The Pennsylvania State University
The Graduate School

# ADVANCED FUZZING METHODS FOR SOFTWARE SECURITY

A Dissertation in
Information Sciences and Technology
by
Rui Zhong

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2023

# Abstract

Fuzzing is an increasingly popular technique for testing software functionality and identifying security weaknesses. Recent research has primarily focused on improving the effectiveness and efficiency of fuzzing. To make fuzzing more effective, researchers apply different methods, such as diverse initial seeds, varied mutation strategies, and different feedback to help produce better test cases. To make fuzzing more efficient, many research projects design specialized operating systems or parallel fuzzers to speed up the fuzzing process and accelerate bug detection.

However, there are still some limitations in current fuzzing approaches. Firstly, most existing approaches, mutation-based and generation-based, have problems in exploring program states for software that requires structural input, especially when considering syntax and semantics. Mutation-based fuzzers lack an understanding of the test case format, often resulting in broken test cases and wasted time. Generation-based ones do not utilize feedback, such as code coverage, to guide the program state exploration, leading to repetitive testing of the same part of the code. Secondly, while parallel fuzzing is widely adopted to speed up bug detection, existing parallel approaches are built on top of synchronous serial fuzzers and rely on periodic synchronization to enable collaboration among multiple instances. The serial design of the fuzzer might waste CPU power due to blocking I/O operations. Additionally, the synchronization of fuzzing states between multiple instances presents a challenge, as untimely synchronization can result in suboptimal strategies while synchronizing too frequently creates excessive overhead.

To address the first limitation, we turn our attention to language processors, such as compilers and interpreters, as they require inputs following specific syntactic and semantic rules. To effectively fuzz language processors, we are required to not only generate high-quality test cases for them, but also find a unified way to handle the different features, such as syntax and semantics, in multiple languages. As our first step, we focus on Database Management Systems (DBMSs) since they take a domain-specific language, Structure Query Language (SQL), as input. Empirical study shows that the syntactic and semantic accuracy of test cases is crucial for detecting bugs in DBMSs, especially those hidden within complex logic. Therefore, we design a fuzzing framework called SQUIRREL that uses a lightweight IR to generate syntactically correct SQL test cases and utilizes an instantiator to validate the test cases for DBMSs. By conducting experiments with real-world DBMSs, the results demonstrate that SQUIRREL not only produces accurate test cases but also has the ability to detect bugs in deep logic.

Then, we generalize our approaches used for DBMS fuzzing and build a generic language processor fuzzing framework called POLYGLOT. Our framework takes the specification of a language as input and casts it to IR's specification. Test cases are then transformed into IR statements, removing any language differences, followed by mutations and validations. Evaluation of PolyGlot shows that it can generate high-quality inputs to test various language processors and effectively detect bugs.

To address the second limitation, we investigate the current parallel fuzzing architecture and introduce $\mu$Fuzz, a microservice-based fuzzing framework. $\mu$Fuzz breaks the serial fuzzing loops into concurrent services, each with multiple workers, making more efficient use of CPU power. Besides, $\mu$Fuzz eliminates the synchronization of fuzzing states by partitioning the states across different services, allowing for optimal global decision-making.

Our research focuses on enhancing the current fuzzing approaches in both effectiveness and efficiency. We design Squirrel and PolyGlot to improve fuzzing effectiveness by producing high-quality test cases that require structural formats. Moreover, we introduce $\mu$Fuzz as the first attempt to improve fuzzing efficiency through the implementation of microservice architecture. More importantly, we have found 239 newly identified bugs and got 30 CVEs assigned, demonstrating the practicality of our methods in testing real-world software.

# Contents

# List of Figures

# List of Tables

# Acknowledgments

First of all, I would like to begin by expressing my sincere appreciation to my advisor, Professor Dinghao Wu, for his invaluable guidance, unwavering support, and enduring patience throughout my Ph.D. His expertise and wisdom have not only influenced my academic development but also had a profound impact on my entire life. Without his assistance, I would not have been able to achieve this significant milestone.

Second, I also want to express my gratitude to the rest of my committee members, Hong Hu, Taegyu Kim, and Sencun Zhu, for their valuable feedback, insightful comments, and constructive suggestions that helped me to enhance my research from various angles.

I am fortunate to have the support and companionship of my talented friends and labmates, Biqiao Xu, Jinquan Zhang, Zihao Wang, and Zitong Shang. Without their help, I would never overcome the challenges I faced. I would like to show my special appreciation to Yongheng Chen from the Georgia Institute of Technology. The countless hours we spent working together were filled with both challenges and triumphs, and I will always treasure the experience. His support and friendship made my Ph.D. journey truly special.

Last but not least, I am grateful to my family, especially my parents, for their unconditional support and encouragement throughout my academic journey. Their belief in me and my abilities has been my greatest motivation and their sacrifices have made my achievements possible.

This research was supported in part by the Office of Naval Research (ONR) under grants N00014-16-1-2265, N00014-16-1-2912, and N00014-17-1-2894, and the National Science Foundation (NSF) under grant CNS-1652790. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR.

# Chapter 1 Introduction

Security Bugs in software can not only be harmful to users' experience, but also lead to serious consequences, for example, attackers can exploit bugs to gain complete or partial control of user devices or steal sensitive information. To prevent malicious attacks, many organizations and communities, which implement or use public software and libraries, invest much effort in detecting and fixing security bugs. In addition, many techniques are developed to help make software secure, e.g., static analysis, taint analysis and range analysis.

## 1.1 Background

Fuzzing is a technique used in software testing that involves running the target software repeatedly with random inputs. In recent years, fuzzing has been widely adopted as a software-testing technique to detect security issues [40, 73, 144]. Compared to other software testing techniques, fuzzing has a significant advantage: it ensures high throughput while needing less manual effort and pre-knowledge of the target software [138]. Moreover, fuzzing is verified to be practical enough to detect security issues in critical software and libraries. For example, Google implemented OSSFuzz [42] and found more than 36,000 bugs in over 550 popular software since 2016.

Previous research on fuzzing has primarily focused on scaling up the approach by enhancing the internal components, especially the test case generation. Based on how to produce test cases, state-of-the-art fuzzing can be categorized into two categories: generation-based and mutation-based. In generation-based approaches, developers create a formal model that precisely understands the definition of input data [151]. Based on this model, the fuzzer enumerates all possible inputs to test the software's functionality and identify bugs. On the other hand, mutation-based approaches rely on random mutation, like bit flip, to create new test cases from existing test cases. The key to producing an effective test case is making it slightly different from the valid data so it can trigger unexpected behaviors, such as execution crashes and assertion failures, in the software instead of making the data too invalid, which will be discarded quickly. Mutation-based fuzzers also utilize feedback, like code coverage, to guide the exploration of the input space.

To further improve the fuzzers, researchers also set sight on the scale-out approach, e.g., adding more fuzzing instances into the fuzzing system and improving collaborating methods. The scale-out approach such as parallel fuzzing allows us to concentrate the original multi-group fuzzing resources into one group to improve the efficiency of exploration further. It optimizes the use of resources and shortens the overall time required for fuzzing, making it a popular choice in the industry. For instance, Google utilizes a parallel fuzz infrastructure for continuous integration and delivery (CI/CD) testing of newly submitted code. [45]

### 1.1.1 Generation-based fuzzing

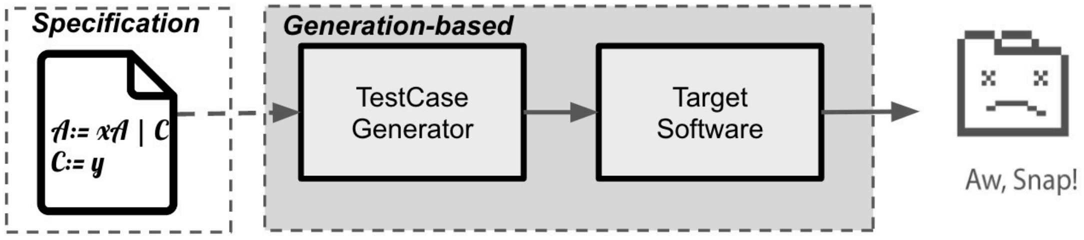


Figure 1.1: The workflow of Generation-based fuzzing.

Figure 1.1 shows the workflow of generation-based fuzzing. The process begins with a specification detailing the file format or protocol, from which test cases are created. Fuzzers then execute these generated test cases on the target software, discovering any bugs that exist.

Previous studies have demonstrated that generation-based fuzzing can produce high-quality test cases for testing complex input-based software, such as compilers and protocols. For example, CSmith [141] is a generation-based fuzzer for C compilers and it found over 400 previously unknown compiler bugs.

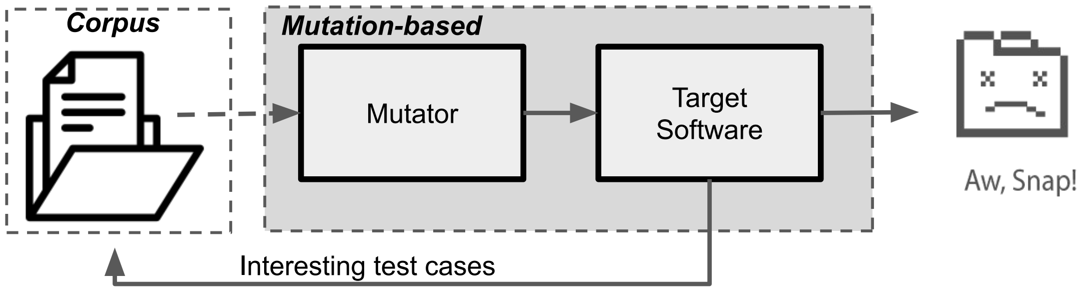


Figure 1.2: The workflow of Mutation-based fuzzing.

### 1.1.2 Mutation-based fuzzing

Figure 1.2 illustrates the process of mutation-based fuzzing. The Fuzzer first selects a test case from the corpus, a collection of test cases. The mutator then applies pre-defined strategies, such as bit flipping, to mutate the test case. Next, the fuzzer executes the mutated test case and gathers feedback from the execution. Finally, it evaluates the test case based on the feedback and updates the corpus accordingly.

The advantage of mutation-based fuzzing is fast and easy to implement. It requires a minimal understanding of the input format of the target software. Although the mutation strategies employed are simple, they can still effectively detect bugs through feedback-based test case evolution. Recent research has shown that the mutation-based fuzzing approach is effective in discovering bugs [108].

### 1.1.3 Parallel fuzzing

Instead of enhancing internal components, parallel fuzzing attempts to concentrate multi-group computational resources into one fuzzer. The state-of-the-art parallel fuzzing techniques involve launching multiple fuzzing instances across separate processes and periodically synchronizing the states between instances by synchronizing the corpus. Each instance follows the same logic as a standalone fuzzer. For example, it can adopt a mutation-based serial fuzzing loop that first takes a test case from the corpus, then mutates it and produce a new test case, and finally runs software with that test case and collects the feedback. Each instance maintains its own local corpus and states, and when it receives test cases from other instances' corpus, it updates its local fuzzing states. This states synchronization allows one instance to catch up on the latest progress from other instances. Thus, all instances make contributions to the program state exploration for detecting bugs.

The advantage of parallel fuzzing is reducing the time cost of program state exploration, i.e., it costs less time to detect bugs. Existing projects show that parallel fuzzing is practical in both industry and academia [19, 43, 54, 92, 144].

## 1.2 Motivation

Fuzzing is an increasingly popular technique to detect software security bugs. Recent research shows that fuzzing is practical for securing software and productions. Therefore, companies and organizations invest significant effort in building their fuzzing infrastructure. For example, Microsoft [80] and Google [43, 45] built their fuzzing infrastructure several years ago.

As fuzzing continues to be widely adopted for identifying security issues, developers and researchers devote themselves to improving its effectiveness and efficiency. For the effectiveness aspect, current fuzzing methods are sufficient for trivial software, but they need to be improved for more complex systems. In particular, software that requires inputs in structured formats presents a significant challenge for effective testing by fuzzing, especially for those further require semantic correctness. The language processor is a typical type of software that requires these features. The language processor can be divided into the domain-specific processor and the general-purpose processor. A domain-specific language processor works with a computer language that is targeted to a particular kind of problem. It can have more features than a general-purpose language in structure and semantics to match the specific purpose of the usages. One such example is Database Management Systems, which are integral components of modern data-intensive systems [55, 77, 85]. Like other complex systems, DBMSs are built with a large codebase prone to security bugs These security bugs can have serious consequences for both functionality and the potential for malicious attacks. Unlike other systems, DBMSs take SQL as input, making it challenging to generate effective test cases. Generally, DBMSs process a query in four continuous phases: parse, validation, optimization, and execution. The parse and validation will check the syntax and semantics, respectively. If any error occurs, the DBMS will immediately terminate the process. Thus, bugs in the optimization and execution phases are considered deeper and more difficult to detect.

The current state-of-the-art fuzzing methods are ineffective for detecting deep bugs in DBMSs. Existing research [106] focuses on using generation-based fuzzing approaches to test DBMSs, and these approaches require developers to provide precise SQL grammar as the specification to fuzzers so the fuzzer can generate syntactically correct test cases.

While these approaches enumerate all possibilities based on the grammar, they fail to effectively detect bugs in DBMSs due to the considerable input space and the limited number of bugs. Different from generation-based approaches, mutation-based fuzzing relies on the random mutation to construct new test cases from existing test cases and utilizes feedback to guide input space exploration. However, it is difficult for mutation-based fuzzing to construct test cases that can pass the syntactic check as it performs random mutations on existing test cases. Further, both mutation-based and generation-based approaches have limited ability to consider semantic correctness since they are unaware of semantics.

General-purpose language processors, such as compilers and interpreters, also face challenges in bug detection. These tools translate high-level programming languages into low-level machine code, and if they contain bugs, they can result in security vulnerabilities by translating correct programs into incorrect code. For instance, a miscompilation in the Rust compiler caused a memory vulnerability in compiled software [24]. Like DBMSs, general-purpose language processors require input that follows the corresponding grammar and performs syntactic and semantic checks during the compilation process. This makes it difficult to detect bugs that occur after the correctness checks have been performed.

Testing general-purpose language processors using existing fuzzing approaches is challenging. Both generic generation-based and mutation-based fuzzing have limitations. Generation-based methods can generate test cases that conform to the grammar but often waste time on grammar enumeration. Mutation-based methods can avoid this problem, but mutated test cases are unlikely to pass the syntactic checks. In addition, existing research on testing language processors has employed specialized analyses to improve test case quality [141]. This makes the fuzzer less generically applicable when it is highly specialized for a single programming language. However, there are over 700 programming languages [130] with different syntax and semantics, making it difficult to use a specialized fuzzer to test a different programming language. Considering the significant engineering work of implementing a language fuzzer(e.g., Fuzzilli [50] consists of 43k LoC), it is unaffordable to make a specific fuzzer for each language. This presents a trade-off between higher generic applicability and better test case quality in language processor fuzzing.

For efficiency, most companies and organizations adopt parallel fuzzing. The state-of-the-art parallel fuzzing runs multiple fuzzing instances of the same fuzzer and performs periodic states synchronization so all instances can progress to the exploration. By utilizing more resources, a parallel fuzzer can significantly reduce testing time compared to a single-instance fuzzer.

However, we found there are two limitations in current parallel fuzzing architecture. First, the CPU can be idle during the fuzzing process because each instance of parallel fuzzing is designed in a serial manner [54, 144]. For example, the typical pipeline of AFL-based fuzzers includes test case selection, test case mutation, execution, feedback evaluation, and looping. During these steps, if I/O blocking occurs, other steps cannot be activated, causing a decrease in fuzzing performance. This was verified through a measurement of the CPU usage of fuzzing `tcpdump` [116] with `AFLPlusPlus` [54], which showed only a 70% utilization rate. After a further investigation using `strace` [29], it was discovered that `tcpdump` was waiting for blocking system calls such as `poll`.

Second, the synchronization between instances is untimely. Although the instance performs a states synchronization periodically, there are still some time windows between 2 synchronizations. During these time windows, the instance is unaware of the latest progress made by other instances. Therefore, it use outdated states to make decision, which can be suboptimal globally. However, too frequent synchronization also brings a problem that introduces a high overhead.

To improve the effectiveness and efficiency of state-of-the-art fuzzing approaches, we make efforts to construct high-quality test cases to fuzz DBMSs. We propose a new fuzzing approach that takes advantage of generation-based and mutation-based approaches. Our method can improve the syntactic and semantic correctness of the SQL test cases and adopt the feedback mechanism to promote the evolution of test cases. We also extend this approach to language processor fuzzing and provide a solution to make fuzzing with both generic applicable and high-quality test cases. Besides, we redesign the parallel fuzzing with the microservice architecture, which can eliminate the CPU idle and allow the fuzzer to make optimal strategies. With our new design, parallel fuzzing can make better use of CPU power than start-of-the-art approaches to make the exploration.

## 1.3 Research Goals

In this section, we summarize our research goals and then present an overview of three research studies in this dissertation. We present our research in three parts:

1. Testing Database Management Systems with Language Validity and Coverage Feedback.

2. Generic Language Processor Testing with Semantic Validation.
3. Redesign Parallel Fuzzing using Microservice Architecture.

Our goal is to improve fuzzing in both scale-up and scale-out aspects. For the scale-up, we attempt to improve certain internal components of current fuzzing and make it more effective in detecting bugs. Although current fuzzing is practical enough for most of the software, we still find that testing software which requires a structural input is challenging, especially for those further require input in specific syntax and semantics, such as language processors. As shown in the first project, our first step is related to Database Management Systems. In this project, we investigate the challenges of testing DBMSs and provide our solution for generating high-quality test cases during DBMS fuzzing. Following by the DBMSs fuzzing, we expand our approach to general-purpose language processor fuzzing, achieving both generic applicability and test case quality. After addressing the weakness in language processor fuzzing, we focus on the scale-out methods, which can be orthogonal to improving components. We attempt to enhance the performance of parallel fuzzing by redesigning its architecture using a microservice architecture, which eliminates the issue of CPU idleness and relieves the untimely synchronization of states.

### 1.3.1 Testing Database Management Systems with Language Validity and Coverage Feedback

The prevalence of Database Management Systems (DBMSs) in our data-driven world has made their robustness a top priority. Any bugs present in these complex systems can not only disrupt functionality but also create security bugs leading to serious consequences such as financial loss and data theft.

