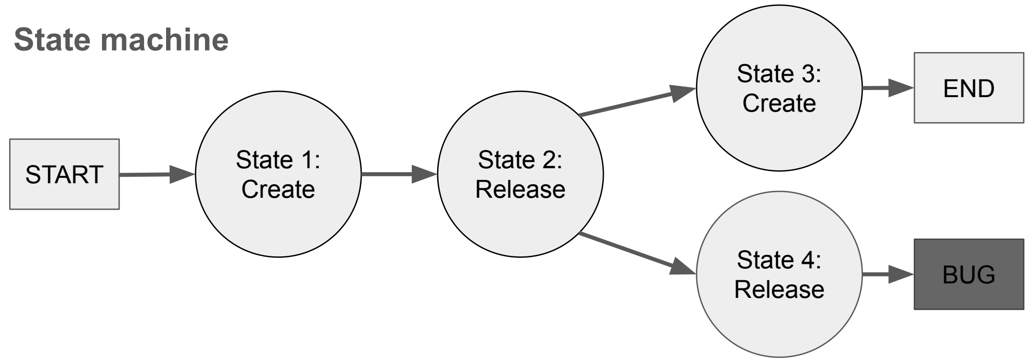
5.3.1 Specification Language

To describe the bug pattern that we want to find during fuzzing easily and accurately, we design a specification language for bug modeling. The bug patterns in PropGuard consist of two parts: the syntax, which includes a set of code patterns, which can match code locations like regular expression, and the semantics, which include the value requirements and the relationship such as control flow or data flow dependencies between the code patterns. For example, a `use-after-free` pattern involves two code patterns: the free pattern and the use pattern. The free pattern can match all the code locations that buffer pointers, and the use pattern can match where buffer pointers are used. The free must happen before the use, which is the control flow dependency. Besides, the use must use the same buffer pointer as the one in the free , which is the data flow dependency. That is, the code `free(ptr1); use(ptr1);` is a `use-after-free` bug while the code `free(ptr1); use(ptr2);` is not. Sometimes we might want extra value requirements on the code patterns. See the bug specification in Fig. 5.2(a) as another example. At line 9, we want to match the `SendRequest` function calls. However, we want to match the calls that have the first argument as a string that comes from the user input. Besides, we further want to only match the string containing `127.0.0.1`, which is the value requirement, meaning that the request is sent to the local host and also controlled by the user. In the specification language of PROPGUARD, we specify the syntax and value requirements of code patterns, which define **basic units**. These basic units can spread across different functions. And we further describe the semantic flow of the basic units with flow dependencies using connectors or shared variables. Together, the specification language can describe complex bug patterns.

Basic Unit. In PROPGUARD, a basic unit defines what code and what values to look for (**match**, *e.g.*, what function calls and which arguments of the function calls), what results we want to get from the values (**operation**, *e.g.*, the value of the argument should be a printable string), and what requirements the calculated results should satisfy (**assert**). The **match** is a regular expression that matches code locations (*e.g.*, a function call to `malloc`) and the around variables. For example, at line 5 of Fig. 5.2(a), we match all the calls to `readn`, which reads from user input into a buffer. We are also interested in the buffer and the number of read bytes so that we match them in two variables `$BUFFER` and `$SIZE`, which are defined at line 1. These variables are defined by the specification language and can be used to reference the captured values in the basic units. The **operation** of a basic unit is a list of operations that we want to perform on the matched variables. For example, we want to mark user inputs as "tainted" and thus at line 6 of Fig. 5.2(a), we use the primitive operation `Taint` provided by the specification language. As `$BUFFER` contains user input, we mark it as tainted. At line 11 of Fig. 5.2(a), we want to check whether the `$URL` is trying to send a request to the local host, so we use another primitive `RegexFind` to do string matching. The results of these operations can also be stored in the variables. The **assert** of a basic unit is a predicate on the results of the operation. If the predicate is violated, we consider that a bug has been found. For example, at the line 13 of Fig. 5.2(a), we assert that the `$URL` should not be controlled by the user and at the same time try to make an internal request. Otherwise, we consider it as a SSRF bug. Both the **operation** and **assert** are optional as shown in Fig. 5.2(b).

Flow Dependencies. As we see above, one basic unit models a single code pattern. However, non-trivial bugs usually involve multiple code locations, each corresponding to a different code pattern. And these code locations might spread across the program, and thus have control flow and data flow dependencies or requirements. For example, the double free bug specification in Fig. 5.2(b) defines four basic units. A second `$Release` (line 13) should happen after the first one (line 7), and in between there should be no `Create` (line 9-11). These are the control flow dependencies. Additionally, all these basic units access the same variable `$RESOURCE`, meaning that the value of the `$RESOURCE` flows across these basic units. These are the data flow dependencies. Therefore, for control flow requirements, we define four connectors for the basic units: `sequential`, `logical AND`, `logical OR` and `logical NOT`. The `sequential` connector requires the basic units to be executed in the order specified by the relation connector. The `logical AND` connector requires all the basic units to be executed in any order. The `logical OR` connector requires at least one of the basic units to be executed. The `logical NOT` connector requires the basic unit to not be executed. Data flow requirements are provided at two levels: value-based and tainting-based. In value-based data flow, if two basic units are referring to the same variable, it means that there is a data flow dependency between them and the referred values should be the same during dynamic execution to match the pattern. For example in Fig. 5.2(b), by referring to the same variable `$RESOURCE` at lines 7 and 13, we require that the value of `$RESOURCE` should be the same at these two locations. That meaning said, `Release(pointer1)` and `Release(pointer1)` have a data flow dependency, while `Release(pointer1)` and `Release(pointer2)` do not. On the other hand, tainting-based data flow dependency requires users to explicitly use the data flow primitives such as `Taint` to model. For example, in Fig. 5.2(a), the variables `$BUFFER` (line 5) and `$URL` (line 11) are different, so we do not require them to be the same for the pattern to match. However, the operation of the first basic unit (line 6) explicitly marks the `$BUFFER` as tainted, and the operation of the second basic unit (line 11) checks whether the

$URL is tainted, so we still have a data flow dependency between them.



(a) The state machine constructed from the bug specification in Fig. 5.2(b). Each basic unit corresponds to a state in the state machine. The constraint about `$RESOURCE` (*e.g.*, data dependency) is not demonstrated in the figure.

```
// Example program
1. Pointer p1, p2;
2. p1 = Create(100);
3. p2 = Create(100);
4. Release(p1);
5. Release(p2);
6. p2 = Create(100);
7. Release(p1);
```

(b) An example program that we track the state transition.

Instances at line 3	$RESOURCE
State 1	pointer_1
State 1	pointer_2

Instances at line 4	$RESOURCE
State 2	pointer_1
State 1	pointer_2

Instances at line 6	$RESOURCE
State 2	pointer_1
State 3 (END)	pointer_2
State 1	pointer_2

Instances at line 7	$RESOURCE
State 4 (BUG)	pointer_1
State 1	pointer_2

(c) The queue of the instances during the execution of the program in Fig. 5.3(b).

Fig. 5.3: The construction and dynamic state transition tracking of the state machine from the bug specification in Fig. 5.2(b). Each state of Fig. 5.3(a) corresponds to a basic unit in the bug specification in Fig. 5.2(b). State 1 is the start state, and state 3 and state 4 are the final states, but only state 4 will report the bug. By instrumenting the state machine into the program in Fig. 5.3(b), we can track the state transition of the program as shown in Fig. 5.3(c). PROPGUARD tracks multiple instances at the same time, each with its own current state ID and values (*e.g.*, `pointer_*` are the concrete values of `$RESOURCE`). An instance will be created at the start state and will be removed when it reaches a final state. Different basic units might have the same code pattern. For example, `Create` corresponds to state 1 and state 3.

5.3.2 Finite State Machine Construction

Given the bug specification, we need to find an efficient way to check whether the executed code has matched the pattern during fuzzing. Essentially, this involves translating the bug specification into a practical fuzzing oracle. One option is to use static analysis to find all the possible program paths that match the bug pattern and then instrument the program to signal the fuzzer when any of the path is matched. However, it is infeasible to find all the program paths statically and accurately, especially for large and complex programs. And keeping track of such a large amount of paths will incur huge overhead and lose the scalability of fuzzing.

Instead, we decide to use a state machine to model the bug pattern and track its transition during fuzzing based on the following reasons. Firstly, program execution can be modeled as a finite state machine [214]. The bug pattern, which describes a slice of the execution, should be able to similarly be represented as such. Secondly, matching bug patterns through tracking the state transition can be efficiently implemented, as we can analyze the flow dependencies and decide the possible locations that states might transition and ignore the others. This allows us to reduce the overhead of matching, which is crucial for fuzzing.