Fuzzing is widely adopted to assure the quality of DBMSs. However, state-of-the-art mutation-based fuzzing struggles to produce syntactically correct test cases, while generation-based fuzzing cannot explore states effectively. Furthermore, neither approach can guarantee semantic correctness. Our goal is to address these challenges and find an effective way to fuzz DBMSs.

**Adopt feedback mechanism to guide test cases evolution.** The feedback mechanism, which utilizes metrics such as coverage, helps in evaluating the usefulness of a test case. If the test case does not produce any new results in coverage, it is deemed uninteresting and should be discarded. Since generation-based fuzzing does not incorporate mutations, it is incompatible with feedback mechanisms. Therefore, our fuzzing approach will be based on mutation-based techniques to enable a feedback mechanism.

**Improve syntactic correctness.** DBMSs perform syntactic checks in parsing and filter out incorrect SQLs. Our aim is to find deeper bugs in DBMSs, not just the ones caused by incorrect SQLs that fail syntactic checks. To achieve this, we focus on generating syntactically correct test cases that can pass the parsing checks. We use a set of intermediate representations (IR) to help us in this endeavor. The process of converting the SQL test cases into IR involves translating them into Abstract Syntax Trees (ASTs) and then lifting them to IR. Further mutations are performed on the IRs while preserving the syntactic structure to improve the syntax correctness.

**Improve semantic correctness.** Semantics describes the processes a DBMS follows when executing a SQL. It is fragile, and any slight violation can result in errors during executing SQLs, e.g., use a non-existing table name. For this reason, generating semantic correct test cases is challenging. However, passing the semantic checks is crucial if we want to detect deep bugs. Our approach uses a process called semantics-guided instantiation to improve the semantic correctness of test cases. In the semantics-guided instantiation, we analyze the data dependencies and reconstruct the semantics accordingly. After this step, the test cases will likely be valid for a DBMS.

### 1.3.2 Generic Language Processor Testing with Semantic Validation

Language processors are considered as a foundational component of modern software. Buggy language processors can translate even correct source codes to malfunctional machine codes, which can cause severe consequences such as memory corruption. For this reason, the security of language processors is important.

However, testing language processors automatically is nontrivial as they check the validity of the input test cases. Existing specialized fuzzers are efficient in producing high-quality test cases for a specific language processor, but they cannot be easily adapted for other language processors. On the other hand, generic fuzzers are versatile but often struggle to generate effective test cases. Under this dilemma, our goal is to develop a language processor fuzzer that has both generic applicability and the ability to generate effective test cases without sacrificing one for the other one.

**Achieve generic applicability.** Generic applicability makes fuzzers can be easily implemented to different language processors. However, current fuzzers are highly specialized for specific language processors as they want to generate high-quality test cases. Our goal is to find an approach that can neutralize differences in the different programming languages. To achieve this, we define an IR which can carry necessary syntactic and semantic properties. With a frontend generator, all programming languages can be cast to our IR. Further operations will be performed on the IR, so that generic applicability can be achieved.

**Improve semantic correctness.** Semantic correctness is crucial in fuzzing language processors as most of the engineering work is behind the validity checks. Researchers attempt to specialize their fuzzers for higher semantic correctness [50, 84, 141]. Our goal is to increase the semantic accuracy of language processor fuzzing through the implementation of a type system and scope system for our Intermediate Representation (IR). The type system is designed to perform type inference based on semantic properties, while the scope system helps determine the lifetime of variables. With these systems in place, the mutated test cases can be validated, reducing semantic errors.

### 1.3.3 Redesign Parallel Fuzzing using Microservice Architecture

The use of parallelism in bug detection can greatly reduce the time cost by utilizing all available resources. In theory, with more resources, the detection time can be reduced. However, current parallel fuzzing models face limitations such as the problem of CPU idleness during blocking I/O operations, which can negatively impact performance and lead to low CPU utilization. Additionally, untimely synchronization of states can result in suboptimal test case selection from the corpus. Our goal is to address these issues through the implementation of a microservice architecture.

**Microservice architecture.** Microservice architecture organizes applications in a set of collaborating services. These services can run concurrently and dynamically switch when a service gets blocked. As each service is self-contained (i.e., it does not rely on other services.), there is no need for synchronization between services, and the capability can be scaled by employing multiple workers for each service. This design aims to eliminate the limitations present in the current parallel fuzzing model.

**Avoid CPU idling.** Blocking I/O, which can be caused by system calls or socket operations, is a common cause of CPU idling. In the traditional parallel fuzzing approach, the blocking significantly impacts CPU utilization and slows down performance as the fuzzing loop is serial and nothing can be done during the blocking time. However, by implementing a microservice architecture, the serial components can run concurrently, allowing for continuous progress even during a blocking event. The fuzzing loop is partitioned into four services: Corpus management, test case generation, test case execution, and feedback collection, with dependencies still present (e.g., execution relies on generation outputs). To overcome this, we use output caching to decouple the services and allow them to run concurrently. When a service becomes stuck, other services can continue making progress and cache their output. Once the stuck service resumes, it can use the cached output, eliminating the need to wait for generation and maximizing CPU utilization.

**Avoid untimely states synchronization.** In state-of-the-art parallel fuzzing, states are synchronized through sharing the corpus, but during the time window between two synchronizations, an instance may make suboptimal decisions in selecting test cases. Our goal is to relieve the untimely synchronization with microservice architecture. With microservice architecture, states are divided into services and then further partitioned into workers. This allows for concurrent progress and direct updates to each worker's states, ensuring that decisions are always made with the latest information, and optimal strategies are chosen globally.

## 1.4 Thesis Organization

In this thesis, we aim to improve both the effectiveness and efficiency of fuzzing. In particular, we attempt to improve the quality of test cases and make it easier to implement a fuzzer for databases management systems and language processors. Additionally, we redesign the parallel fuzzing model and address several limitations to make it more efficient. The first work presents an novel approach to test DBMSs, which can achieve both high syntactic and semantic correctness in generating test cases. In the second work, we propose a versatile fuzzing framework for all language processors with high language validity. The third work takes advantage of microservice architecture to redesign the current parallel fuzzing model.

The rest of the thesis are organized as follows. We present related works of of this thesis in Chapter 2. In Chapter 3, we outline our novel approach for testing Database Management Systems. In Chapter 4, we investigate the difficulties in testing language processors and present a comprehensive solution. Chapter 5 describes the implementation of the parallel fuzzing model based on microservice architecture. Chapter 6 discusses current limitations and proposes potential improvements. In Chapter 7, we give out a conclusion of the entire thesis.

# Chapter 2 Related Work

In this chapter, we present recent research related to this thesis. First, we go over several research of improving the strategies of fuzzing. We will discuss the generic fuzzing and domain-specific fuzzing approaches. Then we present the research of improving the efficiency of fuzzing.

## 2.1 Fuzzing Effectiveness Improvement

Improving the fuzzing effectiveness focuses on enhancing the internal components of a fuzzer, which can include test case generation, feedback, and seed scheduling. There are mainly two types to test case generation in fuzzing: generation-based fuzzing [39, 84, 141, 142] and mutation-based fuzzing [76, 136, 144].

### 2.1.1 Generation-based Fuzzing

Generation-based fuzzing focuses on testing software that consumes structural inputs [59, 84, 95, 106, 120]. They typically utilize a model that describes the format of the inputs to generate structural inputs that can reach deeper logic of the software. SQLSmith [106] uses the SQL grammar and database schemes to generate more valid queries. MoWF [95] leverages the file format information to fuzz the deeper program code beyond the parser. Considering the infinite input space, blindly generating test cases is inefficient for exploring program states. For this reason, some researchers attempt to utilize feedback to guide test case generation. Apollo [60] measures the difference in execution time to favor generated SQL queries.

For the software which further requires semantic correctness of input, researchers propose several approaches to improve the semantic correctness so they can find deep bugs.

CodeAlchemist [52] proposes semantics-aware assembly to synthesize semantics-correct javascript test cases. CSmith [141] specializes its analysis for C semantics and produces completely correct test cases. Dewey et al. uses constraint logic programming to specify syntactic features and semantic behaviors in test case generation [22, 23], which relies on symbolic executions and complex constraint programming.

### 2.1.2 Mutation-based Fuzzing

Mutation-based fuzzing differs from generation-based fuzzing in that it performs mutation on existing test cases to generate new ones. In this way, the fuzzer can use various feedback information collected from the execution phase to guide its test case generation. AFL [144] uses edge coverage to model program states to guide its mutation, which is highly effective. The mutation strategy, feedback quality, and seed scheduling algorithm can all influence the performance of mutation-based fuzzers. Adopting the methodology from generation-based fuzzers, some language processor fuzzers [3] utilize language grammar to perform constrained mutation. Other fuzzers [14, 114, 143] use symbolic execution or concolic execution to get through complex program conditions. T-Fuzz [94] further proposes a way to dynamically transform the program in order to remove certain checks that are hard for the fuzzer to bypass successfully. To improve feedback quality, researchers try to find better models for the program states. CollAFL [35] provides more accurate coverage information by mitigating path collisions in AFL. Some fuzzers [5, 8, 16, 34, 37, 101] use taint analysis to incorporate data flow information into their coverage metrics. PATA [70] further proposes a path-aware taint analysis by distinguishing between multiple occurrences of the same constraint. The learning-enabled fuzzer Neuzz [109] leverages a surrogate neural network to smoothly approximate the branching behavior of the program in order to generate useful test cases.

Some research attempt add constraints to feedback, so they can guide the priority of fuzzing direction. AFLGo [10] introduces directed greybox fuzzing with the objective of reaching a given set of target program locations efficiently. TortoiseFuzz [128] considers the security impacts indicated by code coverage to prioritize inputs. TaintScope [127] uses checksum as feedback guidance to help fuzz file segments. SAVIOR [18] prioritizes its concolic execution towards the locations with potential vulnerabilities. Ijon [4] annotates the data that represent the internal program states to guide the fuzzer.

Another way is to improve the seed scheduling algorithm [110, 147]. AFLFast [12], MOpt [11], DigFuzz [148] collect information about the test cases and prioritize those with higher potential to reach new code regions.

## 2.2 Fuzzing Efficiency Improvement

Improving fuzzing efficiency aims to make a better use of computational resources, e.g., CPU and memory. Recent research focus on speeding fuzzing up, which allows fuzzers to run more executions in the same amount of time with the same fuzzing strategy [15, 26, 57, 61, 88, 89, 104, 126]. There are two typical strategies to improve the efficiency for fuzzing. Some projects try to speed up the single fuzzing instance by applying optimizations on it, while other researchers focus on making progress on the parallel fuzzing to enhance the collaboration between instances.

### 2.2.1 Improving Efficiency of Single Fuzzer

Improving on the speed of fuzzing is orthogonal to the fuzzing strategy. Various techniques [25, 88, 90, 126] have been proposed to improve the instrumentation of the target program to reduce its overhead. UnTracer [88] proposes coverage-guided tracing to trace block coverage only when new ones are discovered. Nagy et al. [90] further extend the idea of coverage-guided tracing to support edge coverage recording. Odin [126] adopts dynamic recompilation to prune necessary instrumentation on the fly. RetroWrite [25] 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 [105], Honggfuzz [40], and PTrix [19] 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 [143] implements a symbolic execution engine tailed 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 [97] 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 [98] 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.

### 2.2.2 Improving Efficiency on Parallel Fuzzing

Existing works improve the performance of parallel fuzzing also by either improving the fuzzing strategy [20,69,96,111,129,150] or improving the fuzzing speed [139]. One popular way to improve the fuzzing strategy is task partitioning. PAFL [69] proposes an efficient guiding information synchronization method and statically divides fuzzing tasks based on branching information to reduce the overlap between instances. AFLEdge [129] further utilizes static analysis to dynamically create mutually exclusive and evenly weighted fuzzing tasks. Another way to improve the fuzzing strategy is to combine the capabilities of different fuzzers, which is also called ensemble fuzzing [20] or collaborative fuzzing [51]. 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 [20] designs three heuristics for evaluating the diversity of existing fuzzers and choosing the most diverse subset to perform ensemble fuzzing through efficient seed synchronization. Cupid [51] 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. [139] 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 [40, 54, 73] support persistent fuzzing mode, which reuses the same process for multiple test cases to reduce the overhead of forking. Moreover, in-memory test cases [54] are also adopted to reduce the I/O overhead and file system contention.

# Chapter 3 Testing Database Management Systems with Language Validity and Coverage Feedback

In this chapter, we propose Squirrel, a novel fuzzing framework that considers both language validity and coverage feedback to test DBMSs. We design an intermediate representation (IR) to maintain SQL queries in a structural and informative manner. To generate syntactically correct queries, we perform type-based mutations on IR, including statement insertion, deletion and replacement. To mitigate semantic errors, we analyze each IR to identify the logical dependencies between arguments, and generate queries that satisfy these dependencies. We also conduct a set of experiments to show Squirrel has ability to effectively detect deep bugs in DBMSs.

## 3.1 Introduction

Database Management Systems (DBMSs) play a crucial role in modern data-intensive systems [30, 65, 86, 100, 112, 115]. However, these systems are prone to bugs that not only impact their functionality but also provide an opening for malicious attacks. One particularly dangerous type of bug is the memory error, which can result in the memory leakage or corruption in running DBMS processes, potentially leading to remote code execution [21,64], database breaches [32,113], or denial-of-service (DoS) attacks [1,13]. For example, the massive "Collection #1" data breach that exposed 773 million email addresses and 21 billion passwords [58] highlights the devastating impact that memory error bugs. This is of great concern, especially considering the high availability demands of DBMSs, which can exacerbate the effects of memory error bugs, including the weaponization of non-exploitable bugs to launch DoS attacks against centralized data services like e-commerce websites.

The traditional approach to finding bugs in Database Management Systems (DBMSs) is generation-based fuzzing [106]. These techniques require a formal model that accurately represents SQL (Structured Query Language). The testing tools then generate all possible SQL queries based on this model to verify the functionality of DBMSs or detect bugs Unfortunately, this method has limited effectiveness due to the even distribution of effort across all SQL queries. Considering the infinite input space and rare bug-triggering queries, this brute-force style enumeration approach is ineffective in detecting memory error bugs in DBMSs.

In recent years, mutation-based fuzzing is a widely adopted technique for detecting memory error vulnerabilities in software [40, 73, 81, 144]. Unlike generation-based fuzzing techniques, which require a formal model to generate test cases, mutation-based fuzzing uses random mutation and feedback (e.g., code coverage) to explore the input space. The mutation-based fuzzer starts with a seed corpus and randomly mutates existing inputs, like flipping bits, to create new variants. It runs the target program with these inputs and identifies abnormal behaviors like crashes and assertion failures. The fuzzer also records the code path information and prioritizes inputs that trigger new code paths for the next round of mutation. With a large amount of effort spent on improving the fuzzing efficiency [42, 88, 140] and effectiveness [17, 36, 68, 94], mutation-based approaches have successfully found thousands of bugs from popular applications [108].

However, it is challenging to apply mutation-based techniques to test DBMSs, as DBMSs perform two correctness checks, the *syntactic check* and the *semantic check* before executing an SQL query. Specifically, the DBMS first parses each SQL query to get its syntactic meaning. If the query has any grammar errors, the DBMS will stop the execution and immediately bail out with an error message. Otherwise, the DBMS further checks the query for semantic errors, like using a non-existent table, and will bail out in any case of semantic errors. After these two checks, the DBMS creates several execution plans and picks the most efficient one to execute the query. Therefore, to reach the deep logic of a DBMS, the query should be correct syntactically and semantically.

The current random byte mutation techniques used by fuzzing have difficulty in generating syntax-correct or semantically-correct inputs. For instance, `AFL` [144], a widely used mutation-based fuzzer, is able to produce 20 million queries for `SQLite` [55] within 24 hours, but only a small portion of these queries were syntax-correct (around 30%) and semantically correct (around 4%). However, the majority of DBMS code is responsible for tasks such as query plan construction, optimization, and execution, with only a small portion dedicated to syntax and semantic checks. In fact, 20,000 semantically-correct queries generated by `AFL` triggered 19,291 code branches in `SQLite`, while the same number of syntax-incorrect queries only covered 50.8% (9,809 branches) of the former. Therefore, current mutation-based fuzzing has limitations in triggering the complex logic of DBMSs and thoroughly exploring program states.

In this chapter, we introduce a system, Squirrel, to address these challenges so that we can effectively fuzz DBMSs. The system implements two core techniques: *syntax-preserving mutation* and *semantics-guided instantiation*. To ensure the generated SQL queries are syntax-correct, an intermediate representation (IR) is utilized to maintain the query structure and information. Each IR statement contains at most two operands, and each operand is also another IR statement. Each IR has a structure type that indicates the syntactic structure (e.g., `SELECT a FROM b`), and data types (e.g., table name). Before the mutation, our system strips concrete data from the IR and only keeps a skeleton of operations. Then, we perform three random mutations, including inserting type-matched operands, deleting optional operands or replacing operands with other type-matched ones. The type-based mutation ensures the generated query has the correct syntax. Next, the system analyzes the expected dependencies between different IR data, such as the data in a `SELECT` clause being a column of the table in the `FROM` clause. The stripped IR is filled with concrete data that satisfies all expected dependencies, and translated back into SQL and fed to the DBMS for testing. Squirrel combines the benefits of the mutation-based fuzzing (i.e., guided exploration) and the generation-based fuzzing (i.e., high language correctness), and thus can trigger the deep logic of DBMSs and find severe bugs.

The implementation of Squirrel consists of 43,783 lines of C++ code, primarily focused on the integration of the syntax-preserving mutation and semantics-guided instantiation. The code for coverage collection and input prioritization has been adapted from `AFL`. The generic design of Squirrel can be utilized with other fuzzers with some additional engineering efforts.