Therefore, we transform the bug specification into a finite state machine. Each basic unit is a state. The first basic units that we try to match are the start states, and the last to match basic units are the end states (or final states). There are two types of end states. The first type indicates that we cannot match the pattern if we have met this state so that we should stop. The second type indicates that we have matched the pattern if we have met this state so that we should report the bug. The control and data flow dependencies between the basic units are transformed into the transitions among the corresponding states. If the flow dependencies indicate that a basic unit can be matched after another basic units, there is a transition between the two states. The basic units wrapped by the `not` operator become stopping end states. If they are met, we should stop matching the pattern. Otherwise, their previous states can transition to next states without matching the stopping states.

For example, the bug specification in Fig. 5.2(b) can be transformed into the state machine shown in Fig.5.3(a). There are four basic units in the bug specification, so there are four states in the state machine. The first `Create` is the start state, meaning that the pattern matching will not start until a `Create` is met. The second state (State 2) can transition into two final states (State 3 and State 4) based on the flow dependency. When the state machine reaches State 3, it means that the pattern cannot be matched, because the resource is already reallocated. As State 3 is a `not` state, State 2 can transition to State 4 without matching State 3. When the state machine reaches State 4, it means that the pattern is matched, because the resource is freed twice consecutively. With more complex bug specifications, the state machine will have more states and transitions.

5.3.3 Dynamic State Transition

Once we have the finite state machine that models the bug, PROPGUARD instruments the program under test to track the state transition during the execution. As discussed, a state models a basic unit, which consists of the code pattern and the related values represented by the variables. During dynamic execution, the concrete values can be different in different occurrences of the same code pattern (*e.g.*, a function get called multiple times, or different code locations calls the same function.). This means that the same code pattern with different concrete values should be tracked differently.

For example, in the example program Fig. 5.3(b), line 2 and line 3 both call `Create` and thus should be in state 1 of Fig. 5.3(a). However, during dynamic execution, the two `Create` calls will return different pointer values (`p1` and `p2`), which requires us to track them separately. Otherwise, we might mistakenly report a bug when the two `Release` calls are executed at line 4 and line 5.

Therefore, instead of tracking the transition of only one global state machine, we track multiple, each of which is called a **state instance**. A state instance is a copy of the state machine with the unique values of the variables. Each instance is tracked independently. To achieve our goal, we instrument the program a transition point for each code location that is matched by any basic unit. For example, the bug specification in Fig. 5.2(b) will instrument a transition point to every calls to `Create` and `Release`. The transition point is a code segment that performs the state instance management in five steps: **Context Collection**, **Instance Filtering**, **Operation Execution**, **State Transition** and **Bug Reporting**. Let us describe each step in detail below with a complete example in Fig. 5.3, where we instrument the program (Fig. 5.3(b)) to track the transition of the state machine (Fig. 5.3(a)).

Context Collection. As a basic unit not only looks for a code pattern but also the values of related variables, we need to collect the concrete values from the execution context. Whenever we reach a transition point, we assign the values specified by the specification into the corresponding variables.

In Fig. 5.3(b), when we reach the transition point of `Create` at line 2, `$RESOURCE` will be assigned the concrete value `$p1=pointer_1`. When we reach line 3, `$RESOURCE` will be assigned the concrete value `$p2=pointer_2`.

Instance Filtering. After collecting the relevant context, the next step is to determine which state instances to operate on. If the transition point corresponds to a start state, a new instance of the state machine is created, containing the special state ID `0` and the previously collected values. These instances are then stored in a queue. If the transition point does not correspond to a start state, we then filter the instances in the queue by the state ID and the values. An instance is selected for the next step if its state ID matches one of the potential next states and its values align with those collected. Instances that do not meet these criteria remain in the queue.

In Fig. 5.3, when we reach line 2 and line 3, we are in the transition point of `Create`, which corresponds to the start state 1 and the intermediate state 3. And we try to handle both states at the same time. As State 1 is a start state, we create two instances with the ID 1 but different `$RESOURCE` values (`p1=pointer_1` and `p2=pointer_2`), as shown in the top left of Fig. 5.3(c). These two instances continue to the next step for the State 1 handling. For State 3, there are no instances in the queue, so we skip the State 3 handling.

Operation Execution. Once we have all the filtered instances, we can execute operations on the collected values as specified in the specification. The results of the operations are stored for the next step. In the example of Fig. 5.3, the operations are empty, so we skip this step, while in Fig. 5.2(a), we perform regex string matching and taint checking operations at line 11 and 12.

State Transition. After executing the operations, we transition the state instances to the next state by setting the state ID of the instances to be the state ID corresponding to the transition point. If the state ID is a first-type end state (bug pattern is matched), we go to the next step. If the state ID is a second-type end state (pattern cannot be matched), we discard the instance from the queue. For all other intermediate states, we push the instances back to the queue.

Continue with Fig. 5.3, the two instances created at line 2 and line 3 are transitioned into State 1, so their State ID change from `0` to 1. At line 4, we see a `Release`, which corresponds to the intermediate state 2 and the end state 4. However, all the two instances are in state 1 and thus expect to transition to state 2, so only the transition point of state 2 is handled. By checking the values of the instances, we find that only the first instance (`p1 = pointer_1`) matches the values of the transition point, so we transition the first instance to state 2. The second instance is untouched at line 4 and then transitions to State 2 at line 5 similarly. At line 6 (bottom left of Fig. 5.3(c)), we see another `Create`. The second instance matches the value and thus transitions to state 3, which is an end state without a triggering bug. The instance is then removed from the queue. Besides, `Create` also corresponds to the start state, therefore a new instance of State 1 with `p2=pointer_2` is created and pushed to the queue.

Bug Reporting. If we reach a first-type end state, we report the bug to the fuzzer. The bug report contains information such as the state ID, the values of the variables. To utilize the bug reporting capability of the underlying fuzzing engine, we then simply throw an unrecoverable exception to the fuzzer, which will then collect the corresponding test case and other information for further analysis.

At line 7 of Fig. 5.3(b), we see a `Release` with `$RESOURCE = pointer_1`. Only the first instance matches the value and thus transitions to state 4. State 4 is an end state with a triggering bug, so we report the bug to the fuzzer.

5.3.4 Directed Fuzzing

In traditional fuzzing, fuzzer probes the program randomly without a specific target in mind because there is no bug modeling and the fuzzer can only rely on program crash. However, PROPGUARD provides us with a definitive objective to uncover, the user-specified bug pattern. Therefore, our testing should be strategically concentrated on this particular bug pattern rather than aimlessly investigating irrelevant sections of code. That is, we need to further optimize the bug searching capability of fuzzing.

We employ directed fuzzing in PROPGUARD to steer our efforts toward triggering the specified bug. To achieve our goals, we perform static analysis on the source program with the help of the bug specification, and extract the following guidance information: 1. What code segments can possibly lead to the matching of the bug pattern, which we use to debloat all the unrelated code in code coverage tracking. 2. At what code the state transition can happen, which we use to augment the feedback guidance to consider not only new code coverage but also new state transition. 3. What are the possible values and magic numbers involved in the bug specification, which we use to generate a dictionary that improves the chances of following the bug triggering program path.

Code Coverage Debloating. We prune the code coverage tracking to only track the control flow that can possibly lead to a full matching of the bug pattern. For example, if we want to find a double free bug, we will skip tracking the coverage of the code without any memory allocation because they will not have subsequent free operations. To refine our code coverage analysis, we employ the Control Flow Graph (CFG) analysis to identify all basic blocks that could potentially execute the code relevant to the bug pattern. As we are looking for basic blocks instead of program paths, we will not suffer from the path explosion problem. Then we selectively instrument only these identified basic blocks, excluding others, during the code coverage collection instrumentation.

Feedback Guidance Augmentation. Next, we augment the feedback mechanism by considering the state transition as a feedback. As previously explained, the state transitions occur only at transition points when specific control and data flow criteria are met. This indicates that the same segment of code might be executed repeatedly, but only certain executions will prompt the state change. Code coverage metrics alone fail to differentiate these scenarios. Therefore, besides code coverage, we also monitor state transitions and use them as feedback guidance. We instrument the program to log the IDs of state transitions into a shared memory buffer during the program execution. After each execution, the fuzzer checks the buffer as well as the code coverage bitmap. If a particular input does not lead to new code coverage but induces a new state transition, it is still valued as an effective input. Such inputs will be saved into the fuzzing corpus and used for further mutation.

Dictionary Generation. From the above code coverage analysis, we figure out which code segments are relevant to the bug pattern. Also, the operations in the bug specification tell us what kind of values we look for to trigger the bug. From this, we compile a dictionary encompassing all potential values and magic numbers derived from both the specification and associated code segments. For example, we extract all the magic numbers in the compare instructions of all the related code segments. When generating inputs, we strategically utilize the dictionary by randomly inserting the values into the generated inputs to enhance the probability of matching the bug pattern, which is also used in the mutation existing fuzzers [12, 2].

It should be noted that the aforementioned techniques are the applications of existing state-of-the-art techniques in directed fuzzing. We are not proposing new techniques but instead adapting these techniques to fully utilize the information from the bug specification into PROPGUARD to guide the fuzzing towards the bug triggering direction.