To understand the effectiveness of our system, we use Squirrel to test four popular DBMSs: `SQLite`, `MySQL`, `PostgreSQL` and `MariaDB`. Squirrel successfully found 63 memory error issues within 40 days, including 51 bugs in `SQLite`, 7 bugs in `MySQL` and 5 bugs in `MariaDB`. As a comparison, Google OSS-Fuzz detected 19 bugs from `SQLite` in 40 months and 15 bugs from `MySQL` in 5 months [42]. We have responsibly reported all of these bugs to the DBMS developers and received positive feedback. At the time of

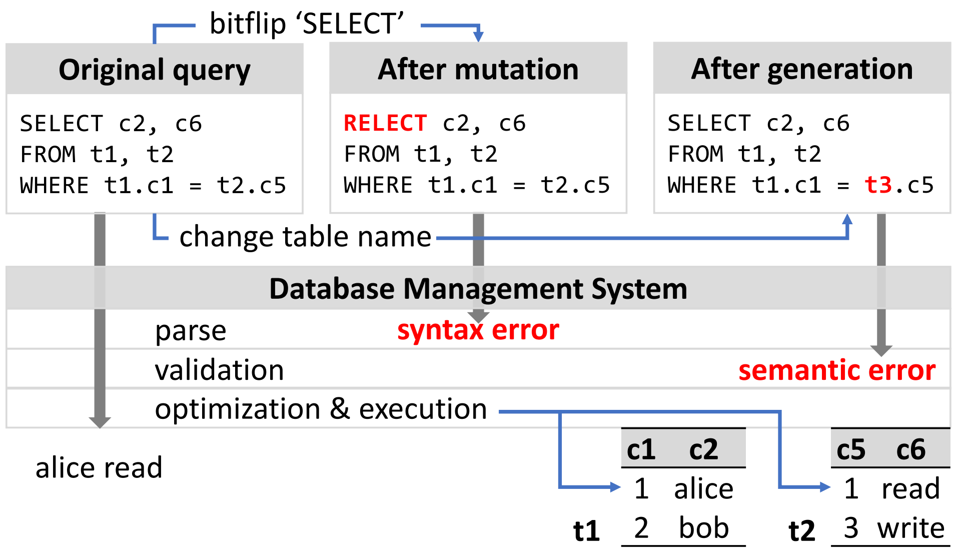


Figure 3.1: **Challenges of testing DBMSs.** A DBMS takes four steps to process one SQL query. Among them, *parse* checks syntactic correctness, and *validation* examines semantic validity. Random mutation unlikely guarantees the syntactic correctness, while grammar-based generation may fail to enforce semantic correctness.

this thesis writing, all bugs have been fixed. We even get CVE numbers for 12 bugs due to their severe security consequences, like stealing database contents.

We inspect various aspects of fuzzing, and compare Squirrel with other state-of-the-art tools, including the mutation-based fuzzer `AFL` and `Angora`, the generation-based tool `SQLsmith`, the structural fuzzer `GRIMOIRE` and the hybrid fuzzer `QSYM`. During the 24-hour testing, Squirrel successfully finds nine unique bugs while others detect one or zero bugs. Squirrel discovers 2.0×-10.9× more new edges than mutation-based tools, and achieves a comparable result to the generation-based tester `SQLsmith`. It also gets 2.4×-243.9× higher semantic correctness than other tools. These results demonstrate that Squirrel is a highly effective tool for detecting memory error issues and outperforms existing generation-based and mutation-based tools.

## 3.2 Problem Definition

In this section, we first briefly describe how a DBMS handles SQL queries. Then, we introduce existing DBMS testing techniques and illustrate their limitations in finding bugs hidden in the deep logic. Finally, we present our insight to solve this problem.

### 3.2.1 Query Processing in DBMS

The processing of SQL queries by modern DBMSs involves four phases: parsing, validation, optimization, and execution [79], as shown in Figure 3.1. The DBMS starts by parsing the SQL query to determine its syntactic meaning. The parser breaks the query into individual tokens and verifies that they conform to the grammar rules. If any syntactic error is found, the DBMS will immediately terminate the execution and return an error message to the client. Next, the validation phase checks the semantic correctness of the query, such as the existence of tables in the database or the unambiguity of columns. Most semantic errors can be detected in this phase. The query optimizer then constructs different possible query plans and determines the most efficient one for execution in the optimization phase. Finally, the chosen plan is executed on the database, and the result is sent back to the client. The execution will reach the second phase if the query is *syntactically* correct and will move to the last two phases only if the query is *semantically* correct.

**Motivating Example.** The "Original query" in Figure 3.1 first joins two tables `t1` and `t2`, and searches for the rows where the `c1` column of `t1` is the same as the `c5` column of `t2`. For each matched row, the query returns the value of `c2` and `c6`. The DBMS finds that this query passes the syntactic check and the semantic check. It searches in the database and finally returns "alice read".

### 3.2.2 Challenges of DBMS Testing

The two main methods of generating SQL queries for testing DBMSs are generation-based and mutation-based approaches Generation-based approaches construct syntactically correct inputs using a precise grammar-model. For example, `SQLsmith` [106], a popular DBMS testing tool, generates syntax-correct test cases from the abstract syntax tree (AST) [125] directly. However, without any guidance, this method cannot efficiently explore the program's state space, as many queries are handled in a similar way by DBMSs. Additionally, the generation-based approach can hardly guarantee the semantic correctness [52], and queries with incorrect semantics will be filtered out by the DBMS during the validation. Figure 3.1 shows a query constructed by a generator(After generation). Although this query is syntactically correct, it cannot be executed because the table `t3` in the `WHERE` clause does not exist in the current database.

Mutation-based fuzzer modifies existing inputs to generate new ones. The effectiveness of this approach is improved by utilizing feedback from previous executions to evaluate the priority of the new inputs. If the feedback indicates that a particular input is of interest, such as triggering a new execution path, the fuzzer will prioritize it for further mutation. In this way, fuzzers will collect more and more interesting test cases and thus can explore the program's state space efficiently. Statistics show that random mutation with feedback-driven works well in many software. For instance, Google's feedback-driven mutation-based fuzzer [73, 108] has discovered over 5,000 vulnerabilities. However, this method is ineffective in handling structured inputs like SQL or JavaScript [125] as random mutations, unaware of grammar, may result in a syntactically incorrect input. An example of this is shown in the Figure 3.1 where flipping a bit in the SQL keyword `SELECT` leads to an invalid keyword `RELECT`, causing the DBMS to reject the query in the parsing phase.

We design evaluation to understand the quality of `AFL`-generated SQL queries, and the importance of syntax-correctness and semantics-correctness. Specifically, we use `AFL` to test `SQLite` for 24 hours, which generates 20 million queries. However, only about 30% of them are syntactically correct, and merely 4% for them can pass semantic checks. We randomly pick 20,000 semantics-correct queries, and find that they trigger 19,291 distinct code branches in `SQLite`. The same number of syntax-incorrect and semantics-incorrect queries only reach 9,809 and 12,811 branches, respectively. These results show the low validation rate of `AFL`-generated queries, and the importance of semantics-correctness for exploring the program state space.

### 3.2.3 Our Approach

Our idea is incorporating both syntax-correctness and semantics-awareness into mutation-based fuzzing to take advantage of the strengths of both generation-based and mutation-based techniques for testing DBMSs.

**Generating Syntax-Correct Queries.** We design a new intermediate representation (IR) to maintain SQL queries in a structural and informative way and adopt type-based mutations to guarantee the syntactic correctness. Each IR statement simply contains at most two operands, and therefore our mutation just has to handle two values. Each statement has an associated grammar type, like `SelectStmt` for `SELECT` statement, while each data has a semantic type, like table name. Our mutation performs type-based operations, including inserting type-matched operands, deleting optional operands and replacing operands with type-matched ones. We strip the concrete data from each IR, like table names, to focus on mutating the skeleton. The IR-based mutation effectively

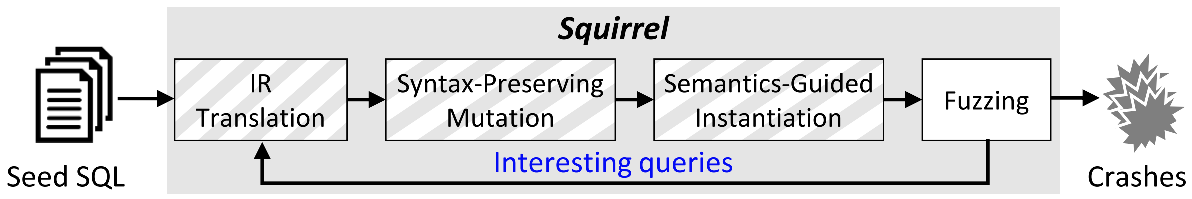


Figure 3.2: **Overview of Squirrel.** SQUIRREL aims to find queries that crash the DBMS. SQUIRREL first lifts queries from SQL to IR; then, it mutates IR to generate new skeletons; next, it fills the skeleton with concrete operands; finally, it runs the new query and detects bugs.

preserves the syntactic correctness. Some generation-based tools generate SQL queries from the AST. However, due to the strict type-constraint and complicated operations, mutating AST is as challenging as modifying SQL queries.

**Improving Semantic Correctness.** Since ensuring the semantic correctness of generated SQL queries is proved to be NP-hard [74], we will try practical solutions to improve the semantic correctness as much as possible. Existing generation-based tools define a set of query templates. Each template represents a complete query and contains specific, static constraints between operands [7]. However, due to the limited human effort, these frameworks cannot guarantee the expressiveness of their SQL templates. We tackle this problem through dynamic *query instantiation*. Given the skeleton of a syntax-correct SQL query (i.e., without concrete operands), our method first builds its data dependency graph according to predefined basic rules. For example, the operand of `SELECT` can be a column name of the table used in `FROM`. Then, we try to fill the skeleton with concrete operands whose relations satisfy the data dependency graph. With the instantiation, the semantic correctness rate is high enough for testing DBMSs.

## 3.3 Overview of Squirrel

Figure 3.2 shows an overview of our DBMS testing framework, SQUIRREL. Given a set of normal SQL queries, SQUIRREL aims to find queries that render the execution of DBMSs crash. A query means a test case and may contain multiple SQL statements. SQUIRREL starts with an empty database and requires the query to create the content. SQUIRREL achieves its goal with four key components: Translator, Mutator, Instantiator, and SQL Fuzzer. First, SQUIRREL selects one query $I$ from a queue that consists of both initial queries and saved interesting queries. Second, the Translator translates $I$ into a vector of IRs $V$. Meanwhile, the Translator strips the concrete values from $V$ to make it a

```
// l: left child, r: right child, d: data, t: data type
V1 = (Column,     l=0,  r=0,  op=0, d="c2", t=ColumnName);
V2 = (ColumnRef,  l=V1, r=0,  op=0, d=0);
V3 = (Expr,       l=V2, r=0,  op=0, d=0);
V4 = (Column,     l=0,  r=0,  op=0, d="c6", t=ColumnName);
V5 = (ColumnRef,  l=V4, r=0,  op=0, d=0);
V6 = (Expr,       l=V5, r=0,  op=0, d=0);
V7 = (SelectList, l=V3, r=V6, op=0, d=0);
// the optional left child can be DINSTRICT
V8 = (SelectClause, l=0, r=V6, op.prefix="SELECT", d=0);
...
//Unknown type for intermediate IRs
Va = (Unknown,    l=V8, r=V14, op=0, d=0);
Vb = (Unknown,    l=Va, r=V25, op=0, d=0);
// the optional right child can be an ORDER clause
V26 = (SelectStmt, l=Vb, r=0,  op=0, d=0);
```

Figure 3.3: **IR of the running example SQL query.** The corresponding AST tree is shown in Figure 3.4.

query skeleton. Our Mutator modifies $V$ through insertion, deletion and replacement to produce a new IR vector $V'$ — $V'$ is syntactically correct. Next, our Instantiator performs data dependency analysis of $V'$ and builds a data dependency graph. Then, the Instantiator selects new concrete values that satisfy the data dependency and fills $V'$ with these values. Since the data dependency is satisfied, $V'$ is likely to be semantically correct. Finally, we convert $V'$ back to a SQL query $I'$ and run the DBMS with $I'$. If the execution crashes, we find an input that triggers a bug. Otherwise, if $I'$ triggers a new execution path of the program, we save it into the queue for further mutation.

## 3.3.1 Intermediate Representation

We design an intermediate representation (IR) of SQL to facilitate syntax-correct mutation of queries. This involves converting SQL queries into the IR format, making mutations to the IR representation, and then converting the mutated IR back to a new SQL query for execution. Our design of the IR aims to achieve three goals: the IRs can represent any SQL statements (**expressive**); the format and operation of IRs are uniform (**general**); the translation between IR and SQL is efficient (**simple**).

The IR is in the static single assignment (SSA) form. A query, or a test case, contains one or more IR statements. Each statement is an assignment, where the left-hand side is the destination variable and the right-hand side is either a literal or an operator with operands. We add the following fields in IR to store the necessary information.

- `ir_type`: the type of one IR statement. This type is based on the corresponding node in the AST, like `column` type for column names or `expr` type for expressions. We also define a special type `Unknown` to represent intermediate statements that have no corresponding node in the AST.
- `operator`: consisting of SQL keywords [121] and mathematical operators [122]. It indicates the operation the IR performs and includes three parts: the prefix `op_prefix`, the interfix `op_mid` and the suffix `op_suffix`. For example, the IR of "`CREATE trigger BEGIN list END`" has prefix `CREATE`, interfix `BEGIN` and suffix `END`.
- `left_operand, right_operand`: the operands of the IR operator. The operand is either another IR statement, or can be `NULL` if the operand is optional or not required.
- `data_value`: the concrete data the IR carries, like table name `t1`.
- `data_type`: data type, like `ColumnName` for column names.

Figure 3.3 shows the IRs of our motivating example in Figure 3.1 (Original query). The corresponding AST is given in Figure 3.4. `V1` and `V4` represent the column names `c2` and `c6`, which are corresponding to nodes ① and ④ in Figure 3.4. They do not contain any operator or operand but have the `ColumnName` data type and proper data values. `V2` and `V5` define references to columns (`V1` and `V4`), and `V3` and `V6` create two expressions. Each of them only has one operand. `V7` describes the parameter list of `SELECT`, including `c2` and `c6`. `V8` represents the `SELECT` clause, which could have `DISTINCT` as its left operand (`NULL` here). `SELECT` appears before the left operand, so it is the operator `prefix`. Since our IR only allows at most two operands, we have to use two intermediate nodes, `Va` and `Vb`, to connect three nodes ⑧, ⑭ and ㉕ to construct the IR of `SelectStmt`. Their `ir_type`s are set to `Unknown`. Finally, `V26` defines the `SELECT` statement, which is node ㉖ in Figure 3.4.

Our IR is just a sequence of assignment statements. This linear representation, different from tree or graph structures (like AST), helps developers to adopt unified and simple mutation strategies. We can perform statement insertion, deletion and replacement while keeping the syntactic correctness.

## 3.4 Syntax-Preserving Mutation

We classify tokens in an SQL query into two groups based on their functionalities. SQL keywords and mathematical operators define what operations to be performed, and we

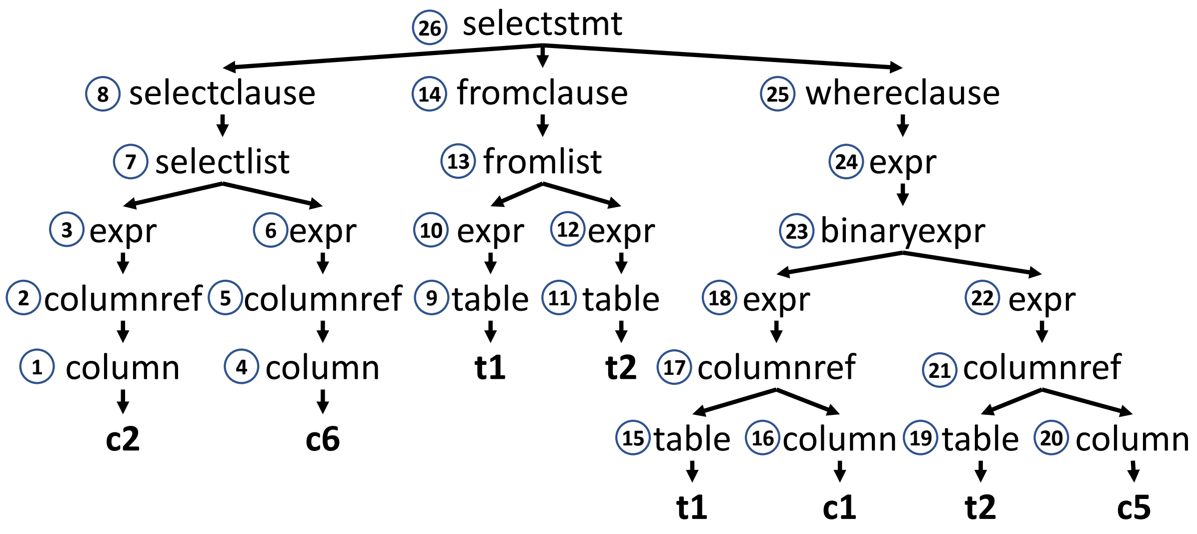


Figure 3.4: **AST of the running example.** SQUIRREL parses the SQL query and represents it in AST, and finally translate AST to IR.

call these tokens *structure*. Other tokens specify the targets of defined operations, and we call them *data*. Data can be literal that makes basic sense, like a constant value `1`, or can express semantic meaning, like table names.

We observe that changing structure tokens has more impact on the DBMS execution than that of changing data tokens. The difference comes from two reasons. First, altering structure will change the operations of the query, and thus trigger different functions, while a DBMS may use the same logic to handle different literal data. For example, `SQLite` takes almost the same path to process query `A:"SELECT c FROM t WHERE c=1"` and query `B:"SELECT c FROM t WHERE c=10"`, but uses significantly different code to handle `C:"SELECT c FROM t WHERE c>1"`. Second, randomly modifying semantic-related data is likely to generate a semantically incorrect query, which a DBMS will refuse to execute. For example, replacing `c` in query `A` with the column in another table leads to an invalid query. In either case, random data mutation is less productive than random structure mutation.

Therefore, we strip data from the query IR and apply mutation mainly on structures. We leave the data modification in Section 3.5.

### 3.4.1 Structure-Data Separation

We walk through the IRs to replace each data with a predefined value based on its type `data_type`. Specifically, we replace semantic data with string `"x"`, change con-

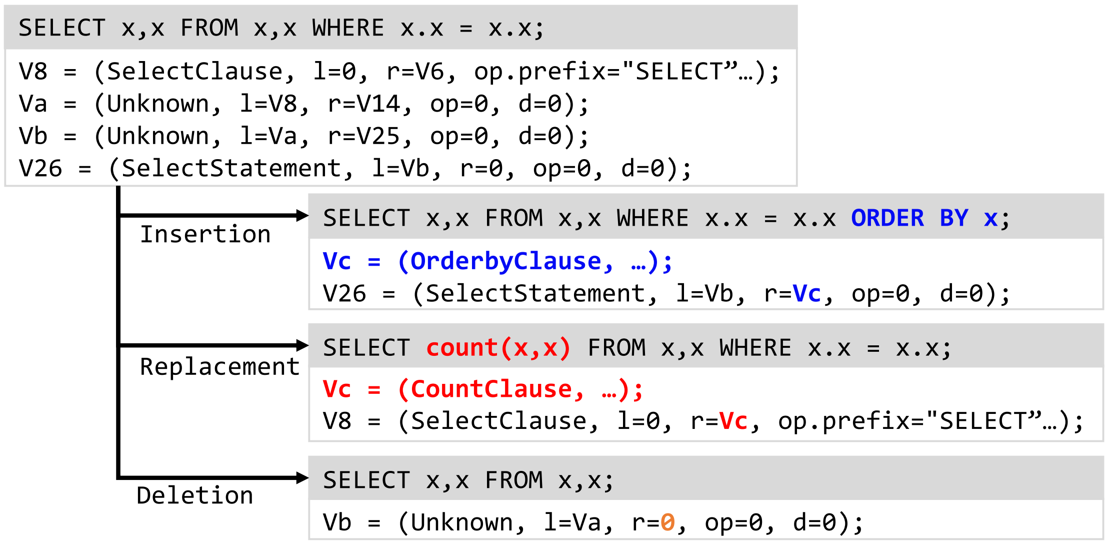

SELECT x,x FROM x,x WHERE x.x = x.x;
V8 = (SelectClause, l=0, r=V6, op.prefix="SELECT"…);
Va = (Unknown, l=V8, r=V14, op=0, d=0);
Vb = (Unknown, l=Va, r=V25, op=0, d=0);
V26 = (SelectStatement, l=Vb, r=0, op=0, d=0);
Insertion
SELECT x,x FROM x,x WHERE x.x = x.x ORDER BY x;
Vc = (OrderbyClause, …);
V26 = (SelectStatement, l=Vb, r=Vc, op=0, d=0);
Replacement
SELECT count(x,x) FROM x,x WHERE x.x = x.x;
Vc = (CountClause, …);
V8 = (SelectClause, l=0, r=Vc, op.prefix="SELECT"…);
Deletion
SELECT x,x FROM x,x;
Vb = (Unknown, l=Va, r=0, op=0, d=0);

Figure 3.5: **Mutation strategies on IR programs.** We have type-based insertion and replacement, and deletion of optional operands.

stant numbers to 1 or 1.0 and update all strings to `"a"`. Therefore, after the separation, the running example `"SELECT c2,c6 FROM t1,t2 WHERE t1.c1=t2.c5"` becomes `"SELECT x,x FROM x,x, WHERE x.x=x.x"`. Both query `A` and `B` become `"SELECT x FROM x WHERE x=1"`, while `C` is changed to `"SELECT x FROM x WHERE x>"`.

**Storing IR in Library.** We use a dictionary called *IR library* to store various IRs. The key of the dictionary is the IR type, while the value is a list of IRs. IRs in one list have the same type, and their structures are exclusively different. For example, after separation, queries `A`, `B`, and `C` have the same `SelectStmt` type, and they should be stored in the same list. We remove query `B` from the list as it shares the same structure as `A`. Whenever we need an IR of a certain type, we find the corresponding list from the dictionary and randomly return one element. As we show in Figure 3.2, SQUIRREL accepts seed queries to initialize the IR library. Whenever SQUIRREL finds that the generated IR has a new structure, we add it to the corresponding list in the library. We set a limit on the maximum number of IRs in the library to avoid excessive memory usage.

### 3.4.2 Type-Based Mutation

We define a set of type-based mutations to update the left and right operands of an IR or the IR itself. Our mutation focus on operands as other members of the IR cannot be easily changed: the operator is closely related to the IR type, like the `SELECT` operator in

`SelectClause` IR, while `data_type` is decided by its position in the query, like a variable after `"CREATE TABLE"` must be a table name. Therefore, our mutations either operate on the whole IR or modify its operands. Specifically, for each IR $v$ in the IR program we perform the following mutations with certain probability:

- **Insertion** adds an IR into an appropriate position of $v$. If the left child of $v$ is empty, we randomly pick an IR $w$ from the IR library that shares the same type as $v$. If the left child of $w$ is not empty, we use it as the left child of $v$. The same operation applies to the right child.
- **Replacement** changes $v$ or its operands. We first randomly pick one IR $w$ of the same type as $v$ from the IR library. Then we copy the children of $w$ to $v$, or we can replace $v$ with $w$ and update all $v$'s references to $w$.
- **Deletion** removes a $v$ as a whole by simply replacing it with an empty IR. The same operation can be applied to its children.

Since we essentially manipulate IRs based on their types, the syntactic correctness is preserved with a high probability. To further improve the syntactic correctness, we convert the mutated IRs back to an SQL query and perform syntax validation with our parser. If the parsing succeeds, we conclude that this query has no syntax errors and will use it in the next stage. Otherwise, we discard the new IRs and try to generate another one.

Figure 3.5 shows an example of mutating the IRs of our running example to generate three new queries. Specifically, we insert an `ORDERBY` clause to the right child of `V26`; we replace the right child of `V8` with a `CountClause`, where the new query counts the rows of the original results; we delete the right child of `Vb` to effectively remove the `WHERE` clause. All three new IRs are syntactically correct.

**Unknown Type.** As we mention in Section 3.3.1, some IRs have type `Unknown` as they do not have corresponding nodes in the AST. We use `Unknown` type to perform fuzzy type-matching without searching for concrete types, which may accelerate our query generation. The syntax validation, which always needs one-time parsing, is unaffected. However, without accurate type-matching, SQUIRREL may create some invalid queries.

## 3.5 Semantics-Guided Instantiation

Semantics-correct queries enable fuzzers to dig deeply into DBMSs' execution logic and discover bugs effectively. However, generating semantics-correct test cases is an unsolved

```
CREATE TABLE x1 (x2 INT, x3 INT)
CREATE TABLE x4 (x5 INT, x6 INT)
CREATE TABLE x7 (x8 INT, x9 INT)
SELECT x10, x11 FROM x12, x13 WHERE x14.x15 = x16.x17
```

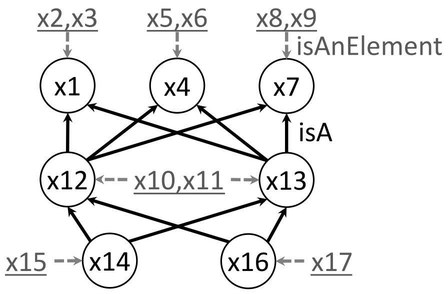
| Data | Type | Relation |
|---|---|---|
| x1, x4, x7 | CreateTable |  x2,x3<br>x5,x6<br>x8,x9<br>isAnElement<br>x1<br>x4<br>x7<br>isA<br>x12<br>x10,x11<br>x13<br>x15<br>x14<br>x16<br>x17 |
| x2, x3 | CreateColumn | |
| x5, x6 | CreateColumn | |
| x8, x9 | CreateColumn | |
| x10, x11 | UseFromColumn | |
| x12, x13 | UseAnyTable | |
| x14, x16 | UseFromTable | |
| x15 | UseTableColumn | |
| x17 | UseTableColumn | |

Figure 3.6: **Data dependency example.** This example consists of three new `CREATE` statements and our running example. In `Relation`, we show two types of relations: `isAnElement` (dashed line) and `isA` (solid line).

challenge for fuzzing programs that take structured, semantics-binding inputs [74]. Previous research shows that 90% of the test cases generated by jsfunfuzz [84], a state-of-the-art JavaScript fuzzer, are semantically invalid [52]. Similar problems also exist in DBMS testing.

We propose a data instantiation algorithm to improve the semantic correctness of generated SQL queries. As mentioned in Section 3.4, after mutation the IR program is a syntax-correct skeleton with data stripped. Our instantiator first analyzes the dependency between different data, and fill the skeleton with concrete values that satisfy all dependencies. After the instantiation, the query has a high chance to be semantically correct.

## 3.5.1 Data Dependency Inference

Data dependency describes the relationship between semantic-binding data. Any unsatisfied dependency will make the query fail semantic checks. Figure 3.6 shows the data dependencies among four SQL statements, including three `CREATE` statements and our motivating example. Our syntax-correct mutation has replaced each variable with `x`. To distinguish different `x`, we assign an index to each of them. These four statements contain two types of relationships: one defines `A` is an element of `B` (`isAnElement`), shown as gray-dashed lines like `x2` is a column of `x1`; another describes that `A` can be `B` (`isA`), shown as black-solid lines like `x12` can be `x1`.

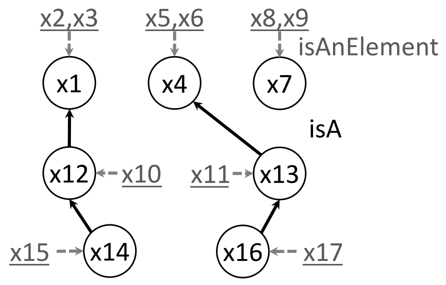
We define a set of rules to automatically infer dependencies between data in the query. These rules follow two principles. The *lifetime* principle requires us to create SQL variables before using them, and stop using a variable after its deletion. The *customization* principle requires us to consider the data types, scopes and operations to determine the relationship. We need to refine data types mentioned in Section 3.3.1 to accurately describe data dependencies.

**Data Type.** We refine each data type so that it not only describes the semantic meaning but also reflects the usage context. Database elements in different statements, even with the same basic type, can have different dependencies. Based on the lifetime principle, we include the define/use information into the data type, to indicate whether the element is a new definition or use of an existing one. For example, a table in a `CREATE TABLE` clause will have `CreateTable` type, while the table in a `FROM` clause will have `UseTable` type. Based on the customization principle, we also include the scope of the data to show where to find the potential candidate values. For example, a table in a `FROM` clause can be any defined table, thus having type `UseAnyTable`, while the table in a `WHERE` clause must be one of the tables in the `FROM` clause, thus having the `UseFromTable` type. The refined data type of a variable is determined by its position in the AST. Therefore, SQUIRREL identifies and sets the data types during the query parse and translation.

Figure 3.6 shows the refined type for each IR data. For example, `x1` is a newly defined table and thus has type `CreateTable`. `x2` and `x3` have type `CreateColumn`. `x12` has type `UseAnyTable` as it can be any defined table (`x1`, `x4`, `x7`), while `x14` can only be tables listed in the `FROM` clause. `x10` can be any column of tables in `FROM`, while `x15` can only be the column of table `x14`.

**Data Relation Rule.** With refined data types, we can further define data relation rules that can help automatically infer data dependencies. A relation rule is a tuple $(\alpha, \beta, \gamma, S)$ of four elements: $\alpha$ is the relation target and $\beta$ is the relation source; $\gamma$ defines the relationship; $S$ denotes the scope of the relationship, including `intraStmt` for relations in the same statement, `interStmt` for relations across multiple statements, `any` for any instance while `nearest` for the element with the shortest path according to the Define-Use chain. We define eight general rules for all DBMSs, one extra rule for `SQLite`, two extra rules for `MySQL`, two extra for `MariaDB` and one for `PostgreSQL`. For example, the relation rule (`UseFromTable`, `UseTableColumn`, `isAnElement`, `nearest`) means the data of type `UseTableColumn` is an element of the nearest data within the same statement that has type `UseFromTable`. In Figure 3.6, we can infer the relationship

**One possible relation**



| Data | Value |
|---|---|
| x1, x12, x14 | v1 |
| x2, x15 | v2 |
| x3, x10 | v3 |
| x4, x13, x16 | v4 |
| x5, x17 | v5 |
| x6, x11 | v6 |
| x7 | v7 |
| x8 | v8 |
| x9 | v9 |

```
CREATE TABLE v1 (v2 INT, v3 INT)
CREATE TABLE v4 (v5 INT, v6 INT)
CREATE TABLE v7 (v8 INT, v9 INT)
SELECT v3, v6 FROM v1, v4 WHERE v1.v2 = v4.v5
```

Figure 3.7: **Instantiation of IR structure.** We create one concrete dependency graph from the dependencies of Figure 3.6, replace all place-holder `x` and finally get one concrete new query.

between `x15` and `x14` with this relation rule.

**Dependency Graph.** With data types and relations, SQUIRREL automatically constructs a dependency graph $G = \{V, E\}$ for each mutated IR program. Each node in $V$ is an IR data and its data type. Each edge in $E$ describes one relationship from the edge source to the target. If a data type potentially depends on two or more data types, we randomly choose one to avoid circular dependency. Also, if there are multiple candidate values for the dependent type, SQUIRREL randomly picks one to establish the edge. In this way, every node in the graph has at most one parent, and the dependency graph is formed as one tree or several trees.

Figure 3.7 shows one possible dependency graph constructed from Figure 3.6. For example, we choose `x1` as the dependency of `x12`, although it can be any one of (`x1, x4, x7`) according to Figure 3.6. Based on different choices, we can create multiple concrete data dependency graphs for each mutated IR. There are some details not shown in the figure, like `x10` should be one of `x2` and `x3`. SQUIRREL handles these implicit dependencies properly.

## 3.5.2 IR Instantiation

SQUIRREL instantiates the query by filling in concrete data. algorithm 1 shows our instantiation algorithm. For each tree in the dependency graph, we sort its nodes based on the breadth-first search and the statement order, which guarantees the lifetime

**Algorithm 1:** Semantic Instantiation

```
   Input   : graph: The dependency graph of the IR program
   Output  : sqlQuery: A SQL query that can be executed by DBMS. Empty if error
 1 Procedure Instantiation(graph)
 2     dataMap ← map();
 3     relationMap ← map();
 4     for each tree $T \in graph$ do
 5         for each node $N \in T$ do
 6             if N.data is literal then
 7                 N.data ← Random predefined or generated data ;
 8             else if N has no parent then
 9                 if N.datatype is Definition then
10                     N.data ← GenerateUniqueString();
11                     dataMap[N.datatype].insert(N.data) ;
12                 else if N.datatype is Use then
                       // Use before definition
13                     if dataMap[N.datatype] is empty then  return Null ;
                       // Some predefine name, like Function name
14                     else
15                         N.Data ← anyone ∈ dataMap[N.datatype] ;
16                 else
17                     return Null ;
18             else
19                 pN ← The parent node of N;
20                 if N.datatype is Definition then
21                     N.data ← GenerateUniqueString();
22                     dataMap[N.datatype].insert(N.data) ;
23                     relationMap[pN.data].insert(N.data);
24                 else if N.datatype is Use then
25                     if N.type is the same as pN.type then
26                         N.data ← pN.data; ;
27                     else if relationMap[pN.data] is not empty then
28                         N.data ← anyone ∈ relationMap[pN.data];
29                     else
30                         return Null ;
31                 else
32                     N.Data ← pN.data;
33                     Delete pN.data from dataMap and relationMap;
34     return sqlQuery ← Tranlate the instantiated IR program to a SQL query;
```

correctness. For literal data like integers, we set it to a random value or one from a predefined value set (line 5). For semantic-binding data, we fill in appropriate, valid names. During the process, we maintain two maps: `dataMap` tracks unique names with different types, while `relationMap` maps each element to its dependency. If the current node has no dependency (line 6), it either defines a new variable, where we create a new unique string as its name (line 7-9), or is a predefined term, like a function name (line

10-13). If the current node has a parent, we know that it has some dependencies (line 15-27): if the current node creates a new variable, we simply generate a unique string for it (line 16-19); if the current node uses a variable, we check the `relationMap` to find a proper value for it (line 22-23). Finally, we translate the IR program back into an SQL query and return. If the process fails because of unsatisfiable dependency, the IR program will contain semantic errors.

Figure 3.7 shows the result of instantiation the in `Data` and `Value` column, and the final SQL query. SQUIRREL assigns `v1` to `x1` as it is a `CreateTable` without dependency. Due to the statement order, we process `x2` and `x3` next and allocate names `v2` and `v3` for them, respectively. We handle `x4`-`x9` in a similar way. For `x12`, it is with type `UseAnyTable` and has a dependency on `x1`, so based on line 23 of algorithm 1, we assign `x1`'s name `v1` to `x12`. Other data can be instantiated in the same way.

## 3.6 Implementation

Table 3.1: **Code size of Squirrel components**, totally 43,783 LoC.

| **Module** | Translator | Mutator | Instantiator | Fuzzer | Others |
|---|---:|---:|---:|---:|---:|
| **Language** | C++/Bison/Flex | C++ | C++ | C++ | C++/Make |
| **LoC** | 32,947 | 4,572 | 1,880 | 2,208 | 2,176 |

We implement SQUIRREL with 43,783 lines of code. Table 3.1 shows the breakdown of different components.

**AST parser.** We develop a general AST parser that can handle the common features of various DBMSs and customize the parser for each DBMS to support implementation-specific features. Our implementation utilizes Bison 3.3.2 and Flex 2.6.4 and follows the grammar specification in official DBMS documentation While we support most of the specification's grammars related to database manipulation, we leave out parts related to administrative functionalities. In this way, we can focus on testing those grammars related to database manipulation.

**Fuzzer.** We build SQUIRREL on top of `AFL` 2.56b, and replace its mutator with our syntax-preserving mutator and semantics-guided instantiator. Whenever an interesting test case is discovered, we store its stripped IRs into the IR library. We drop the database after each query to minimize the interplay between different queries.

**Effort of Adoption.** The effort of adopting SQUIRREL to other DBMSs could be

DBMS-dependent. First, we should customize the general parser to support the unique features of the target DBMS. Second, we will write semantic relation rules according to the grammar. Third, if the DBMS runs in client-server mode (e.g., `MySQL` and `PostgreSQL`), we need to implement a client for it. In our cases, it took one of our author one day to customize the parser, and less than six hours to implement the semantic rules and the client for each DBMS we tested. We believe it should take no more than two days to adopt Squirrel to another DBMS.