5.4 Implementation

We implement PROPGUARD in 7,016 lines of code. Table 5.1 shows the breakdown. In this section, we first describe the features of the specification language in §5.4.1. Next,

Table 5.1: Line of codes of different components of PropGuard, which sum up to 7,016 lines.

Module	Language	LOC
Specification Language & Analysis	Python, Antlr	1335
Oracle Instrumentation	C++	2928
Directed Fuzzing	C++	614
`AFLplusplus` integration	C++	47
Total		**7,016**

we describe how we debloat the code coverage in directed fuzzing by utilizing the bug specification in §5.4.2. Lastly, we describe how instrument the oracle into the tested program in §5.4.3 and integrate PropGuard with `AFLplusplus` to make it an end to end fuzzing framework in §5.4.4.

5.4.1 Specification Features

The specification language in PropGuard is intentionally crafted for both expressiveness and user-friendliness. It adopts a unified format for specifying a basic unit, streamlining the writing process while offering robust operation primitives and various composition connectors to ensure expressivity. A basic unit's structure is segmented into four essential components: defined variables, a matching pattern, operation computations, and a validation predicate to confirm requirement fulfillment on the operations or end the state tracking early. The defined variables are defined to reference concrete values during dynamic execution and used within the matching pattern, operation computations, and the predicate. The name of defined variables in PropGuard starts with a dollar sign (`$`) to distinguish itself from the code symbols in the source program in the matching pattern. For example, in the matching pattern `SendRequest($URL)` at line 9 of Fig.5.2(a), we are matching all the calls to `SendRequest` with the value of its first argument captured into the variable `$URL`. The defined variables are typed which is important for context collection. The currently supported types include integer, string, boolean and some commonly used types from the standard library such as `std::string` and `std::vector`. The operation primitive is a set of predefined calculations that allow users to transform the captured values into a desired result. The currently supported operations include arithmetic operations (*e.g.*, addition, multiplication), string operations (*e.g.*, string concatenation, regular expression matching), and taint tracking operations, etc. Moreover, to facilitate interaction among various basic units, the language allows for the storage of computed outcomes in a shared dictionary accessible by different basic units. For example, in one basic unit, we can execute the operation `Set("key", 1)` and in another basic unit, we can execute the operation `Get("key")` to retrieve the value 1. A predicate is a check on the computed result. After we have computed the operations, we assert that the computed result satisfies the predicate or ends the tracking of the instance early. The current supported constraint primitives include true, false, equal, not equal, greater than and less than, etc.

5.4.2 Related Code Detection

For code coverage debloating, we need to identify which code locations match the pattern in the basic units and which code paths can potentially lead to a full match of the pattern. Identifying code segments is a straightforward task, as we can perform regular expression matching to the pattern against all the basic blocks in the program. However, identifying full paths that could lead to a pattern match is more complex as we might suffer from path explosion problem. To do so efficiently, instead of finding all the paths from the entry to the exit, we find program paths between every pair of states (*i.e.*, basic units) that form a transition in the state machine. For each pair of states, say, a source state and a destination state, we find the paths between them by performing **two-direction sweep**. The first sweep starts from the source state by traversing downwards the control flow graph of the program and marking all the basic blocks that can be reached from the source state. The second sweep starts from the destination state by traversing upwards the control flow graph of the program and marking all the basic blocks that can reach the destination state. The intersection of the two sets of marked basic blocks is the set of basic blocks that exist on the paths between the source state and the destination state. By performing the two-direction sweep for all the pairs of states, we can find relevant code paths efficiently. For example, in the SSRF specification (Fig. 5.2(a)), we first find all the basic blocks that can be reached by the block calling the `readn` function. Then we find all the basic blocks that can reach the block calling the `SendRequest` function. The intersection of the two sets of basic blocks are considered as related code blocks.

5.4.3 Code Instrumentation

In PropGuard, we utilize the LLVM compiler infrastructure [215] to facilitate program instrumentation effectively. This instrumentation serves three critical purposes: monitoring code coverage, tracking state transitions within the finite state machine, and implementing dynamic taint tracking as needed. For code coverage, we apply sanitizer coverage with a focus on efficiency. Our method involves pruning unnecessary paths as discussed in §5.3.4. We only instrument paths that are potentially relevant to the bug pattern match. This targeted approach optimizes performance by reducing extraneous instrumentation. For the state transition tracking, instead of instrumenting all the tracking logics such as checking the data flow into the program itself, we insert a call to the tracking library where we put all the tracking logics. The program itself only incorporates the necessary context collection logic. It then forwards this context to the tracking library. This strategy minimizes direct code alterations and promotes logic reuse across different components. For the dynamic taint tracking, we build a library on top of the LLVM data flow sanitizer [216] for users to easily mark the buffer as tainted and check whether a value is tainted. Through these tools, PropGuard provides comprehensive and efficient instrumentation capabilities, enriching its analysis and tracking functionalities.

5.4.4 Fuzzing Engine Integration

The design of PropGuard is independent of the underlying fuzzer. Therefore, we want to maintain PropGuard's autonomy from any specific fuzzer. To achieve this, we designed PropGuard as an independent module. It starts by parsing the bug specification. Following that, it generates an analysis outcome. Additionally, we developed an LLVM pass that utilizes this analysis. Its role is to instrument the program. This instrumentation helps in monitoring state transitions.

Such a design facilitates the integration of PROPGUARD with any fuzzer. Currently, we integrate PROPGUARD with `AFLplusplus` [2] to leverage its rich set of features and optimizations. For that, we modify the compiler wrapper of `AFLplusplus` to add the linkage of PROPGUARD's LLVM pass and the coverage instrumentation to selectively instrument the code based on the analysis result of PROPGUARD. The specification parsing and static analysis works as an independent standalone module.

5.5 Evaluation

Our evaluation aims to answer the following questions:

- What kind of vulnerabilities does PROPGUARD support finding? (§5.5.2)
- Can PROPGUARD detect non-crashing vulnerabilities that existing fuzzers cannot detect? (§5.5.3)
- Can PROPGUARD find new non-crashing vulnerabilities in real-world software? (§5.5.4)

5.5.1 Evaluation Setup

Benchmark. We evaluate the effectiveness of PROPGUARD on eight categories of bugs from top CWE [210], including command injection, path traversal, information disclosure, uncontrolled resource consumption, server-side request forgery, double free, out of bounds memory access, and null pointer dereference. For five of the non-crashing bug categories (command injection, path traversal, information disclosure, uncontrolled resource consumption, server-side request forgery), we further collect 10 real-world bugs from popular open-source C/C++ projects that are extensively tested by OSS-Fuzz[4], including `imageMagick` [217], `tinygltf` [218], `httpd` [219], `librsvg` [220], `ffmpeg` [221], `libarchive` [222], `libpng` [223]. We focus on C/C++ projects because of PROPGUARD's

Table 5.2: Types of bugs supported by PropGuard and the compared tools. A tool supports a bug type if it supports modeling the bug type and can produce an alert when the bug pattern is matched instead of relying on program crashes. We collect the bug types from top CWEs, including 5 non-crashing types (command injection, path traversal, information exposure, uncontrolled resource consumption and server-side request forgery) and 3 crashing types (use after free, out-of-bound memory access, and null pointer dereference). For each bug type, we craft a synthetic program that contains the bug and evaluate the detection capabilities of PropGuard and the compared tools. We provide the oracles of the same semantics in the specification language of PropGuard and `CodeQL` respectively. For assertion, we try to describe the bug pattern with the standard C/C++ assertion library.

Project \ Bug Type	Command Injection	Path Traversal	Information Exposure	Uncontrolled Resource Consumption	SSRF	Double Free	OOB	Null Pointer Dereference
PropGuard	✔	✔	✔	✔	✔	✔	✔	✔
`CodeQL`	✔	✔	✔	✔	✔	✔	✔	✔
Assertion	✗	✗	✗	✗	✗	✗	✗	✔
`Address Sanitizer`	✗	✗	✗	✔	✗	✔	✔	✔
`System Sanitizer`	✔	✗	✗	✗	✗	✗	✗	✗
`FuzzFactory`	✗	✗	✗	✔	✗	✗	✗	✔

current integration with `AFLplusplus`, which is tailored for these programming environments. We compare PropGuard with assertion that is supported by the C/C++ programming language, the static analysis tool `CodeQL`, the state-of-the-art fuzzer `AFLplusplus`, bug-specific sanitizers `Address Sanitizer`, `System Sanitizer`, and `FuzzFactory` [224]. For PropGuard and `CodeQL`, we write an oracle for the bug pattern with their own specification language for each bug individually. For the sanitizers, we run them with `AFLplusplus` as the underlying fuzzing engine to have a fair comparison of PropGuard. We use the default settings and the same fuzzing seeds, which are the test cases collected from the projects' test suites.