## 3.7 Evaluation

We applied Squirrel on real-world relational DBMSs to understand its effectiveness on finding memory error bugs. Specifically, our evaluation aims to answer the following questions:

- Can Squirrel detect memory errors from real-world production-level DBMSs? (Section 3.7.1)
- Can Squirrel outperform state-of-the-art testing tools? (Section 3.7.2)
- What are the contributions of language correctness and coverage-based feedback in DBMS testing? (Section 3.7.3)

**Benchmarks.** We select three widely used DBMSs for extensive evaluation, including `SQLite` [55], `PostgreSQL` [99], `MySQL` [85]. We also test `MariaDB` [77] with Squirrel just for finding bugs. We compile them with default configurations and compilation options. We compare Squirrel with five fuzzers, including the mutation-based fuzzers `AFL` [144] and `Angora` [17], the hybrid fuzzer `QSYM` [143], the structural fuzzer `GRIMOIRE` [9] and the generation-based fuzzer `SQLsmith` [106]. We try to run as many tests as possible, but as shown in Table 3.2, we encounter several compatibility issues. As `MySQL` requires a client to send the query (i.e., C/S mode), `QSYM`, `Angora` and `GRIMOIRE` cannot test it directly. `SQLsmith` does not officially support `MySQL` due to the lack of interfaces [107]. `PostgreSQL` supports both C/S mode and single mode, and we can use `QSYM` to test it in the single mode. However, `GRIMOIRE` cannot compile `PostgreSQL` successfully to a single static binary; `Angora` can compile it but cannot run the binary. We are actively seeking potential solutions.

**Seed Corpus.** We obtain seed inputs from the official Github repository of each DBMS, where the unit tests usually cover most types of queries. All six fuzzers in our evaluation, except for `SQLsmith` (which does not need any initial inputs.), use the same seeds.

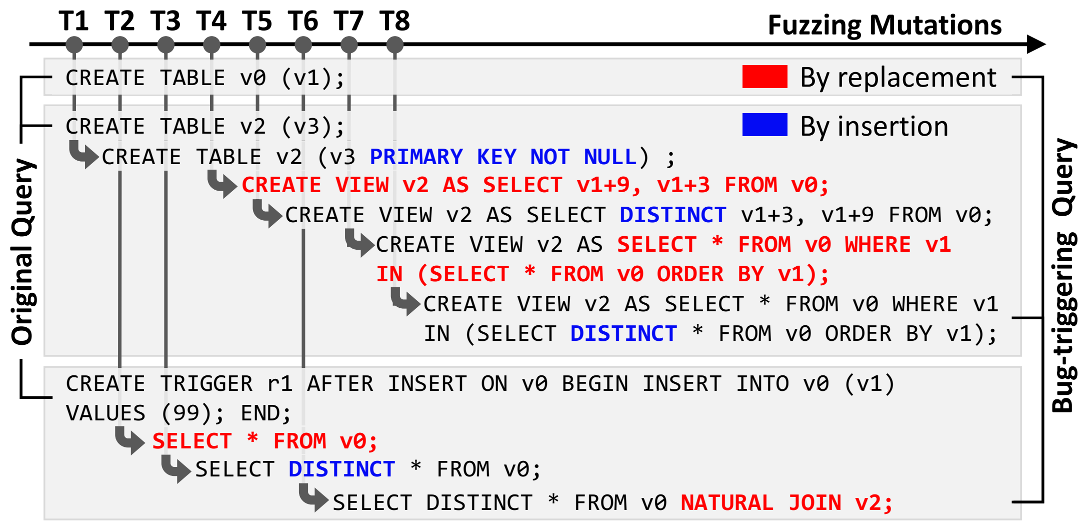
Table 3.2: **Compatibility between fuzzers and DBMSs.** MySQL only permits C/S mode, which is not supported by the last three fuzzers. SQLsmith does not support MySQL's grammar. QSYM supports fuzzing PostgreSQL in the single mode, GRIMOIRE cannot compile it, while Angora cannot run it.

| | Squirrel | AFL | SQLsmith | QSYM | Angora | GRIMOIRE |
|---|---|---|---|---|---|---|
| SQLite | ✔ | ✔ | ✔ | ✔ | ✔ | ✔ |
| PostgreSQL | ✔ | ✔ | ✔ | ✔ (single) | ✗ (execute) | ✗ (compile) |
| MySQL | ✔ | ✔ | ✗ (interface) | ✗ (C/S) | ✗ (C/S) | ✗ (C/S) |

**Setup.** We perform our evaluation in a Ubuntu 16.04 system, on a machine that has Intel Xeon CPU E5-2690 (2.90GHz) with 16 cores and 188GB RAM. We use `afl-clang` with llvm mode to instrument tested DBMSs, and adopt edge-coverage for the feedback. Considering the large codebase of DBMSs, we use a bitmap with 256K bytes to mitigate the path collision issue [36]. Angora uses a 1024K-byte bitmap, the default size by design. For bug detection, due to the time limit and the implementation progress, we have run Squirrel to test SQLite for 40 days, MySQL and PostgreSQL for 11 days, and MariaDB for 1 day. For other evaluations, we run each fuzzing instance (fuzzer+DBMS) for 24 hours and repeat the process five times. Each fuzzing instance runs separately in a docker with one CPU and 10G memory. We report the average results to reduce the random noise and provide the p-values in Table 3.4.

### 3.7.1 DBMS Bugs

Squirrel has successfully detected 63 bugs from tested DBMSs, including 51 bugs from SQLite, 7 from MySQL, and 5 from MariaDB. We have responsibly reported all the bugs to the corresponding DBMS developers and have received their positive feedback. At the time of this thesis writing, all bugs have been fixed, and 12 got CVE numbers due to the severe security consequences. As a comparison, the OSS-Fuzzer project launched by Google [42] have extensively tested the first three DBMSs, and found 19 bugs from SQLite in three years, 15 bugs from MySQL in four months and no bugs from PostgreSQL. We inspect the MySQL bugs detected by OSS-Fuzzer and find that all of them happen at the very beginning of MySQL's logic: before the parsing phase. The proof-of-concepts (PoC) are not even valid SQL queries but just some random bits. Therefore, we believe our fuzzer can find bugs from DBMSs more effectively.



```
CREATE TABLE v0 (v1);
CREATE TABLE v2 (v3);
  CREATE TABLE v2 (v3 PRIMARY KEY NOT NULL) ;
    CREATE VIEW v2 AS SELECT v1+9, v1+3 FROM v0;
      CREATE VIEW v2 AS SELECT DISTINCT v1+3, v1+9 FROM v0;
        CREATE VIEW v2 AS SELECT * FROM v0 WHERE v1
        IN (SELECT * FROM v0 ORDER BY v1);
          CREATE VIEW v2 AS SELECT * FROM v0 WHERE v1
          IN (SELECT DISTINCT * FROM v0 ORDER BY v1);
CREATE TRIGGER r1 AFTER INSERT ON v0 BEGIN INSERT INTO v0 (v1)
VALUES (99); END;
    SELECT * FROM v0;
      SELECT DISTINCT * FROM v0;
        SELECT DISTINCT * FROM v0 NATURAL JOIN v2;
```

Figure 3.8: **Mutations to trigger the 11-year old bug.** SQUIRREL takes eight steps, including four insertions and four replacements to produce the bug-triggering query from the original one.

**Bug Diversity.** The 63 bugs cover almost all common types of memory errors, showing that SQUIRREL can improve DBMS security from a variety of aspects. In particular, buffer overflows and use-after-free bugs are commonly believed to be exploitable, whereas SQUIRREL found **12** bugs and **2** bugs, respectively. SQUIRREL also detected **33** assertion failures from `SQLite`, which indicate that the executions reach unexpected states. Even worse, assertion checks are disabled in the released binary, which may lead to severe security problems. For example, in case study 3, an assertion failure results in a severe use-after-free vulnerability.

```sql
CREATE TABLE v0 (v1);
CREATE VIEW v2 AS SELECT * FROM v0 WHERE v1
    IN (SELECT DISTINCT* FROM v0 ORDER BY v1);
SELECT DISTINCT * FROM v0 NATURAL JOIN v2;
```

Listing 3.1: Case Study 1: 11-Year-Old Bug

**Case Study 1: An 11-Year-Old Bug.** SQUIRREL detected a bug introduced to `SQLite` 11 years ago (PoC in Listing 3.1), which lies in an optimization routine of the `IN` clause. Specifically, `isCandidateForInOpt` checks various conditions to determine whether the subquery inside the `IN` clause can be optimized or not. One of these checks should make sure the subquery does not have any `GROUP BY` clause. As the SQL grammar does not allow `GROUP BY` in an `IN` clause, the developers “were unable to find a test case for”[1] this condition, and thus converted the check “into an assert() by check-in [...] (2009-05-28)”. The assertion is disabled in the released version of `SQLite`. Squirrel found that if two queries with `DISTINCT` are joined by `NATURAL JOIN`, `SQLite` will *internally* set the `GROUP BY` property to these queries. When such queries are used in an `IN` clause, it will make the previous assertion fail. However, the released `SQLite` will continue the optimization incorrectly, and may lead to wrong query results.

Squirrel found this 11-year-old bug within 14 minutes through only eight mutations. Figure 3.8 shows the eight steps for Squirrel to generate the bug-triggering query from a benign one. We denote $T_n$ as the $n$th mutation. The original query contains three `CREATE` statements: the first `CREATE` does not have to change; the second `CREATE` is changed five times with three insertions ($T_1$, $T_5$ and $T_8$) and two replacements ($T_4$ and $T_7$); the last `CREATE` is changed three times with two replacements ($T_2$ and $T_6$) and one insertion ($T_3$). Each round of mutation provides a new syntactical structure, and keeps both syntactical and semantic correctness. The final query satisfies the conditions for failing the assertion, as `SQLite` will put the last `SELECT` into the `IN` of the second statement, which makes the subquery of `IN` contain two naturally joined `SELECT`s.

```
CREATE TABLE v0 (v1 char);
INSERT INTO v0 VALUES ('1');
CREATE TABLE v2(v3 text);
INSERT INTO v2 VALUES ("1"*147), ("2"*42), ("3"*37);
DROP TABLE v2;
INSERT INTO V0 SELECT ZIPFILE(v1, NULL) FROM v0;
INSERT INTO V0 SELECT ZIPFILE(v1, NULL) FROM v0;
INSERT INTO V0 SELECT ZIPFILE(v1, NULL) FROM v0;
SELECT HEX(v1) FROM v0;
```

Listing 3.2: Case Study 2: Database leakage

**Case Study 2: Database Leakage.** Squirrel identified a heap-based buffer overread vulnerability (PoC in Listing 3.2), which allows attackers to read arbitrary data in the memory space. This bug entitles attackers to potentially access all databases stored in the `SQLite` DBMS. As `SQLite` is widely used as a multi-user service, attackers can retrieve the data of other users, which by default they have no access. Even if the database is explicitly deleted by its owner, attackers can still steal it from its memory residue. Other than stealing databases, this bug also enables attackers to read sensitive-critical information that may allow attackers to build further attacks, like remote code execution. For

[1]Quote from the message of the fix commit.

example, reading the code page address will help attackers bypass randomization-based defenses, while leaking the stack canary will make stack buffer-overflow exploitable again.

```sql
CREATE TABLE v0 (a);
CREATE VIEW v2 (v3) AS WITH x1 AS (SELECT * FROM v2);
SELECT v3 AS x, v3 AS y FROM v2;
```

Listing 3.3: Case Study 3: UAF from assertion

**Case Study 3: Assertion Failure Leading to Use-After-Free.** We inspected several assertion failures and found one (PoC in Listing 3.3) finally leads to a **high-severity** use-after-free bug (score 7.5/10). An assertion affirms that a predicate is always true whenever the execution reaches the assertion point. Otherwise, the developer considers the program running into an unexpected state. This bug makes `SQLite` fail a `pParse->pWith` assertion, as `pWith` is a dangling pointer due to a failed creation of a circular view. In the debug mode, `SQLite` will terminate the execution after the assertion failure. However, the released binary disables all assertions. Given the bug-triggering input, `SQLite` will keep running in the unexpected state and finally trigger the use-after-free bug.

```sql
CREATE TABLE v0 ( v1 INTEGER PRIMARY KEY ) ;
INSERT INTO v0 ( v1 ) VALUES ( 0 )
    ON CONFLICT DO NOTHING ;
CREATE VIRTUAL TABLE v2
    USING rtree(v5 UNIQUE ON CONFLICT ABORT, v4, v3);
SELECT 'a' FROM v0
    LEFT JOIN v2 ON v4 = 10 OR v5 = 10 ;
SELECT * FROM v0 , v0 WHERE v1 = v1 AND v1 = 1;
```

Listing 3.4: Case Study 4: Bug exists for 1 day

```sql
CREATE TABLE v0 (v1 DOUBLE CHECK((v1 IN (NULL))),
              v2 UNIQUE AS(v1>v1)) ;
INSERT INTO v0
    VALUES (10) ON CONFLICT DO NOTHING ;
SELECT 10.100000, 10 FROM v0
    CROSS JOIN v0 USING (v1) ;
```

Listing 3.5: Case Study 4: Bug exists for 1 hour

**Case Study 4: Fuzzing as Regression Test.** SQUIRREL can effectively find newly-introduced bugs, and thus can be used for rapid regression test. For example, Listing 3.4 only exists for less than one day before we found it. PoC in Listing 3.5 got detected, reported and even fixed in just one hour after its existence. The commit introducing

Table 3.3: **Distribution of SQLite bugs found by fuzzers.** We perform the evaluation for 24 hours, repeat for 5 times and report the average results. We update SQLite binary per-hour to fix identified bugs.

| Target | Type | Squirrel | AFL | SQLsmith | QSYM | Angora | GRIMOIRE | !semantic | !feedback |
|---|---|---|---|---|---|---|---|---|---|
| SQLite | NULL Ptr Deref | ✔ | ✔ | ✗ | ✔ | ✗ | ✗ | ✔ | ✔ |
| SQLite | Use-After-Free | ✔ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SQLite | Buffer Overflow | ✔ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SQLite | Buffer Overflow | ✔ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SQLite | NULL Ptr Deref | ✔ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SQLite | Stack Overflow | ✔ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SQLite | NULL Ptr Deref | ✔ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SQLite | NULL Ptr Deref | ✔ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |
| SQLite | Buffer Overflow | ✔ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ |

this bug was intended to fix another issue related to the generated column functionality. However, the fix was not completely correct and thus introduced a new problem in the USING clause. These two cases show that SQUIRREL can do fast and effective regression testing for DBMSs.

### 3.7.2 Comparison with Existing Tools

We compare SQUIRREL with five other advanced fuzzers to evaluate its efficacy in testing DBMSs and identify its strengths and weaknesses. Figure 3.9 shows our evaluation results, including the number of unique crashes, the number of unique bugs, the number of new edges, the syntax correctness and the semantic correctness.

The p-values of our evaluations are shown in Table 3.4. Most of the p-values are less than 0.05, which means the differences between SQUIRREL's results and others' are statistically significant. We will discuss exceptional high p-values case by case.

**Unique Crashes.** We use the edge-coverage map to evaluate the number of unique crashes and present the results for SQLite in Figure Figure 3.9a. We exclude PostgreSQL and MySQL as only SQUIRREL finds few crashes in MySQL, and none of the other fuzzing instances find any crashes within 24 hours. While SQUIRREL detects the first crash in four minutes and finds around 600 unique crashes, AFL takes 32 minutes to detect the first crash and collects 30 unique crashes, and QSYM discovers 13 crashes, while the other

Table 3.4: **P-values of Squirrel *v.s.* other fuzzers.** P-value less than 0.05 (shown in **green**) means the result is statistically significant.

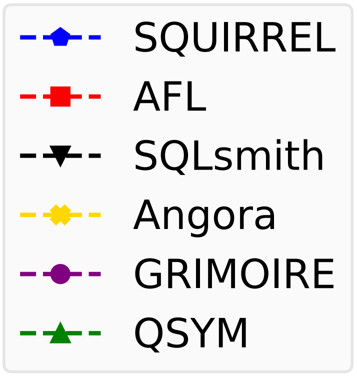
| ***v.s. Fuzzer*** | **DBMS** | **Coverage** | **Syntax** | **Semantics** | **Crash** | **Bug** |
|---|---|---|---|---|---|---|
| AFL | SQLite | **0.00609** | **0.00609** | **0.00609** | **0.00609** | **0.00198** |
| | PostgreSQL | **0.00609** | **0.000167** | **0.00225** | - | - |
| | MySQL | **0.00596** | **0.00609** | **0.00609** | - | - |
| SQLsmith | SQLite | **0.00545** | **0.00609** | **0.00609** | **0.00374** | **0.00198** |
| | PostgreSQL | **0.989** | **0.999** | **0.000166** | - | - |
| Angora | SQLite | **0.00609** | **0.00609** | **0.00609** | **0.00374** | **0.00198** |
| GRIMOIRE | SQLite | **0.00583** | **0.00374** | **0.00374** | **0.00374** | **0.00198** |
| QSYM | SQLite | **0.00405** | **0.00405** | **0.00405** | **0.00377** | **0.00198** |
| | PostgreSQL | **0.00583** | **0.000166** | **0.0181** | - | - |
| !semantic | SQLite | **0.00609** | **0.0183** | **0.00609** | **0.00557** | **0.00198** |
| | PostgreSQL | **0.00609** | **0.198** | **0.0181** | - | - |
| | MySQL | **0.00596** | **0.996** | **0.00609** | - | - |
| !feedback | SQLite | **0.00609** | **0.00609** | **0.00609** | **0.00485** | **0.00198** |
| | PostgreSQL | **0.00609** | **0.000167** | **0.000421** | - | - |
| | MySQL | **0.00596** | **0.0718** | **0.338** | - | - |

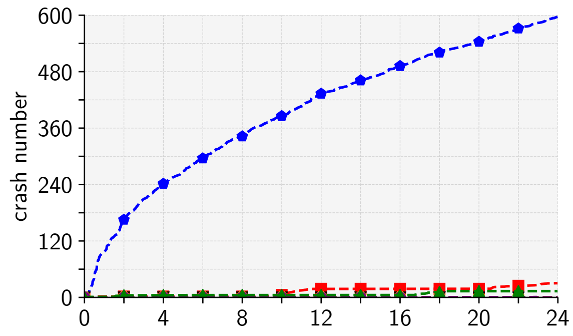
three fuzzers do not find any crashes.