Environment. We perform our evaluation on a server with an AMD 24-core CPU, 256GB RAM, and an Ubuntu 22.04 operating system. The versions of the compared tools are `AFLplusplus` (4.07a), `Address Sanitizer` (LLVM-Clang 15.0), `System Sanitizer` (commit a836616), `CodeQL` (2.6.0), and `FuzzFactory` (commit e02d43). For assertion, we use the assertion supported by the standard C/C++ library. For each fuzzing instance (one bug category and one project), we run it for 24 hours and repeat the experiment 5 times.

Each fuzzing instance is run in a separate and clean docker container with 1 CPU and 15GB RAM allocated.

5.5.2 Supported Bug Types

To ascertain the range of vulnerabilities identifiable by PROPGUARD, we assemble a collection encompassing 8 distinct bug categories. This compilation features 5 types of non-crashing vulnerabilities, such as SSRF and path traversal, alongside 3 types of crashing bugs, including double free, out-of-bound memory access and null pointer dereference. For each category, we craft synthetic test programs as PoCs, embedding a representative bug to evaluate detection capabilities. Our experimental process involves deploying PROPGUARD and various other comparative tools to discern their efficacy in identifying these embedded bugs. A bug is deemed successfully identified only if the tool reports the bug without relying on program crashes.

The result is shown in Table 5.2. As we can see, PROPGUARD and `CodeQL` support finding all eight evaluated bug types, while the other tools can find at most four. The superior performance of PROPGUARD and `CodeQL` stems from their expressive specification languages which allows users to describe a wide range of bug patterns, as opposed to other tools that target limited, predefined bug categories. Notably, all bugs pinpointed by PROPGUARD generate concrete test cases as the PoCs. In contrast, `CodeQL` primarily flagged potential concerns that might be false positives upon closer examination. The synthetic programs are relatively simple, so the false positive rate of `CodeQL` is low in this evaluation. `Address Sanitizer` mainly focuses on finding memory-related bugs by modeling and monitoring the memory. Therefore, besides the three memory corruption bugs (double free, out-of-bound, null pointer dereference), it also finds the uncontrolled resource consumption bug, which `Address Sanitizer` checks when excessive memory is allocated. `System Sanitizer` focuses on finding command injection and `FuzzFactory` provides several hooks to find some specific types of bugs, which are not expressive enough for complex bugs. Noted that to find command injection, we only need to write less than

30 lines of specification using PropGuard, while `System Sanitizer` consists of about 500 lines of C++ code for the same purpose. By embedding comprehensive bug pattern specifications within a fuzzing context, PropGuard significantly broadens the spectrum of detectable bug types, exemplifying its versatility and depth in uncovering a wide array of software vulnerabilities.

5.5.3 Non-crashing Bug Detection

Table 5.3: Real-world bugs found by PropGuard and the compared tools. The evaluated bug types include command injection, path traversal, information disclosure, uncontrolled resource consumption, and SSRF. For each of the bug types, we collect two real-world bugs from the public bug databases. Each bug is evaluated on the commit of the project that contains the bug. The complete information can be found at Table 5.4. An additional bug type, crash, is added for a comprehensive evaluation. As long as the tool finds any crashing bug, the "Crash" category is marked as ✔. The fuzzing instances are run for 24 hours.

Bug / Project	Command Injection		Path Traversal		Information Disclosure		Uncontrolled Resource Consumption		SSRF		Crash
	`imageMagick`	`tinygltf`	`httpd`	`librsvg`	`ffmpeg`	`imageMagick`	`libarchive`	`libpng`	`ffmpeg`	`imageMagick`	*
PropGuard	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
`PropGuard-Undirected`	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔
`PropGuard-NoOracle`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔
`AFLplusplus`	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔
`Address Sanitizer`	✗	✗	✗	✗	✗	✗	✔	✔	✗	✗	✔
`System Sanitizer`	✔	✔	✗	✗	✗	✗	✗	✗	✗	✗	✔
`FuzzFactory`	✗	✗	✗	✗	✗	✗	✔	✔	✗	✗	✔

To evaluate PropGuard's effectiveness in finding non-crashing bugs, we test it on ten real-world non-crashing bugs of five categories from popular open source projects, including `imageMagick`, `tinygltf`, `httpd`, `librsvg`, `ffmpeg`, `libarchive` and `libpng`. For each collected bug, we access the specific repository commit where the bug was present. Given that these old project versions also harbored certain known crashing bugs, we inclusively track these as an additional category called "Crashes" for a comprehensive analysis. We compare the performance of PropGuard with `AFLplusplus`, `Address Sanitizer`, `System Sanitizer` and `FuzzFactory`. To better understand the contribution of components in PropGuard, we also include `PropGuard-Undirected`, which is PropGuard without the bug oriented directed searching, and `PropGuard-NoOracle`, which is PropGuard without the bug revelation (*i.e.*, the oracle). Note that since bug specification is used for bug revelation and bug oriented searching, `AFLplusplus` is PropGuard without the bug revelation and the directed fuzzing. The result of `CodeQL` is skipped because it produces too many false positives which are difficult to verify and not meaningful for comparison. Similarly, the result of assertion is omitted because assertions lacks the expressiveness to model these sophisticated non-crashing bug scenarios as shown in Table 5.2.

As illustrated in Table 5.3, PropGuard identifies all ten targeted non-crashing bugs, whereas the comparative metrics for `AFLplusplus`, `Address Sanitizer`, `System Sanitizer`, and `FuzzFactory` were significantly lower, detecting none, two, two, and two of these bugs, respectively. The result is consistent with that in Table 5.2, showing that the bug-type specific oracles like `Address Sanitizer` and `System Sanitizer` are effective in finding the bugs of their supported types, but not in the other types. PropGuard, on the other hand, can find various types of non-crashing bugs with only the extra effort of modeling the bug types. Benefiting from its specification language, the specification we write for each bug is fewer than 100 lines, typically taking authors less than one hour to complete on average. This requires much less development efforts than that of developing a new sanitizer. All five tools find crashing bugs during the evaluation, because they are all `AFL`-based fuzzers, which can find crashing bugs without additional modeling. This shows that, PropGuard, like the other fuzzing oracles, can find non-crashing bugs without sacrificing the ability to find crashing bugs. Additionally, `PropGuard-Undirected` finds the same bugs as PropGuard, while `PropGuard-NoOracle` finds only crashing bugs. We investigate the results and find that with directed fuzzing, PropGuard can find the bugs faster than `PropGuard-Undirected` by from 4% to 13%. And while `PropGuard-NoOracle` actually reaches some non-crashing buggy states, it fails to report the bugs because it cannot decide whether these states are buggy as they do not crash. In conclusion, by combining the power of specifying bug patterns through specification and searching the pattern through directed fuzzing, PropGuard can find non-crashing bugs effectively.

Table 5.4: The complete information of the 10 public real-world bugs evaluated in §5.5.3. SSRF stands for Server-Side Request Forgery. URC stands for Uncontrolled Resource Consumption. The order follows the same as in §5.5.3.

Project	Bug Type	Version	Reference
`imageMagick`	Command Injection	6.9.3-9	CVE-2016–3714
`tinygltf`	Command Injection	544969b	CVE-2022-3008
`httpd`	Path Traversal	2.4.49	CVE-2021-41773
`librsvg`	Path Traversal	2.56.3	CVE-2023-38633
`ffmpeg`	Information Disclosure	2.8.2	CVE-2016-1897
`imageMagick`	Information Disclosure	7.1.0-51	CVE-2022-44268
`libarchive`	URC	1.2.56	CVE-2019-2023
`libpng`	URC	646f620	CVE-2019-2023
`ffmpeg`	SSRF	2.8.2	CVE-2016-1898
`imageMagick`	SSRF	6.9.3-9	CVE-2016-3718

5.5.4 New Bug Discovery

PROPGUARD has proven its efficacy by identifying two previously undetected non-crashing issues within the `tinygltf`, a widely-used third-party library. The first bug is incorrect file path handling, where an encoding function produces an incorrect file path for opening and might lead to information disclosure. The second bug involves arbitrary file reading, where a maliciously crafted GLTF file can access arbitrary files on the system that parses files with `tinygltf`. We have reported the bugs to the maintainers of `tinygltf`. Both bugs are confirmed, and the first bug has already been fixed. Finding these unknown bugs demonstrates the effectiveness of PROPGUARD in finding non-crashing bugs in real-world software. We skip the detailed information of the bugs now for anonymous review.

5.6 Discussion

5.6.1 Support More Programming Languages

While PROPGUARD currently supports C/C++ applications in order to integrate with `AFLplusplus`, its architecture harbors the potential to extend across a diverse array of programming languages. The push towards memory-safe languages, advocated by both governmental and industrial entities for enhanced security [225], underscores the necessity to broaden PROPGUARD's applicability to detect non-crashing bugs in such environments. One strength of PROPGUARD lies in its modular architecture, particularly its language-agnostic specification language. This design choice ensures that the core components of PROPGUARD, its bug specification and oracle construction with finite state machine, are transferable across different programming contexts without being tethered to a specific language syntax or runtime. To extend PROPGUARD's utility to additional languages, the primary requirement is the integration with language-specific static analysis and code instrumentation tools, alongside compatible fuzzing engines. Fortunately, many modern programming languages have mature static analysis frameworks and fuzzing frameworks [226, 227, 228]. We plan to support more languages such as Rust and JavaScript in the future.

5.6.2 Overhead

The primary sources of overhead in PROPGUARD are attributed to the mechanisms for tracking state machine transitions and conducting dynamic taint analysis. Given the potential necessity to monitor numerous state machine instances concurrently, the overhead at each monitoring point is inherently tied to the count of these instances under surveillance. Through our evaluation process, we observed that PROPGUARD's time overhead fluctuates between 1.5 to 3 times the baseline, whereas its memory consumption generally doubles. When placed in context with similar tools, these overhead figures align closely with those experienced when deploying standard sanitizers like `Address Sanitizer`.

Considering the instrumental role of such tools in unveiling non-crashing bugs—issues that are notoriously subtle and elusive, we believe that the incurred overhead is deemed justifiable. In the future, we plan to perform deeper analysis to reduce the overhead such as merging similar state instances.

5.6.3 Automation by Large Language Models

While PROPGUARD streamlines the bug detection process, it inherently requires users to articulate pattern specifications manually, which might require users to have a deep comprehension of the project under test. However, the emergence of advanced Large Language Models (LLMs) like GPT [229] heralds new possibilities for automating this aspect of PROPGUARD. By harnessing these models, one could potentially generate intricate pattern specifications automatically, significantly reducing the cognitive and temporal investment from the users. More importantly, the integration of LLMs into PROPGUARD could revolutionize its operational model, especially when contrasted with static analysis tools like `CodeQL` [206] or SemGrep [211]. While these tools excel at identifying code patterns indicative of bugs, they do not generate concrete test cases or proof of concepts (POC). PROPGUARD, empowered by LLMs, could not only identify these patterns but also produce actionable test cases that facilitate direct and automated bug replication and validation.

We believe that by combining the advantages of fuzzing and the emerging capability of LLM in understanding large code spaces, we could dramatically elevate the automation level in non-crashing bug discovery processes using PROPGUARD.

CHAPTER 6
CONCLUSION

This dissertation focuses on addressing the inherent limitations of existing fuzzing methodologies, particularly in the realms of program state space exploration and bug modeling. Through a comprehensive analysis and three innovative approaches, we have demonstrated not only the theoretical viability of these solutions but also their practical effectiveness in enhancing the capabilities of fuzzing tools.

Our work began with an insightful examination of the current landscape of fuzzing techniques, identifying critical gaps in both bug searching and bug modeling. The effectiveness of fuzzing in uncovering software vulnerabilities is well-received, yet its potential remains hampered by inadequate exploration of program state spaces and a lack of bug modeling mechanisms. By dissecting these challenges, we set the foundation for our research contributions, which span two primary dimensions: the enhancement of bug searching both vertically and horizontally and the refinement of bug modeling through a novel specification language.

We first present, POLYGLOT, a framework for improving test cases generation. By incorporating a unified representation of input syntax and semantics, POLYGLOT facilitates the generation of semantically valid test cases, thereby achieving deeper and more meaningful program state exploration. The empirical success of POLYGLOT, evidenced by its identification of significant vulnerabilities across various language processors, underscores the practical value of our research.

Parallel to our advancements in bug searching, we introduced μFUZZ, a fuzzing framework that reimagines parallel fuzzing through a microservice architecture. By decoupling the fuzzing process into asynchronous components, μFUZZ overcomes the inefficiencies of traditional parallel fuzzing, harnessing the full potential of multi-core environments.

This architectural innovation not only accelerates the fuzzing process but also enhances its breadth and depth, as demonstrated by μFUZZ's superior performance in code coverage and bug detection.

The final pillar of our research, PROPGUARD, addresses the critical need for bug modeling. By facilitating the definition and detection of diverse bug patterns, PROPGUARD empowers users to pinpoint specific vulnerabilities with ease and accuracy. This capability is particularly vital in the context of non-crashing bugs, which pose significant security risks yet often elude detection. Through its intuitive specification language and targeted searching mechanisms, PROPGUARD offers a robust tool for identifying and mitigates a wide array of software vulnerabilities.

In conclusion, this dissertation presents a holistic approach to enhancing fuzzing techniques, addressing both the breadth of program exploration and the depth of bug detection. The success of our proposed solutions, as evidenced by their empirical validation, not only contributes to the academic discourse but also offers practical tools for improving the resilience and reliability of software systems. Going forward, we plan to continue refining and expanding upon these methodologies, especially in the integration with the advancements of AI, with the ultimate goal of advancing the state of the art in software testing and security.

REFERENCES

[1] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing As Markov Chain."

[2] *Aflplusplus*, https://github.com/AFLplusplus/AFLplusplus, 2022.

[3] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels."

[4] Google, *OSS-Fuzz - Continuous Fuzzing For Open Source Software*, https://github.com/google/oss-fuzz, 2018.

[5] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be Sensitive And Collaborative: Analyzing Impact Of Coverage Metrics In Greybox Fuzzing."

[6] R. Zhong, Y. Chen, H. Hu, H. Zhang, W. Lee, and D. Wu, "SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Orlando, USA, Nov. 2020.

[7] *SQLSmith*, https://github.com/anse1/sqlsmith, 2016.

[8] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI'11, New York, NY, USA, 2011.

[9] S. Park, W. Xu, I. Yun, D. Jang, and T. Kim, "Fuzzing JavaScript Engines with Aspect-preserving Mutation," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2020.

[10] V.-T. Pham, M.-D. Nguyen, Q.-T. Ta, T. Murray, and B. I. Rubinstein, "Towards systematic and dynamic task allocation for collaborative parallel fuzzing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 1337–1341.

[11] Y. Wang, Y. Zhang, C. Pang, P. Li, N. Triandopoulos, and J. Xu, "Facilitating parallel fuzzing with mutually-exclusive task distribution," in *International Conference on Security and Privacy in Communication Systems*, Springer, 2021, pp. 185–206.

[12] M. Zalewski, *American Fuzzy Lop (2.52b)*, http://lcamtuf.coredump.cx/afl, 2019.

[13] *Fuzzing beyond memory corruption: Finding broader classes of vulnerabilities automatically*, https://security.googleblog.com/2022/09/fuzzing-beyond-memory-corruption.html, 2022.

[14] Wikipedia contributors, *Backus-naur form — Wikipedia, the free encyclopedia*, [Online; accessed 1-September-2020], 2020.

[15] G. Ye *et al.*, "Automated conformance testing for javascript engines via deep compiler fuzzing," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021, pp. 435–450.

[16] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," *SIGPLAN Not.*, vol. 43, no. 6, pp. 206–215, Jun. 2008.

[17] MozillaSecurity, *funfuzz*, https://github.com/MozillaSecurity/funfuzz, 2020.

[18] P. Xu, Y. Wang, H. Hu, and P. Su, "Cooper: Testing the binding code of scripting languages with cooperative mutation," in *29th Annual Network and Distributed System Security Symposium.*, 2022.

[19] C. Lyu *et al.*, "Ems: History-driven mutation for coverage-based fuzzing," in *29th Annual Network and Distributed System Security Symposium. https://dx. doi. org/10.14722/ndss*, 2022.

[20] S. Veggalam, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *European Symposium on Research in Computer Security*, Springer, 2016, pp. 581–601.

[21] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016, Singapore, Singapore.

[22] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, Montpellier, France: Association for Computing Machinery, 2018, pp. 259–269, ISBN: 9781450359375.

[23] C. Mishra, N. Koudas, and C. Zuzarte, "Generating targeted queries for database testing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08, Vancouver, Canada: Association for Computing Machinery, 2008, pp. 499–510, ISBN: 9781605581026.

[24] J. Wang, P. Zhang, L. Zhang, H. Zhu, and X. Ye, "A model-based fuzzing approach for dbms," in *2013 8th International Conference on Communications and Networking in China (CHINACOM)*, Aug. 2013, pp. 426–431.

[25] B. Chandra, B. Chawda, B. Kar, K. V. M. Reddy, S. Shah, and S. Sudarshan, “Data generation for testing and grading sql queries,” *The VLDB Journal*, vol. 24, no. 6, pp. 731–755, Aug. 2015.

[26] K. Dewey, J. Roesch, and B. Hardekopf, “Language fuzzing using constraint logic programming,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE ’14, Vasteras, Sweden: Association for Computing Machinery, 2014, pp. 725–730, ISBN: 9781450330138.

[27] H. Han, D. Oh, and S. K. Cha, “Codealchemist: Semantics-aware code generation to find vulnerabilities in javascript engines.”