The evaluation results reveal that although advanced fuzzing tools have been developed, they do not significantly improve performance compared to AFL. In fact, in some instances, these tools have even reported fewer unique crashes. This outcome can be explained by the stochastic nature of the fuzzing process and the rigorous semantic constraints necessary for adequately testing DBMS systems.

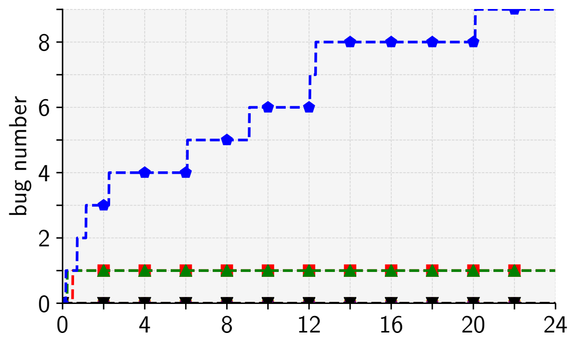
**Unique Bugs.** We map each crash with the corresponding bug by referring to the official patches. In SQLite, the 600 crashes found by SQUIRREL only are identified to two bugs, while the 30 crashes detected by AFL and the 13 found by QSYM belong to the same bug. Since the small number of bugs is not statistically useful, we take a different strategy to get more bugs: after every hour, we check the detected crashes (if any), map them to real bugs and patch them to avoid future similar crashes. We show the result of the new strategy in Figure 3.9b. By using this method, SQUIRREL is able to discover more bugs (from two to nine). AFL and QSYM can only find one bug within one hour, and have no progress even with the patching. Table 3.3 shows the distribution of detected bugs, where the only bug found by AFL and QSYM is also covered by SQUIRREL.

**New Edges.** Compared to mutation-based fuzzing tools, SQUIRREL can discover 2.0×-10.9× more new edges, indicating its effectiveness in exploring new program paths.

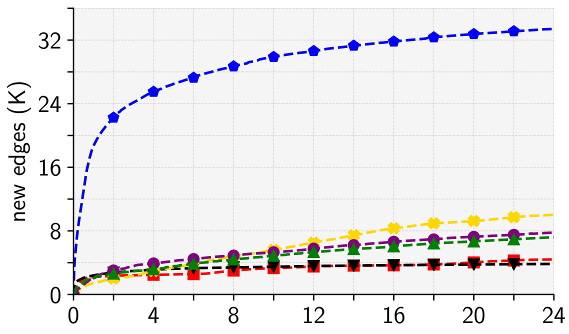




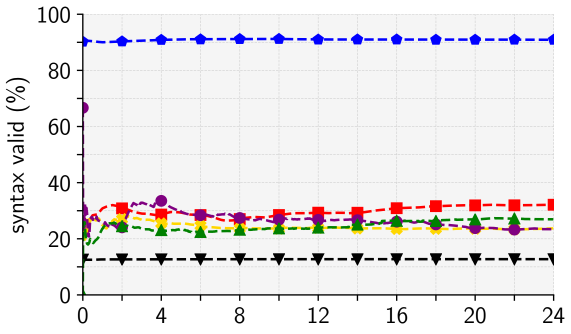
(a) `SQLite` crashes



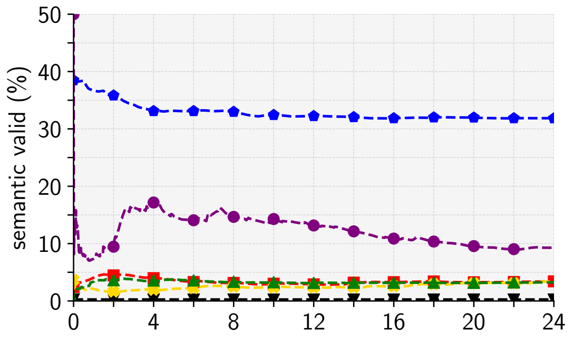
(b) `SQLite` bugs



(c) `SQLite` new edges

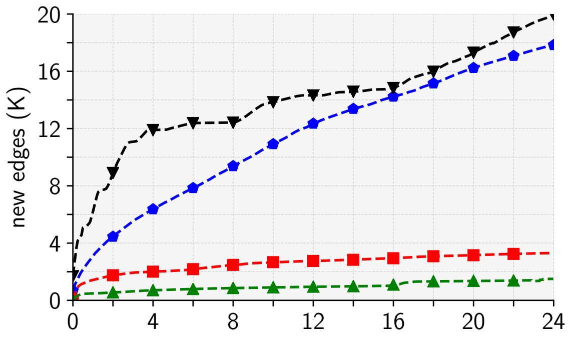


(d) `SQLite` syntax

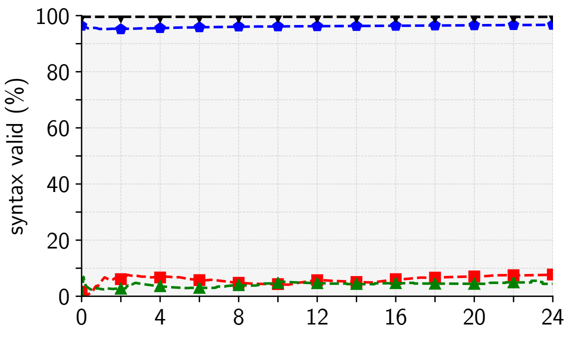


(e) `SQLite` semantic

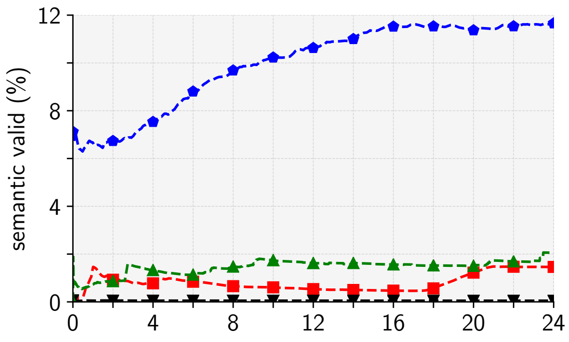
Additionally, our evaluation shows that Squirrel achieves similar results to `SQLsmith`, a generation-based fuzzer, regarding edge coverage. Figure 3.9c, f and i show the increase of new edges of `SQLite` (S), `PostgreSQL` (P) and `MySQL` (M), respectively. Squirrel outperforms other fuzzers in eight comparisons out of nine: it collects 6.6× (S), 4.4× (P) and 2.0× (M) more new edges than `AFL`, 7.7× (S) more than `SQLsmith`, 3.6× (S) and 10.9× (P) more than `QSYM`, 2.3× (S) more than `Angora`, and 3.3× (S) more than



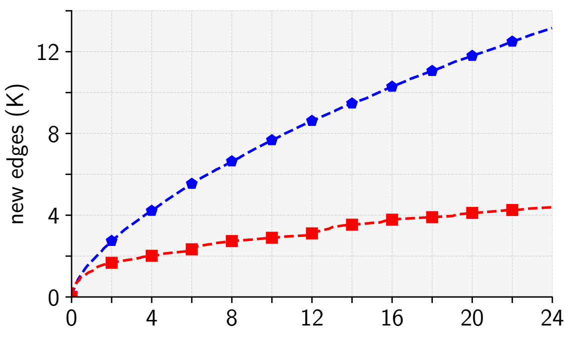
(f) `PostgreSQL` new edges



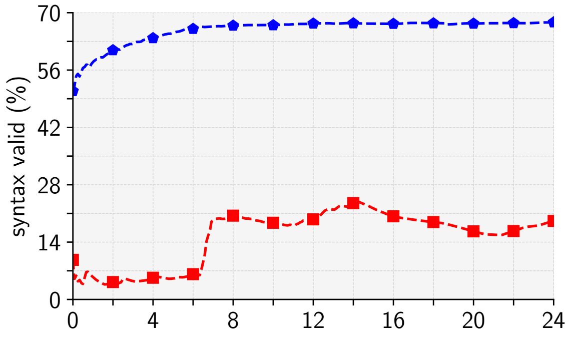
(g) `PostgreSQL` syntax.



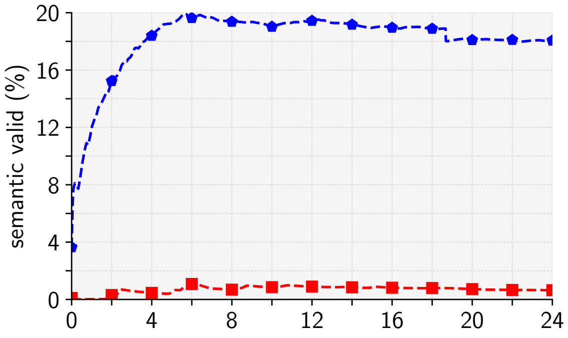
(h) `PostgreSQL` semantic



(i) `MySQL` new edges



(j) `MySQL` syntax



(k) `MySQL` semantic

Figure 3.9: **Comparison with existing tools.** Figures a-k show the bugs numbers of fuzzing `SQLite`, new edges, the syntax correctness, and the semantic correctness for fuzzing each DBMS

`GRIMOIRE`. All tested fuzzers show comparable performance, except for `SQLsmith` on `PostgreSQL`, where SQUIRREL detects 89.3% new edges compared to `SQLsmith`. This result is an expected result since `SQLsmith` is designed to the specific grammar of `PostgreSQL`. The corresponding p-value in Table 3.4 is larger than 0.05 due to the slightly better performance of `SQLsmith` on `PostgreSQL`.

**Syntax Validity.** Squirrel achieves 1.8×-20.9× *higher* syntax correctness than mutation-based tools, and gets a comparable result to SQLsmith. Figure 3.9d, g and j show the change of syntactic validity during testing SQLite (S), PostgreSQL (P) and MySQL (M), respectively. Squirrel achieves 1.8× (S), 11.5× (P) and 2.5× (M) higher syntax correctness than AFL, 6.1× (S) higher than SQLsmith, 2.4× (S) and 20.9× (P) higher than QSYM, 2.8× (S) higher than Angora, and 2.9× (S) higher than GRIMOIRE. The exception comes from fuzzing PostgreSQL with SQLsmith, where Squirrel achieves 97.1% syntactic validity of that by SQLsmith. Again, we believe the reason is that SQLsmith is highly customized for the specific grammar of PostgreSQL. For example, SQLsmith only achieves 12.7% syntax correctness in fuzzing SQLite, while getting almost 100% syntax accuracy when fuzzing PostgreSQL. The p-value of Squirrel *vs* SQLsmith on PostgreSQL is more than 0.05 due to the similar results.

Table 3.5: **The absolute number of generated test cases for the evaluated fuzzers in 24 hours.** We categorize them into three groups: one with syntax error, one with syntax correctness and semantic error, and one with syntax correctness and semantic correctness.

| Fuzzer | DBMS | Syntax-error | Semantics-error | Correct | Total |
|---|---|---:|---:|---:|---:|
| Squirrel | SQLite | 1,627,034 | 10,561,457 | 5,696,308 | 17,884,799 |
| | PostgreSQL | 7,287 | 188,055 | 25,762 | 221,104 |
| | MySQL | 50,728 | 77,750 | 28,314 | 156,792 |
| AFL | SQLite | 29,731,018 | 12,576,971 | 1,496,807 | 43,804,796 |
| | PostgreSQL | 3,604,530 | 245,226 | 57,166 | 3,906,922 |
| | MySQL | 185,102 | 42,332 | 1,478 | 228,912 |
| SQLsmith | SQLite | 20,205,598 | 2,891,250 | 58,351 | 23,155,199 |
| | PostgreSQL | 3,752 | 821,485 | 394 | 825,631 |
| Angora | SQLite | 14,540,639 | 3,866,102 | 654,526 | 19,061,267 |
| GRIMOIRE | SQLite | 4,799,412 | 886,471 | 581,500 | 6,267,383 |
| QSYM | SQLite | 8,652,045 | 2,805,585 | 385,028 | 11,842,658 |
| | PostgreSQL | 78,161 | 1,350 | 1,832 | 81,343 |
| !semantic | SQLite | 24,443,355 | 15,754,468 | 809,209 | 19,008,012 |
| | PostgreSQL | 9,591 | 184,441 | 3,174 | 197,206 |
| | MySQL | 17,140 | 165,175 | 15,370 | 197,685 |
| !feedback | SQLite | 19,415,174 | 8,146,566 | 27,368 | 27,589,108 |
| | PostgreSQL | 29,283 | 26,678 | 174 | 56,135 |
| | MySQL | 66,695 | 26,746 | 19,974 | 113,415 |

**Semantic Validity.** Squirrel achieves 2.4×-243.9× *higher* semantic correctness than other tools. Figure 3.9e, h and k show the trend of semantic validity during testing

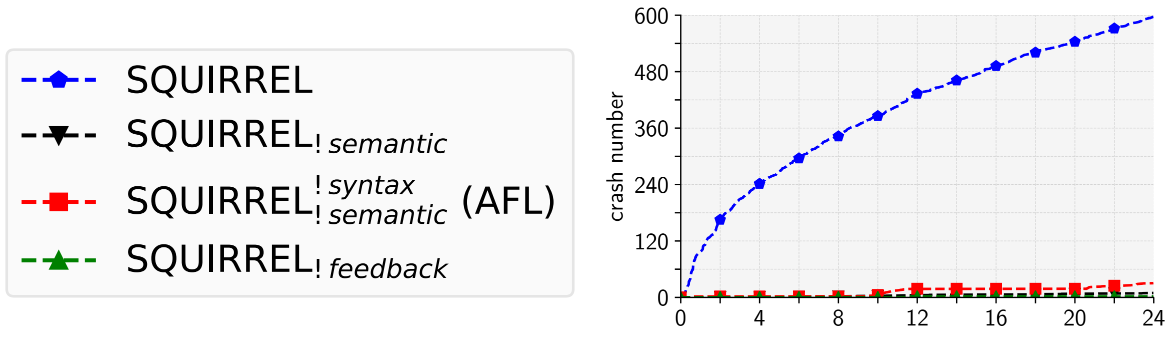
SQLite (S), PostgreSQL (P) and MySQL (M), respectively. Squirrel achieves 8.3× (S), 7.0× (P) and 27.0× (M) higher semantic correctness than AFL, 125.4× (S) and 243.9× (P) higher than SQLsmith, 8.8× (S) and 4.7× (P) higher than QSYM, 8.3× (S) higher than Angora, and 2.4× (S) higher than GRIMOIRE. Interestingly, although SQLsmith performs slightly better on testing PostgreSQL with respect to new edges and syntax correctness, Squirrel achieves significantly higher accuracy on semantics. Another worth-noting observation is that AFL actually generates more correct inputs for PostgreSQL than Squirrel (2.2×, see Table 3.5), but still achieves lower edge coverage. This indicates that larger numbers of correct queries or higher correct rates cannot guarantee the exploration of more program states. One extreme example is to keep using the same correct query, which will have more executions (no generation overhead) and 100% correctness. But apparently, it will lead to no increase in code coverage. The strength of Squirrel stems from both the syntax-preserved mutation, which generates queries of various structures, and the semantics-guided instantiation, which infers semantic relationships between arguments to assist query synthesis.

Squirrel outperforms all mutation-based tools, even if they are augmented with taint analysis or symbolic execution, or consider structural information. It achieves comparable results to SQLsmith, which is customized for PostgreSQL. More importantly, Squirrel detects significantly more bugs than other tools.

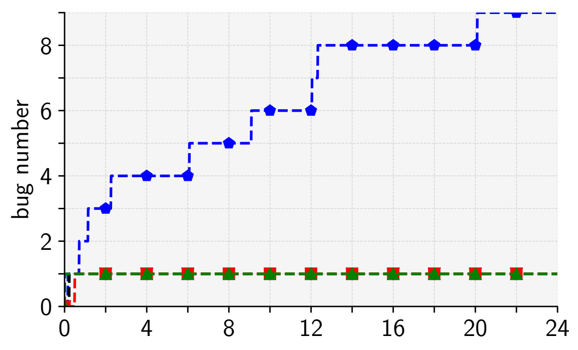
## 3.7.3 Contributions of Validity and Feedback

We conducted unit tests to evaluate the impact of different factors in Squirrel on the fuzzing process. Specifically, we disabled each factor individually, including syntax-preserving mutation, semantic-guided instantiation, and coverage-based feedback, and measured various aspects of the fuzzing process. The results are presented in Figure 3.10. In Squirrel$_{!semantic}$, we only disabled semantic-guided instantiation, while in Squirrel$_{!feedback}$, we only disabled coverage-based feedback. Squirrel$^{!syntax}_{!semantic}$ disabled both semantic-guided instantiation and syntax-correct mutation, making it identical to AFL. However, since our semantics-guided instantiation requires syntax-correct queries, we could not create a version that only disabled mutation. We did not include the all-disabled setting, which would represent the dumb mode of AFL.

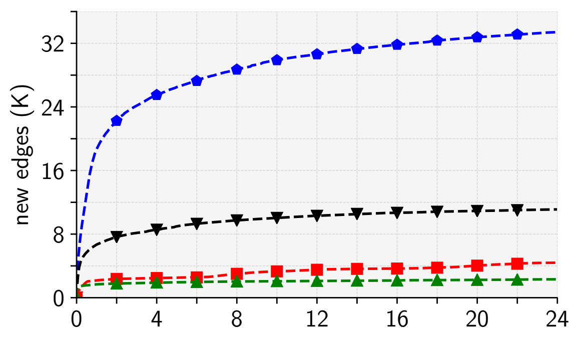
The p-values of our evaluations are shown in Table 3.4. Most p-values are less than 0.05, showing that the differences between Squirrel's results and others' are statistically significant. We will explain exceptional p-values higher than 0.05.



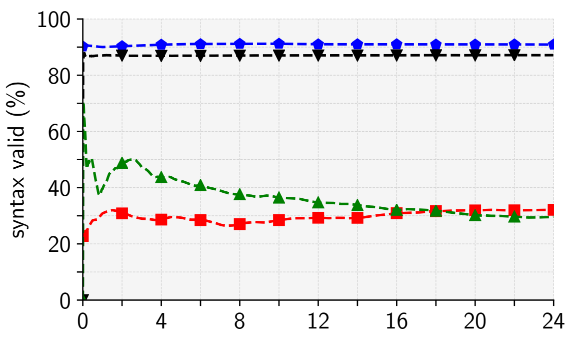
(a) `SQLite` crashes



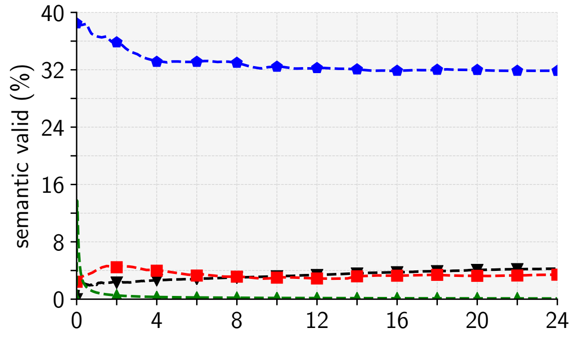
(b) `SQLite` bugs



(c) `SQLite` new edges

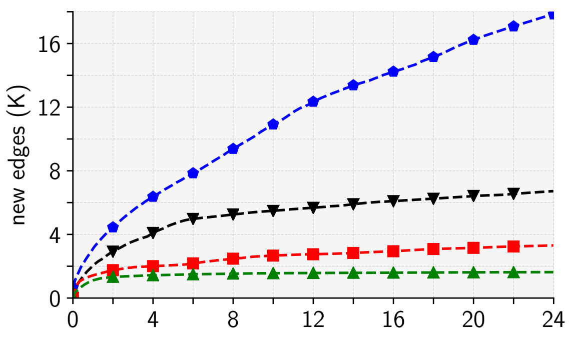


(d) `SQLite` syntax

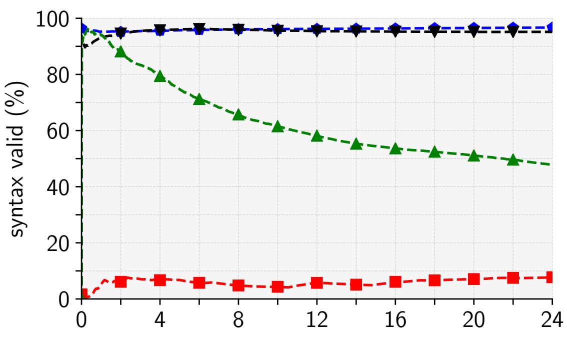


(e) `SQLite` semantic

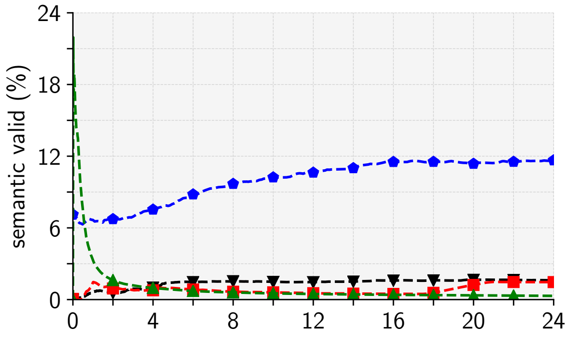
**Unique Crashes.** Figure 3.10a shows the number of unique crashes found in `SQLite` by each setting. Similarly, we skip the results of `PostgreSQL` and `MySQL` due to the small number of crashes during the 24-hour evaluation. The full-featured SQUIRREL achieves the best results in both the first-crash time and the total number of crashes. First, SQUIRREL finds the first crash within four minutes. Without semantic-guided instantiation, SQUIRREL$_{!semantic}$ takes 60× more time to detect the first crash (261 minutes). Interestingly, SQUIRREL$^{!syntax}_{!semantic}$, which directly uses `AFL`'s random mutation, finds the first crash within 32 minutes — worse than SQUIRREL, but better than SQUIRREL$_{!semantic}$.



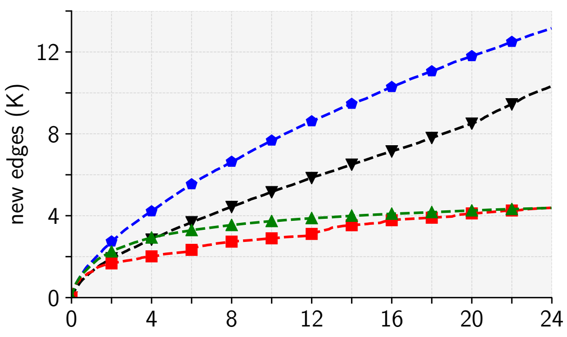
(f) `PostgreSQL` new edges



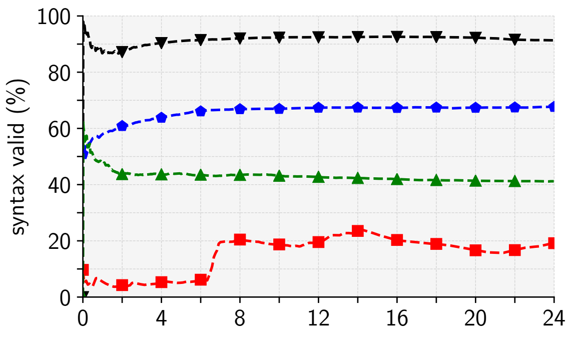
(g) `PostgreSQL` syntax



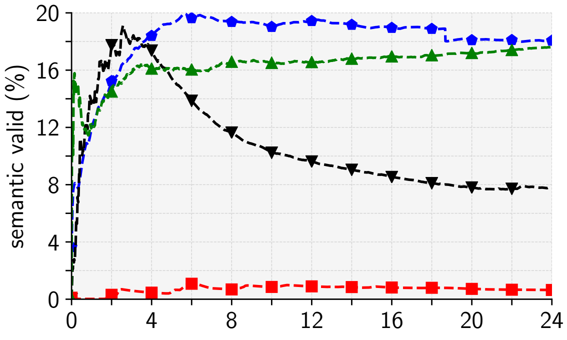
(h) `PostgreSQL` semantic



(i) `MySQL` new edges



(j) `MySQL` syntax



(k) `MySQL` semantic

Figure 3.10: **Contributions of Validity and Feedback.** Figures a - k show the bugs numbers of fuzzing `SQLite`, new edges, the syntax correctness, and the semantic correctness for fuzzing each DBMS.

Considering that the latter runs 220 queries per second while the former can execute 507 (i.e., 1.3× faster), we believe the advantage of Squirrel$^{!syntax}_{!semantic}$ over Squirrel$_{!semantic}$ mainly comes from the faster generation speed. Squirrel$_{!feedback}$ takes 700 minutes to find the first crash. The total number of unique crashes follows the same pattern, where Squirrel, Squirrel$_{!semantic}$, Squirrel$^{!syntax}_{!semantic}$ and Squirrel$_{!feedback}$ detect 600, 30, 10 and 3 crashes, respectively. The results above show that all three factors of Squirrel contribute critically to crash detection. Further, coverage-based feedback plays the most important role, while syntax-only testing cannot beat `AFL`.

**Unique Bugs.** We adopt the same strategy to measure the unique bugs, where we patch the detected bug after each hour. Figure 3.10b shows the results. The full-featured SQUIRREL finds nine unique bugs, while other variants (`!feedback` and `!semantic`) only detect one unique bug which is covered by SQUIRREL.

**New Edges.** Figure 3.10c, f and i demonstrate an (almost) consistent pattern of finding new edges for `SQLite`, `PostgreSQL` and `MySQL`: SQUIRREL > SQUIRREL > SQUIRREL$^{!syntax}_{!semantic}$ >SQUIRREL$_{!feedback}$. The coverage-based feedback helps achieve 2.0× more new edges for fuzzing `SQLite`, `PostgreSQL` and `MySQL`. Additionally, the syntax-correct mutation enables SQUIRREL to find 1.0×-1.5× more edges than `AFL`, while semantic-guided instantiation improves the number by 0.3×-1.7×. Therefore, improving the syntax or semantic correctness of queries helps to achieve more diverse states of the DBMS. These results are presented in Figure 3.10.

**Syntax Validity and Semantic Validity.** Figure 3.10d, g and j show the syntax changes during the tests of three DBMSs, while Figure 3.10e, h and k show the semantic changes. The validity of SQUIRREL is generally the highest, while `AFL` achieves the lowest. This result is reasonable because SQUIRREL is specifically designed to improve language validity, while `AFL` randomly mutates SQL queries. However, the figures contain some interesting anomalies. First, we notice that in Figure 3.10d and g, SQUIRREL$_{!semantic}$ achieves syntax accuracy that is comparable to SQUIRREL. This suggests that enhancing semantic correctness does not necessarily lead to an increase in syntax correctness. However, since it removes shorter queries, the instantiation procedure can reduce syntax correctness, as seen in Figure 3.10j. Although shorter queries are more likely to be syntactically correct, our instantiator cannot correct their semantics. For example, `SELECT a FROM b` is semantically incorrect as no table `b` exists. As `SQLsmith` performs similarly or even better than SQUIRREL, the p-values for `PostgreSQL` and `MySQL` are larger than 0.05.

The second anomaly is observed in Figure 3.10j, where SQUIRREL$_{!feedback}$ achieves similar semantic correctness as SQUIRREL, suggesting that feedback has no impact on `MySQL`'s semantic correctness. However, further analysis reveals that SQUIRREL$_{!feedback}$ produces highly divergent semantic correctness results, with two of the five experiments achieving over 40% correctness and the other three with less than 10We find that the initial seeds in `MySQL` are smaller than those in `SQLite` and `PostgreSQL`. The smaller initial seeds in `MySQL` may cause SQUIRREL$_{!feedback}$ to generate correct but simple and repetitive inputs. Although the p-values in Table 3.4 are larger than 0.05 (caused by the randomness of `MySQL` results), Figure 3.10i shows that Squirrel generates more diverse query structures than Squirrel$_{\mathit{!feedback}}$ and finds more execution paths with similar semantic correctness.

In summary, Squirrel benefits from syntax, semantics, and feedback to discover more memory errors in DBMSs. Among these factors, coverage-based feedback has the most significant impact, while syntax and semantics have varying effects. The interplay of all three factors determines the final result.

# Chapter 4 | Generic Language Processor Testing with Semantic Validation

In this chapter, we propose PolyGlot, a generic fuzzing framework that generates high-quality test cases for exploring processors of different programming languages.

## 4.1 Introduction

Language processors [132], such as compilers and interpreters, are crucial components in modern software development. They are responsible for converting programs written in high-level languages to machine code that can be executed by the hardware. Ensuring the correctness of language processors is essential as it guarantees that the compiled code is semantically consistent with the source code. Buggy language processors can translate even correct programs to malfunctional codes, which might lead to security problems. For instance, a miscompilation of memory-safe programs can result in memory-unsafe binaries [24, 49], leaving the system vulnerable to security breaches. Similarly, vulnerabilities in interpreters can lead to denial-of-service (DoS) attacks, sandbox escape, or remote code execution (RCE) [93, 103, 145], providing attackers with unauthorized access to the system. Furthermore, these defects can propagate to other translated programs, including other language processors, resulting in a chain reaction of vulnerabilities [117].

However, traditional software testing methods are often inadequate for identifying bugs in language processors due to the strict requirements for input syntax and semantic validity. This makes it challenging to automatically test and identify errors in the translation logic, because even minor errors in the input code can cause the language processor to terminate execution, making it difficult to explore deeper into the translation process.

Recent works on software testing, such as grey-box fuzzing, try to meet these requirements to effectively test language processors [9, 40, 42, 62, 143, 144]. Originally, structure-unaware mutation [40, 143, 144] can hardly generate syntax-correct test cases; advanced fuzzers [50, 56] 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 [41, 84, 106], 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 [93, 141, 149].

However, when a fuzzer is highly customized for a specific language, it loses its ability to be applied generically. As a result, users cannot use the same specialized fuzzer to test a different programming language, and they have to develop a new one from scratch. However, considering the large number (over 700 currently [130]) of programming languages and the complexity (e.g., `CSmith` [141] consists of 80k lines of code) of language-specific fuzzers, which can consist of tens of thousands of lines of code, it is not feasible to implement a specific fuzzer for each language. Therefore, current fuzzers face a dilemma: if they prioritize high semantic validity, they sacrifice their generic applicability, and if they maintain their generic applicability, they cannot ensure the quality of the test cases.

In this thesis, we introduce PolyGlot, a fuzzing framework designed to produce *semantically valid* test cases that thoroughly evaluate processors of *different* programming languages. To achieve versatility, we implement a uniform intermediate representation (IR) that mitigates syntax and semantics disparities between programming languages. Given the BNF grammar of a language, PolyGlot generates a frontend that translates source code into the IR. Moreover, users can provide semantic annotations that describe scopes and types of definitions within the language, such as functions, defined variables, and composite types. In this thesis, 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 between 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 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.

The implementation of PolyGlot consists of approximately 7,016 lines of C++ and Python code, which primarily handle intermediate representation (IR) generation, constrained mutation, and semantic validation. At present, PolyGlot can handle 9 programming languages, and it can be easily adapted to support additional languages.

We evaluate the effectiveness of PolyGlot by testing it on 21 popular compilers and interpreters for 9 programming languages, resulting in the discovery of 173 previously unknown bugs. As of the writing of this thesis, 113 of these bugs have been addressed with 18 assigned CVEs. Our experiments indicate that PolyGlot is superior to state-of-the-art general-purpose fuzzers, such as `AFL`, `QSYM`, and `Nautilus`, in generating high-quality test cases (with up to a 100× improvement in language validity), exploring program states (with up to a 30× increase in new paths discovered), and identifying vulnerabilities (with 8× more unique bugs detected). Furthermore, we compare PolyGlot with language-specific testing tools, such as `CSmith` for C and `DIE` for `JavaScript`, and our results show that PolyGlot is more effective in exploring program states.

## 4.2 Problem

This section commences with a brief overview of how language processors deal with input programs, along with the ways in which syntax and semantic errors can halt this process.

Subsequently, we outline the restrictions and difficulties faced by current fuzzing tools when it comes to testing language processors. We then summarize the typical semantic errors that arise in test cases generated by fuzzers. Finally, we present our approach to address this issue.

## 4.2.1 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.

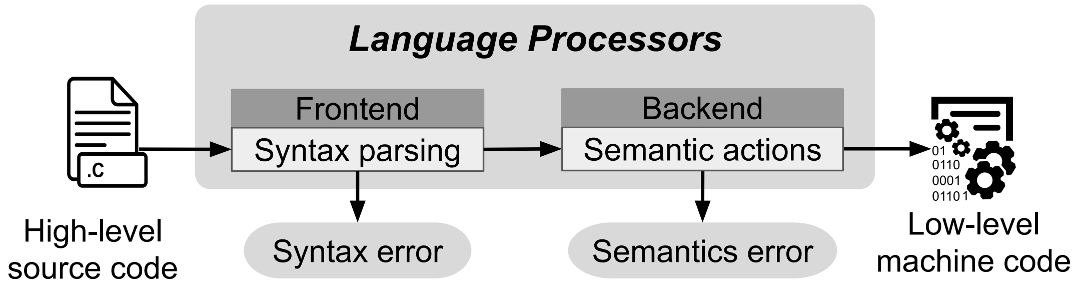


Figure 4.1: **Workflow of language processors**. Given a high-level source-code program, a language processor checks it for syntactic and semantic errors. If none, the processor translates the program into low-level machine code.

Language processors perform syntax and semantic checks on the input program, and any errors can lead to the termination of program execution. The process of language processing is illustrated in Figure 4.1. At an early stage of processing, the frontend checks for syntax errors. Afterward, the backend checks for semantic errors, which cannot be detected by the parser. Only programs that are semantically correct can be considered valid and processed successfully.