[28] V. Vikram, R. Padhye, and K. Sen, “Growing a test corpus with bonsai fuzzing,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, IEEE, 2021, pp. 723–735.

[29] K. Dewey, J. Roesch, and B. Hardekopf, “Fuzzing the rust typechecker using clp (t),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2015, pp. 482–493.

[30] E. Lo, C. Binnig, D. Kossmann, M. Tamer Özsu, and W.-K. Hon, “A Framework for Testing DBMS Features,” *The VLDB Journal*, vol. 19, no. 2, pp. 203–230, Apr. 2010.

[31] S. Abdul Khalek and S. Khurshid, “Automated SQL Query Generation for Systematic Testing of Database Engines,” in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE10, Antwerp, Belgium, 2010.

[32] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut, “Test Input Generation for Database Programs Using Relational Constraints,” in *Proceedings of the Fifth International Workshop on Testing Database Systems*, ser. DBTest12, Scottsdale, Arizona, 2012.

[33] Alloy, *Alloy - Documentation of Alloy SAT solver*, https://alloytools.org/documentation.html, 2019.

[34] J. Viide *et al.*, “Experiences with model inference assisted fuzzing,” in *Proceedings of the 2nd Conference on USENIX Workshop on Offensive Technologies*, ser. WOOT’08, San Jose, CA: USENIX Association, 2008.

[35] O. Bastani, R. Sharma, A. Aiken, and P. Liang, “Synthesizing program input grammars,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017, Barcelona, Spain: Association for Computing Machinery, 2017, pp. 95–110, ISBN: 9781450349888.

[36] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2016, pp. 720–725.

[37] R. Gopinath, B. Mathis, M. Höschele, A. Kampmann, and A. Zeller, "Sample-free learning of input grammars for comprehensive software fuzzing," *arXiv preprint arXiv:1810.08289*, 2018.

[38] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 803–817.

[39] LLVM, *LibFuzzer - A Library For Coverage-guided Fuzz Testing*, http://llvm.org/docs/LibFuzzer.html, 2017.

[40] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, Aug. 2018.

[41] P. Chen and H. Chen, "Angora: Efficient Fuzzing By Principled Search."

[42] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence," in *Symposium on Network and Distributed System Security (NDSS)*, 2019.

[43] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution.," in *NDSS*, vol. 16, 2016, pp. 1–16.

[44] Y. Chen *et al.*, "Savior: Towards bug-driven hybrid testing."

[45] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[46] S. Gan *et al.*, "GREYONE: Data Flow Sensitive Fuzzing."

[47] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[48] M. Wu *et al.*, "One fuzzing strategy to rule them all," in *Proceedings of the International Conference on Software Engineering*, 2022.

[49] P. Xu, Y. Wang, H. Hu, and P. Su, "Cooper: Testing the binding code of scripting languages with cooperative mutation.," in *NDSS*, 2022.

[50] C. Lemieux and K. Sen, “Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage,” in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 475–485.

[51] L. Binosi, L. Rullo, M. Polino, M. Carminati, S. Zanero, *et al.*, “Rainfuzz: Reinforcement-learning driven heat-maps for boosting coverage-guided fuzzing.,” in *ICPRAM*, 2023, pp. 39–50.

[52] H. L. Nguyen and L. Grunske, “Bedivfuzz: Integrating behavioral diversity into generator-based fuzzing,” in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 249–261.

[53] P. Jauernig, D. Jakobovic, S. Picek, E. Stapf, and A.-R. Sadeghi, “Darwin: Survival of the fittest fuzzing mutators,” *arXiv preprint arXiv:2210.11783*, 2022.

[54] P. Chen and H. Chen, “Angora: Efficient fuzzing by principled search,” in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 711–725.

[55] S. Bekrar, C. Bekrar, R. Groz, and L. Mounier, “A taint based approach for smart fuzzing,” in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, IEEE, 2012, pp. 818–825.

[56] C. Cadar, D. Dunbar, D. R. Engler, *et al.*, “Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs.,” in *OSDI*, vol. 8, 2008, pp. 209–224.

[57] V. Ganesh, T. Leek, and M. Rinard, “Taint-based directed whitebox fuzzing,” in *2009 IEEE 31st International Conference on Software Engineering*, IEEE, 2009, pp. 474–484.

[58] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “Vuzzer: Application-aware evolutionary fuzzing.,” in *NDSS*, vol. 17, 2017, pp. 1–14.

[59] P. Chen, J. Liu, and H. Chen, “Matryoshka: Fuzzing deeply nested branches,” in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 499–513.

[60] H. Peng, Y. Shoshitaishvili, and M. Payer, “T-fuzz: Fuzzing by program transformation,” in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 697–710.

[61] T. Cloosters, J. Willbold, T. Holz, and L. Davi, “{Sgxfuzz}: Efficiently synthesizing nested structures for {sgx} enclave fuzzing,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3147–3164.

[62] J. Corina *et al.*, “Difuze: Interface aware fuzzing for kernel drivers,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas, Texas, USA.

[63] T. Blazytko *et al.*, “GRIMOIRE: Synthesizing structure while fuzzing,” in *USENIX Security Symposium*, 2019.

[64] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, “A Genetic Approach for Random Testing of Database Systems,” in *Proceedings of the 33rd International Conference on Very Large Data Bases*, ser. VLDB07, Vienna, Austria, 2007, pp. 1243–1251.

[65] C. Aschermann, T. Frassetto, T. Holz, P. Jauernig, A.-R. Sadeghi, and D. Teuchert, “Nautilus: Fishing for deep bugs with grammars.,” in *NDSS*, 2019.

[66] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, “Semantic fuzzing with zest,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.

[67] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, “Slf: Fuzzing without valid seed inputs,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 712–723.

[68] J. Wang, B. Chen, L. Wei, and Y. Liu, “Skyfire: Data-driven seed generation for fuzzing,” in *2017 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017, pp. 579–594.

[69] P. Srivastava and M. Payer, “Gramatron: Effective grammar-aware fuzzing,” in *Proceedings of the 30th acm sigsoft international symposium on software testing and analysis*, 2021, pp. 244–256.

[70] B. Mathis, R. Gopinath, M. Mera, A. Kampmann, M. Höschele, and A. Zeller, “Parser-directed fuzzing,” in *Proceedings of the 40th acm sigplan conference on programming language design and implementation*, 2019, pp. 548–560.

[71] W. You *et al.*, “Profuzzer: On-the-fly input type probing for better zero-day vulnerability discovery,” in *2019 IEEE symposium on security and privacy (SP)*, IEEE, 2019, pp. 769–786.

[72] S. Groß, “Fuzzil: Coverage guided fuzzing for javascript engines,” *Master thesis, TU Braunschweig*, 2018.

[73] C. Holler, K. Herzig, and A. Zeller, “Fuzzing with code fragments,” in *Proceedings of the 21st USENIX Conference on Security Symposium*, ser. Security’12, Bellevue, WA: USENIX Association, 2012, p. 38.

[74] D. Yang, Y. Zhang, and Q. Liu, “Blendfuzz: A model-based framework for fuzz testing programs with grammatical inputs,” in *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*, IEEE, 2012, pp. 1070–1076.

[75] J. Wang, B. Chen, L. Wei, and Y. Liu, “Superion: Grammar-Aware Greybox Fuzzing.”

[76] Z. Wu, J. Liang, M. Wang, C. Zhou, and Y. Jiang, “Unicorn: Detect runtime errors in time-series databases with hybrid input synthesis,” in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 251–262.

[77] X. He *et al.*, “Sofi: Reflection-augmented fuzzing for javascript engines,” in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2229–2242.

[78] S. T. Dinh *et al.*, “Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases.,” in *NDSS*, 2021.

[79] S. Lee, H. Han, S. K. Cha, and S. Son, “Montage: A neural network language {model-guided}{javascript} engine fuzzer,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2613–2630.

[80] K. Even-Mendoza, A. Sharma, A. F. Donaldson, and C. Cadar, *Grayc: Greybox fuzzing of compilers and analysers for c.* 2023.

[81] Y. Liu, Y. Wang, P. Su, Y. Yu, and X. Jia, “Instruguard: Find and fix instrumentation errors for coverage-based greybox fuzzing,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 568–580.

[82] S. Gan *et al.*, “CollAFL: Path Sensitive Fuzzing.”

[83] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, “Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing,” in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 1–15.

[84] V. Ganesh, T. Leek, and M. Rinard, “Taint-based directed whitebox fuzzing,” in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 474–484.

[85] J. Liang *et al.*, “Pata: Fuzzing with path aware taint analysis,” in *IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, Los Alamitos, CA, USA*, 2022, pp. 154–170.

[86] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," *IEEE Transactions on Software Engineering*, vol. 45, no. 5, pp. 489–506, 2017.

[87] Y. Wang *et al.*, "Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization," NDSS, 2020.

[88] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 497–512.

[89] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "Ijon: Exploring deep state spaces via fuzzing," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1597–1612.

[90] S. Pailoor, A. Aday, and S. Jana, "{Moonshine}: Optimizing {os} fuzzer seed selection with trace distillation," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 729–743.

[91] Z. Shen, R. Roongta, and B. Dolan-Gavitt, "Drifuzz: Harvesting bugs in device drivers from golden seeds," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1275–1290.

[92] D. She, A. Shah, and S. Jana, "Effective seed scheduling for fuzzing with graph centrality analysis," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022, pp. 2194–2211.

[93] K. Zhang, X. Xiao, X. Zhu, R. Sun, M. Xue, and S. Wen, "Path transitions tell more: Optimizing fuzzing schedules via runtime program states," *arXiv preprint arXiv:2201.04441*, 2022.

[94] C. Lyu *et al.*, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966, ISBN: 978-1-939133-06-9.

[95] B. Zhao *et al.*, "{Statefuzz}: System {call-based}{state-aware} linux driver fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3273–3289.

[96] L. Zhao, Y. Duan, H. Yin, and J. Xuan, "Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing.," in *NDSS*, 2019.

[97] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2022, pp. 36–50.

[98] M. Wang, J. Liang, C. Zhou, Z. Wu, X. Xu, and Y. Jiang, "Odin: On-demand instrumentation with on-the-fly recompilation," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022, San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 1010–1024, ISBN: 9781450392655.

[99] S. Nagy and M. Hicks, "Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 787–802.

[100] J. Chen, J. Wang, C. Song, and H. Yin, "Jigsaw: Efficient and scalable path constraints fuzzing," in *2022 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, 2022, pp. 1531–1531.

[101] R. Ding, Y. Kim, F. Sang, W. Xu, G. Saileshwar, and T. Kim, "Hardware support to improve fuzzing performance and precision," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2214–2228.

[102] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2597–2614.

[103] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1683–1700.

[104] J. Jung, S. Tong, H. Hu, J. Lim, Y. Jin, and T. Kim, "Winnie: Fuzzing windows applications with harness synthesis and fast cloning," in *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*, 2021.

[105] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 351–365, ISBN: 9781450384544.

[106] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1497–1511.

[107] A. Andronidis and C. Cadar, "Snapfuzz: An efficient fuzzing framework for network applications," *arXiv preprint arXiv:2201.04048*, 2022.

[108] J. Li, S. Li, G. Sun, T. Chen, and H. Yu, "Snpsfuzzer: A fast greybox fuzzer for stateful network protocols using snapshots," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 2673–2687, 2022.

[109] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, "Instrim: Lightweight instrumentation for coverage-guided fuzzing," in *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*, vol. 40, 2018.

[110] C. Zhang, W. Y. Dong, and Y. Z. Ren, "Instrcr: Lightweight instrumentation optimization based on coverage-guided fuzz testing," in *2019 IEEE 2nd International Conference on Computer and Communication Engineering Technology (CCET)*, IEEE, 2019, pp. 74–78.

[111] C. Zhou, M. Wang, J. Liang, Z. Liu, and Y. Jiang, "Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 858–870.

[112] M. Wang *et al.*, "{Riff}: Reduced instruction footprint for {coverage-guided} fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 147–159.

[113] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "{Kafl}:{hardware-assisted} feedback fuzzing for {os} kernels," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 167–182.

[114] Google, *Honggfuzz*, https://google.github.io/honggfuzz/, 2016.

[115] Y. Chen *et al.*, "Ptrix: Efficient hardware-assisted fuzzing for cots binary," in *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, 2019, pp. 633–645.

[116] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" In *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 181–198, ISBN: 978-1-939133-17-5.

[117] S. Poeplau and A. Francillon, "Symqemu: Compilation-based symbolic execution for binaries.," in *NDSS*, 2021.

[118] D. Maier, B. Radtke, and B. Harren, "Unicorefuzz: On the viability of emulation for kernelspace fuzzing," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, 2019.

[119] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019, pp. 754–768.

[120] G. Pan *et al.*, "V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2197–2213.

[121] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun, "Efficient greybox fuzzing of applications in linux-based iot devices via enhanced user-mode emulation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022, pp. 417–428.

[122] J. Liang, Y. Jiang, Y. Chen, M. Wang, C. Zhou, and J. Sun, "Pafl: Extend fuzzing optimizations of single mode to industrial parallel mode," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 809–814.

[123] C. Song, X. Zhou, Q. Yin, X. He, H. Zhang, and K. Lu, "P-fuzz: A parallel grey-box fuzzing framework," *Applied Sciences*, vol. 9, no. 23, p. 5100, 2019.

[124] X. Zhou *et al.*, "Unifuzz: Optimizing distributed fuzzing via dynamic centralized task scheduling," *arXiv preprint arXiv:2009.06124*, 2020.

[125] Y. Chen *et al.*, "{Enfuzz}: Ensemble fuzzing with seed synchronization among diverse fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1967–1983.

[126] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2313–2328.

[127] *Clusterfuzz*, https://github.com/google/clusterfuzz, 2022.

[128] *Clusterfuzzlite*, https://google.github.io/clusterfuzzlite/, 2022.

[129] P. Di, B. Liu, and Y. Gao, "Microfuzz: An efficient fuzzing framework for microservices," *arXiv preprint arXiv:2401.05529*, 2024.

[130] T. Gu *et al.*, "Group-based corpus scheduling for parallel fuzzing," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1521–1532.

[131] X. Zhou *et al.*, "Ultrafuzz: Towards resource-saving in distributed fuzzing," *IEEE Transactions on Software Engineering*, 2022.

[132] E. Güler *et al.*, "Cupid: Automatic fuzzer selection for collaborative fuzzing," in *Annual Computer Security Applications Conference*, 2020, pp. 360–372.

[133] S. Österlund *et al.*, "Collabfuzz: A framework for collaborative fuzzing," in *Proceedings of the 14th European Workshop on Systems Security*, 2021, pp. 1–7.

[134] Y. Chen, T. Su, C. Sun, Z. Su, and J. Zhao, "Coverage-directed differential testing of jvm implementations," in *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 85–99.

[135] Y. Chen, T. Su, and Z. Su, "Deep differential testing of jvm implementations," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 1257–1268.

[136] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on security and privacy (SP)*, IEEE, 2017, pp. 615–632.

[137] M. Cao *et al.*, "Different is good: Detecting the use of uninitialized variables through differential replay," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1883–1897.

[138] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: Differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 176–187.

[139] G. S. Reen and C. Rossow, "Dpifuzz: A differential fuzzing framework to detect dpi elusion strategies for quic," in *Proceedings of the 36th Annual Computer Security Applications Conference*, 2020, pp. 332–344.

[140] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2021, pp. 1286–1303.

[141] X. Zhang *et al.*, "Duo: Differential fuzzing for deep learning operators," *IEEE Transactions on Reliability*, vol. 70, no. 4, pp. 1671–1685, 2021.

[142] P. Li, W. Meng, and K. Lu, "Sediff: Scope-aware differential fuzzing to test internal function models in symbolic execution," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 57–69.

[143] L. Bernhard, T. Scharnowski, M. Schloegel, T. Blazytko, and T. Holz, "Jit-picking: Differential fuzzing of javascript engines," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 351–364.

[144] Y. Liang, S. Liu, and H. Hu, "Detecting logical bugs of {dbms} with coverage-based guidance," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4309–4326.

[145] *Addresssanitizer, threadsanitizer, memorysanitizer*, https://github.com/google/sanitizers, 2024.

[146] E. Trickel *et al.*, "Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities," in *2023 IEEE symposium on security and privacy (SP)*, IEEE, 2023, pp. 2658–2675.

[147] R. Meng, Z. Dong, J. Li, I. Beschastnikh, and A. Roychoudhury, "Linear-time temporal logic guided greybox fuzzing," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1343–1355.

[148] H. Kim, M. O. Ozmen, A. Bianchi, Z. B. Celik, and D. Xu, "Pgfuzz: Policy-guided fuzzing for robotic vehicles.," in *NDSS*, 2021.

[149] M. Rigger and Z. Su, "Testing database engines via pivoted query synthesis," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 667–682.

[150] J. Wang, Z. Zhang, S. Liu, X. Du, and J. Chen, "{Fuzzjit}:{oracle-enhanced} fuzzing for {javascript} engine {jit} compiler," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1865–1882.

[151] S. Kim and T. Kim, "Robofuzz: Fuzzing robotic systems over robot operating system (ros) for finding correctness bugs," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 447–458.

[152] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 259–269.

[153] C. Shou, I. B. Kadron, Q. Su, and T. Bultan, "Corbfuzz: Checking browser security policies with fuzzing," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 215–226.

[154] Wikipedia contributors, *Translator (computing) — Wikipedia, the free encyclopedia*, [Online; accessed 1-September-2020], 2020.

[155] *Memory unsafety problem in safe Rust*, https://github.com/rust-lang/rust/issues/69225, 2020.

[156] *Miscompilation: for range loop reading past slice end*, https://github.com/golang/go/issues/40367, 2020.

[157] *THE STORY OF TWO WINNING PWN2OWN JIT VULNERABILITIES IN MOZILLA FIREFOX*, https://www.thezdi.com/blog/2019/4/18/the-story-of-two-winning-pwn2own-jit-vulnerabilities-in-mozilla-firefox, 2019.

[158] C. Rohlf and Y. Ivnitskiy, "Attacking clientside jit compilers," *Black Hat USA*, 2011.

[159] K. Thompson, "Reflections on Trusting Trust," *Commun. ACM*, vol. 27, no. 8, Aug. 1984.

[160] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing."

[161] T. Blazytko *et al.*, "Grimoire: Synthesizing structure while fuzzing," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19, Santa Clara, CA, USA: USENIX Association, 2019, pp. 1985–2002, ISBN: 9781939133069.

[162] *Domato, A DOM fuzzer*, https://github.com/googleprojectzero/domato, 2017.

[163] *List of programming languages*, https://en.wikipedia.org/wiki/List_of_programming_languages, 2017.

[164] *Bugs in PHP interpreter found by OSS-FUZZ*, https://bugs.chromium.org/p/oss-fuzz/issues/list?q=-status%3AWontFix%2CDuplicate%20-component%3AInfra%20PHP&can=1, 2020.

[165] *Data type*, https://en.wikipedia.org/wiki/Data_type, 2020.

[166] *Gnu bison*, https://www.gnu.org/software/bison/, 2020.

[167] *Flex, the fast lexical analyzer generator*, https://github.com/westes/flex/, 2020.

[168] *Grammars written for ANTLR v4*, https://github.com/antlr/grammars-v4, 2020.

[169] *Tiobe index for august 2020*, https://www.tiobe.com/tiobe-index/, 2020.

[170] *JavaScript: Standard built-in objects*, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects, 2020.

[171] *PHP: Internal (built-in) functions*, https://www.php.net/manual/en/functions.internal.php, 2020.

[172] S. Gan *et al.*, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, IEEE, 2018, pp. 679–696.

[173] Wikipedia contributors, *Mann-whitney u test — Wikipedia, the free encyclopedia*, [Online; accessed 31-August-2020], 2020.

[174] *PHP Sandbox, Test your PHP code with this code tester*, https://sandbox.onlinephpfunctions.com/, 2020.

[175] *Online PHP Sandbox*, https://wtools.io/php-sandbox/, 2020.

[176] *PHP 7.0-7.4 disable_functions bypass*, https://github.com/mm0r1/exploits/tree/master/php7-backtrace-bypass, 2019.

[177] *Dismissed PHP flaw shown to pose code execution risk*, https://portswigger.net/daily-swig/dismissed-php-flaw-shown-to-pose-code-execution-risk, 2019.

[178] *From Web to Pwn - FFI Arbitrary read/write without FFI::cdef or FFI::load*, http://blog.huntergregal.com/2020/07/from-web-to-pwn-ffi-arbitrary-readwrite.html, 2020.

[179] Wikipedia contributors, *Scope (computer science) — Wikipedia, the free encyclopedia*, [Online; accessed 23-December-2020], 2020.

[180] Wikipedia contributors, *Rust (programming language) — Wikipedia, the free encyclopedia*, [Online; accessed 23-December-2020], 2020.

[181] R. Gopinath, B. Mathis, and A. Zeller, "Inferring input grammars from dynamic control flow," *arXiv preprint arXiv:1912.05937*, 2019.

[182] R. Gopinath, B. Mathis, M. Höschele, A. Kampmann, and A. Zeller, "Sample-free learning of input grammars for comprehensive software fuzzing," *ArXiv*, vol. abs/1810.08289, 2018.

[183] Z. Wu *et al.*, "Reinam: Reinforcement learning for input-grammar inference," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 488–498.

[184] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 1044–1051.

[185] P. Godefroid, H. Peleg, and R. Singh, "Learn fuzz: Machine learning for input fuzzing," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 50–59.

[186] C. Myung, G. Lee, and B. Lee, "{Mundofuzz}: Hypervisor fuzzing with statistical coverage testing and grammar inference," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1257–1274.

[187] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3559–3576.

[188] S. T. Dinh *et al.*, "Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases.," in *NDSS*, 2021.

[189] C. Lyu *et al.*, "{Mopt}: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1949–1966.

[190] J. Wang, C. Song, and H. Yin, "Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing," in *2021 Network and Distributed System Security Symposium*, 2021.

[191] G. Zhang *et al.*, "Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing," in *Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022.

[192] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021, pp. 2829–2846, ISBN: 978-1-939133-24-3.

[193] S. Nagy and M. Hicks, "Full-speed Fuzzing: Reducing Fuzzing Overhead Through Coverage-guided Tracing."

[194] *A self-hosted fuzzing-as-a-service platform.* https://github.com/microsoft/onefuzz, 2022.

[195] *Microservice architecture*, https://microservices.io/, 2022.

[196] *Build reliable network applications without compromising speed.* https://tokio.rs/, 2022.

[197] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, Dec. 2020.

[198] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya, "Fuzzbench: An open fuzzer benchmarking platform and service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1393–1403.

[199] T. Klooster, F. Turkmen, G. Broenink, R. t. Hove, and M. Böhme, "Effectiveness and scalability of fuzzing techniques in ci/cd pipelines," *arXiv preprint arXiv:2205.14964*, 2022.

[200] *The source code of readdir system call.* https://elixir.bootlin.com/linux/v6.1.9/source/fs/readdir.c#L54, 2023.

[201] *QuickJS*, https://github.com/ldarren/QuickJS, 2020.

[202] *Important features of afl++*, https://github.com/AFLplusplus/AFLplusplus/blob/stable/docs/features.md, 2023.

[203] A. Fioraldi, D. C. Maier, D. Zhang, and D. Balzarotti, "Libafl: A framework to build modular and reusable fuzzers," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1051–1065.

[204] *Fuzzing beyond memory corruption: Finding broader classes of vulnerabilities automatically*, https://security.googleblog.com/2022/09/fuzzing-beyond-memory-corruption.html, 2022.

[205] Wikipedia contributors, *Assertion (software development) — Wikipedia, the free encyclopedia*, [Online; accessed 1-March-2024], 2023.

[206] O. De Moor *et al.*, "Keynote address:. ql for source code analysis," in *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, IEEE, 2007, pp. 3–16.

[207] Wikipedia contributors, *Unit testing — Wikipedia, the free encyclopedia*, [Online; accessed 29-April-2024], 2024.

[208] Google, *Googletest - google testing and mocking framework*, https://github.com/google/googletest, 2024.

[209] Wikipedia contributors, *Server-side request forgery — Wikipedia, the free encyclopedia*, [Online; accessed 29-April-2024], 2023.

[210] *Common weakness enumeration*, https://cwe.mitre.org/, 2024.

[211] *Semgrep — find bugs and enforce code standards*, https://semgrep.dev/, 2024.

[212] P. Anderson, "The use and limitations of static-analysis tools to improve software quality," *CrossTalk: The Journal of Defense Software Engineering*, vol. 21, no. 6, pp. 18–21, 2008.

[213] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, IEEE, 2017, pp. 8–9.

[214] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme, *Modeling software with finite state machines: a practical approach*. Auerbach Publications, 2006.

[215] *The llvm compiler infrastructure*, https://llvm.org/, 2024.

[216] *Dataflowsanitizer*, https://clang.llvm.org/docs/DataFlowSanitizer.html, 2024.

[217] *Imagemagick – convert, edit, or compose digital images*, https://imagemagick.org/, 2024.

[218] S. Fujita, *Header only c++11 tiny gltf 2.0 library*, https://github.com/syoyo/tinygltf, 2024.

[219] apache, *The apache http server project*, httpd.apache.org, 2024.

[220] GNOME, *A library to render svg images to cairo surfaces.* https://gitlab.gnome.org/GNOME/librsvg, 2024.

[221] *A complete, cross-platform solution to record, convert and stream audio and video.* https://ffmpeg.org/, 2024.

[222] *Multi-format archive and compression library*, https://www.libarchive.org/, 2024.

[223] P. D. Group, *Libpng: Portable network graphics support, official libpng repository*, https://github.com/pnggroup/libpng, 2024.

[224] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "Fuzzfactory: Domain-specific fuzzing with waypoints," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[225] The White House, *Statements of support for software measurability and memory safety*, https://www.whitehouse.gov/oncd/briefing-room/2024/02/26/memory-safety-statements-of-support/, 2024.

[226] *Go fuzzing*, https://go.dev/doc/security/fuzz/, 2024.

[227] *Coverage-guided, in-process fuzzing for node.js*, https://github.com/CodeIntelligenceTesting/jazzer.js, 2024.

[228] *Soot - a java optimization framework*, https://github.com/soot-oss/soot, 2024.

[229] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.