We show a C program in Figure 4.2a and a `JavaScript` program in Figure 4.2b. If we add the statements that start with "+", we introduce errors in the program. For example, line 6 in Figure 4.2a and line 6 in Figure 4.2b introduce syntax errors, and the parser detects these errors and bails out. Line 7–8 in Figure 4.2a and line 11–12 in Figure 4.2b contain semantic errors which will be caught by the backend optimizer or translator.

```
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

```
1    function opt(x){return x[1];}
2    let arr = [1, 2];
3
4    if(arr[0]) {let arr2=[1, 2]};
5    // ig is a wrong keyword
6  + ig(arr[0]) {let arr2=[1, 2]};
7    arr[1] += "1234";
8
9    for(let idx=0; idx<100; idx++)
10     opt(arr);
11 + for(let idx=0; idx<100; idx++)
12 +   opt(arr2); // undef var
```

(b) An example `JavaScript` program

Figure 4.2: **Running examples.** Figure 4.2a 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, Figure 4.2b shows a program written in `JavaScript`, a dynamically typed language, which allows more type conversion.

### 4.2.2 Limitations of Current Fuzzers

Fuzzing is a popular technique for discovering software bugs and vulnerabilities [42,44,144], but current fuzzing has limitations in testing language processors. General-purpose mutation-based fuzzers [16, 40, 143, 144] randomly flip input bits or bytes, without considering input structure, which makes it difficult for them to generate syntactically correct inputs. To address this limitation, recent works use higher-level mutations in AST or IR to ensure syntactic correctness [56, 125]. On the other hand, generation-based fuzzers [84, 106] generate inputs based on a model or grammar, which allows them to create structurally correct inputs more effectively than random bitflip mutation-based fuzzers. While they have shown advantages in passing syntactic checks, they often dismiss the semantic correctness of generated test cases, which may result in missed bugs in the optimization or execution code of language processors. We conducted a preliminary experiment to investigate the impact of semantic correctness on fuzzing effectiveness. By compiling the code snippet in Figure 4.2a with the optimization flag "-O3" in gcc-10, we observed that it covers 56,725 branches. In contrast, the invalid variants of the code that start with "+" only trigger less than 27,000 branches, as they are rejected during or right after parsing due to their syntactic errors.

Researchers have attempted to improve the semantic correctness of their fuzzers through various methods [50, 93, 141, 149]. CSmith [141] performs heavy analysis to generate valid C programs without undefined behaviors. `JavaScript` fuzzers [50, 93] take into account the expression types to avoid generating semantically incorrect test cases. Similarly, SQUIRREL constructs the data dependency of SQL to produce valid queries for testing DBMSs. Unfortunately, these methods are highly specific to a particular programming language, which means that users must expend significant effort to apply them to new programming languages. This can be impractical and time-consuming, particularly considering the numerous real-world programming languages [130].

Recent language-based fuzzers [3, 56] try to generate correct test cases for different languages. LangFuzz [56] replaces every variable in the mutated code randomly while `Nautilus` [3] 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 that allow more implicit type conversion, such as `JavaScript` and PHP.

### 4.2.3 Common Semantic Errors

We manually analyzed 1,500 invalid test cases generated by existing fuzzers for different programming languages [3, 84, 93] and identified four common types of semantic errors. Two of them are related to the scope of variables and functions while the other two relate 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 Figure 4.2a 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 Figure 4.2b, 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 introduce 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 Figure 4.2a introduces an error. Line 7 of Figure 4.2b is correct because in `JavaScript` numbers can convert to strings.

### 4.2.4 Our Approach

We aim to create a generic fuzzing framework that generates semantically correct inputs to test different language processors. We separate the goal into two steps. First, we convert the programming languages into a uniform IR which allows us to perform standardized mutation and analysis independent of the underlying programming language's syntax and semantics. 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.** To neutralize differences in syntax and semantics among programming languages, we design an intermediate representation (IR) that maps language-specific features 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 contains IR statements and preserves the source program's syntactic structures, facilitating easy translation back to the original source. Users can describe language scopes and types on top of target programming language by using a simple annotation format, which is encoded into the IR's semantic properties. With the IR, we can perform mutation or analysis regardless of the language's underlying grammar and semantics.

**Improving Language Validity.** We improve the language validity with *constrained mutation* and *semantic validation*. Our mutation approach is designed to preserve two critical aspects of the program: the syntactic correctness of the entire program and the semantic correctness of the unmutated part. First, we preserve the syntactic correctness of the test case by mutating the IRs based on their IR types that correspond to the underlying grammar structures. For instance, when replacing an `IF` statement (in the form of IR), we substitute it with another `IF` statement instead of a function call expression. Second, our constrained mutation approach involves mutating only IRs with *local effects* to maintain the semantic correctness of the unmutated code. These IRs do not contain any definitions or introduce new local scopes. An example of such an IR is line 7 in the JavaScript code in Figure 4.2b, which only uses the variable `arr` without any 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

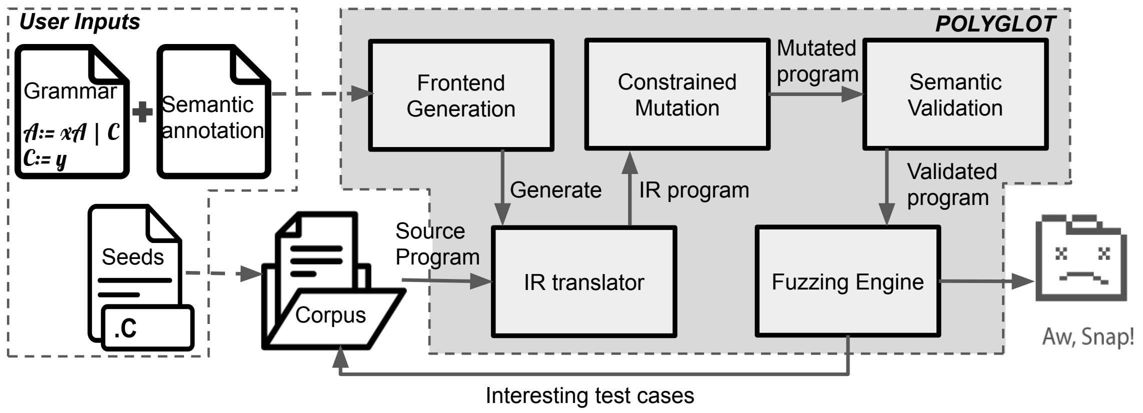


Figure 4.3: **Overview of PolyGlot.** PolyGlot aims to discover bugs that crash language processors. PolyGlot accepts the BNF grammar, semantic annotations, and seeds from users as input. First, the frontend generator generates an IR translator that converts a source program to an IR program. Second, 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.

has potential semantic errors in the mutated part, which might use invalid variables. To fix these errors systematically, 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 properly replace variables' invalid usages. Afterward, the validated test cases should be correct and are helpful for thoroughly fuzzing language processors.

## 4.3 Overview of PolyGlot

Figure 4.3 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 (Section 4.4). 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 (Section 4.5). Afterward, the semantic validator tries to fix the semantic errors in the new IR programs (Section 4.6). 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.

## 4.4 Frontend Generation

To ensure our approach is applicable across multiple programming languages, we generate a translator via our frontend generation. This translator converts a source program into an IR program, thereby lowering the level of mutation and analysis from language-specific source code to a uniform IR.

In Figure 4.4, we show the IR (Figure 4.4a) of a simple C program to demonstrate how the BNF grammar (Figure 4.4b) and semantic annotations (Figure 4.4c) 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 provide users with predefined semantic properties for describing variable types and scopes. The generated IRs will carry these properties as described in the annotations (e.g.,, `ir9` has property `FunctionDefinition`). Users can easily utilize the BNF grammar and semantic annotations to describe the syntax and semantics of a programming language in a specific manner.

### 4.4.1 Intermediate Representation

Our IR is a uniform representation that captures both the syntax and semantics of the source program. It consists of an order, a type, an operator, a value, up to two operands, and a list of semantic properties. The IR order corresponds to the statement order in the source code, while the type corresponds to the symbol in the BNF grammar. The IR operator and the IR value store the information of 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.** Any modifications on IRs will not break the syntactic structures as yyntactic structures retain all the grammar information of the source program. As we see earlier, certain IR statements store a small piece of the source code (e.g.,, a function name is stored in an IR of type `FuncName`). Also, the IR statements are connected in a

```
//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<Ret, 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;}".

```
<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 Figure 4.4b. It is in JSON format.

Figure 4.4: **An example IR program with its corresponding BNF grammar and semantic annotations.** The IRs in Figure 4.4a have corresponding symbols in the BNF grammar in Figure 4.4b. The semantic annotations in Figure 4.4c describe what semantic properties the symbols in the BNF should have, which will be reflected in the semantic_property in IRs.

directed way that forms a tree view of the source program. By performing an in-order traversal of 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 Figure 4.4a has the semantic property `FunctionDefinition`, indicating that it relates to a function definition. Line 14-15 in Figure 4.4c 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`.

### 4.4.2 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.

## 4.5 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 (Section 4.5.1) and the semantic correctness of its unmutated part (Section 4.5.2). 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.

### 4.5.1 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.

### 4.5.2 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 Figure 4.2b 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 Figure 4.2b 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.

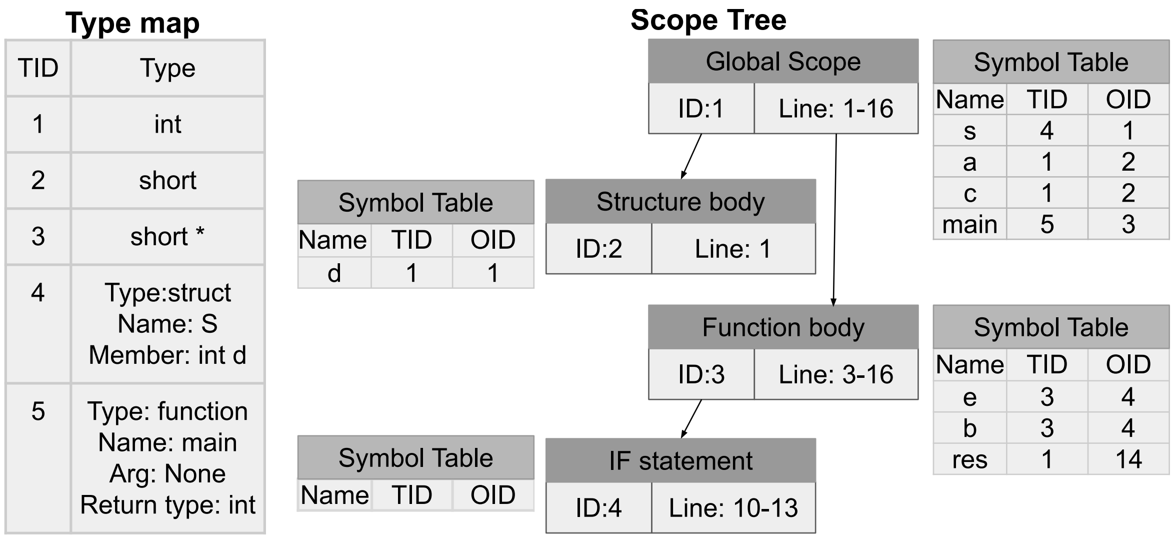
With these two rules, our constrained mutation produces syntax-correct test cases. These test cases might contain semantic errors. According to local mutation, the semantic errors are introduced by the mutated part, which might use invalid variables from other test cases. Next, we will fix all these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

## 4.6 Semantic Validation

```c
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3] = {1, 2, 0};
  e = b;
  do{
    c += *(e++);
  } while(*e);
  int res = c;
  return res;
}
```

```c
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3] = {1, 2, 0};
  e = b;
  if(FIXME >= FIXME){
  //struct X z;
    FIXME += FIXME;
  }
  int res = c;
  return res;
}
```

(a) Left: A pre-mutate program. Right: A mutated program that needs to be validated



Type map

| TID | Type |
|---|---|
| 1 | int |
| 2 | short |
| 3 | short * |
| 4 | Type:struct Name: S Member: int d |
| 5 | Type: function Name: main Arg: None Return type: int |

Symbol Table (Global Scope)

| Name | TID | OID |
|---|---|---|
| s | 4 | 1 |
| a | 1 | 2 |
| c | 1 | 2 |
| main | 5 | 3 |

Symbol Table (Structure body)

| Name | TID | OID |
|---|---|---|
| d | 1 | 1 |

Symbol Table (Function body)

| Name | TID | OID |
|---|---|---|
| e | 3 | 4 |
| b | 3 | 4 |
| res | 1 | 14 |

Symbol Table (IF statement)

| Name | TID | OID |
|---|---|---|

(b) The type map, scope tree and its the symbol tables of the program in Figure 4.5a

Figure 4.5: **A mutated variant of Figure 4.2a and its semantic information collected by the semantic validator.** As shown in Figure 4.5b, the type map contains the types used in the program while the scope tree and symbol table indicate the definition and lifetime of a variable.

In the second step towards language validity, we address semantic errors in the mutated portion of the test case by replacing invalid variables with valid ones. To do this, we must first determine the available variables within the proper scope, to avoid using undefined or out-of-scope variables. Additionally, we need to ensure the appropriate use of these variables by matching their types, avoiding the use of variables with unmatched or undefined types.

Therefore, our semantic validation relies on two components: *type system* that collects type information of variables (Section 4.6.1) and *scope system* that collects scope information (Section 4.6.2). After collecting the necessary information about types, scopes, and names of every definition in the test case, we integrate it into the symbol tables. These symbol tables are then utilized by the semantic validation process to replace the invalid expressions with correct ones, resulting in a semantically valid test case, as shown in Section 4.6.3.

### 4.6.1 Type System

In programming languages, types include predefined ones such as `int` in C, and user-defined ones such as `class` in `JavaScript` [131]. 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, Figure 4.5a and Figure 4.5b 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 of PolyGlot can infer the type of a variable or an expression, and we handle different cases for each of them. 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 (Section 4.6.2), 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 operators, we first recursively infer the types of the operands as they are also expressions. As discussed in Section 4.4.1, 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 Figure 4.2b 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.

### 4.6.2 Scope System

The visibility of variables within a program is determined by the scope system, which partitions the program into different scopes. Variables gain visibility automatically based on the scope in which they are declared. 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.

Figure 4.5b shows the constructed scope tree of Figure 4.5a. "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).

Figure 4.5b shows symbol tables of each scope for Figure 4.5a. 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.

### 4.6.3 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 Figure 4.5a.

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 Figure 4.5a, 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 Figure 4.5a, `"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 Figure 4.5a. 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 Figure 4.5a, 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.

One possible solution to fix `FIXME >= FIXME` at line 11 in Figure 4.5a 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,

Table 4.1: **Lines of code on 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** |

`b[1]` and `s.d` can be compared by `>=` though they are of different types.

## 4.7 Implementation

We implement PolyGlot with 7,016 lines of code. Table 4.1 shows the breakdown.

**Frontend Generation.** We extend the IR format proposed in [149] by introducing semantic properties that are generated based on user-provided annotations 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 [33] and Flex [28] to create a parser with the parsing methods, and these components are compiled together to form 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 processing the children of the IR, the system pops the scope from the stack to complete the partitioning of 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 Figure 4.5b, 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 [2]. 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 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.

## 4.8 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? (Section 4.8.2)

Table 4.2: **21 compliers and interpreters of 9 programming languages tested by PolyGlot.** # refers to the TIOBE index, a measurement of the popularity for programming languages [118], 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** |

- Can semantic validation improve PolyGlot's fuzzing effectiveness? (Section 4.8.3)
- Can PolyGlot outperform state-of-the-art fuzzers? (Section 4.8.4)

### 4.8.1 Evaluation Setup

**Benchmark.** To evaluate the genericity of PolyGlot, we test 21 popular processors of nine programming languages according to their popularity [118] 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 4.2. To understand the contributions of our semantic validation, we perform an in-depth evaluation on the representative processors of four popular languages (two statically typed and two dynamically typed):

```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

Figure 4.6: **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".

`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 [83] and [67] 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 [2]. The official release of `Nautilus` only 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 [144] or AFL-QEMU mode to instrument the tested applications. We enlarge the bitmap to 1024K-bytes to mitigate path collisions [35]. 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 to calculate the P-values for the experiments and provide the result in Table 4.4.

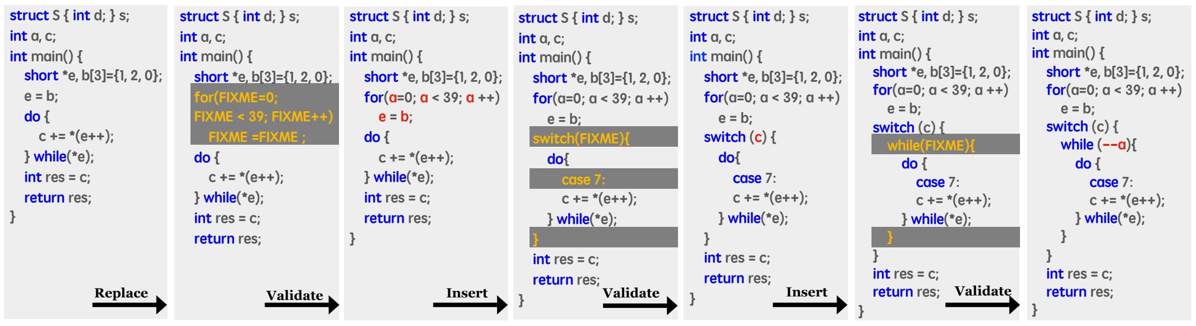
### 4.8.2 Generic Applicability and Identified Bugs

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 Section 4.7, 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 4.2, 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. All the bugs have been reported to and acknowledged by the corresponding developers. At the time of this thesis writing, 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`. Figure 4.6a shows the Proof-of-Concept (PoC), and Figure 4.6b shows its logically equivalent program for understanding the semantics of the PoC easily.

Figure 4.7 shows the process of how PolyGlot turns the benign motivating example (Figure 4.2a) into the bug triggering PoC. After each round of mutation, all the definitions



```
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3]={1, 2, 0};
  e = b;
  do {
    c += *(e++);
  } while(*e);
  int res = c;
  return res;
}
```

```
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3]={1, 2, 0};
  for(FIXME=0;
  FIXME < 39; FIXME++)
    FIXME =FIXME ;
  do {
    c += *(e++);
  } while(*e);
  int res = c;
  return res;
}
```

```
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3]={1, 2, 0};
  for(a=0; a < 39; a ++)
    e = b;
  do {
    c += *(e++);
  } while(*e);
  int res = c;
  return res;
}
```

```
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3]={1, 2, 0};
  for(a=0; a < 39; a ++)
    e = b;
  switch(FIXME){
  do{
    case 7:
      c += *(e++);
    } while(*e);
  }
  int res = c;
  return res;
}
```

```
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3]={1, 2, 0};
  for(a=0; a < 39; a ++)
    e = b;
  switch (c) {
    do{
      case 7:
        c += *(e++);
      } while(*e);
  }
  int res = c;
  return res;
}
```

```
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3]={1, 2, 0};
  for(a=0; a < 39; a ++)
    e = b;
  switch (c) {
    while(FIXME){
      do {
        case 7:
          c += *(e++);
        } while(*e);
    }
  }
  int res = c;
  return res;
}
```

```
struct S { int d; } s;
int a, c;
int main() {
  short *e, b[3]={1, 2, 0};
  for(a=0; a < 39; a ++)
    e = b;
  switch (c) {
    while (--a){
      do {
        case 7:
          c += *(e++);
        } while(*e);
    }
  }
  int res = c;
  return res;
}
```

Figure 4.7: **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 Figure 4.2a. 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.

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 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.

```
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);
```

Figure 4.8: **PoC that hijacks control flow in njs.** njs crashes with `RIP` hitting `0xdeadbeef`. The bug is assigned with `CVE-2020-24349`.

**Case Study 2: Control Flow Hijacking in njs.** PolyGlot identified many exploitable bugs, including the one shown in Figure 4.8, 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`

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

Figure 4.9: **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.

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 Figure 4.9. 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 [91, 133], causing memory corruption and leading to sandbox escape [6, 48, 82].

We show the PoC of one of our bugs in Figure 4.9, 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 [44]. Therefore, our bugs in PHP interpreters, though not assigned with CVEs, can lead to severe security consequences.

### 4.8.3 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

Table 4.3: **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<br>SF: segmentation fault<br>AF: assertion failure<br>UAF: use-after-free<br>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 | ✓ | ✗ | ✗ | ✗ | ✗ | - | - |

number of unique bugs as it can better reflect bug finding capabilities of the fuzzers than the number of unique crashes [62]. 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 (Section 4.8.4). **Unique Bugs.** We manually map each crash found by each fuzzer in 24 hours to its corresponding bug, and show the result in Table 4.3. 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

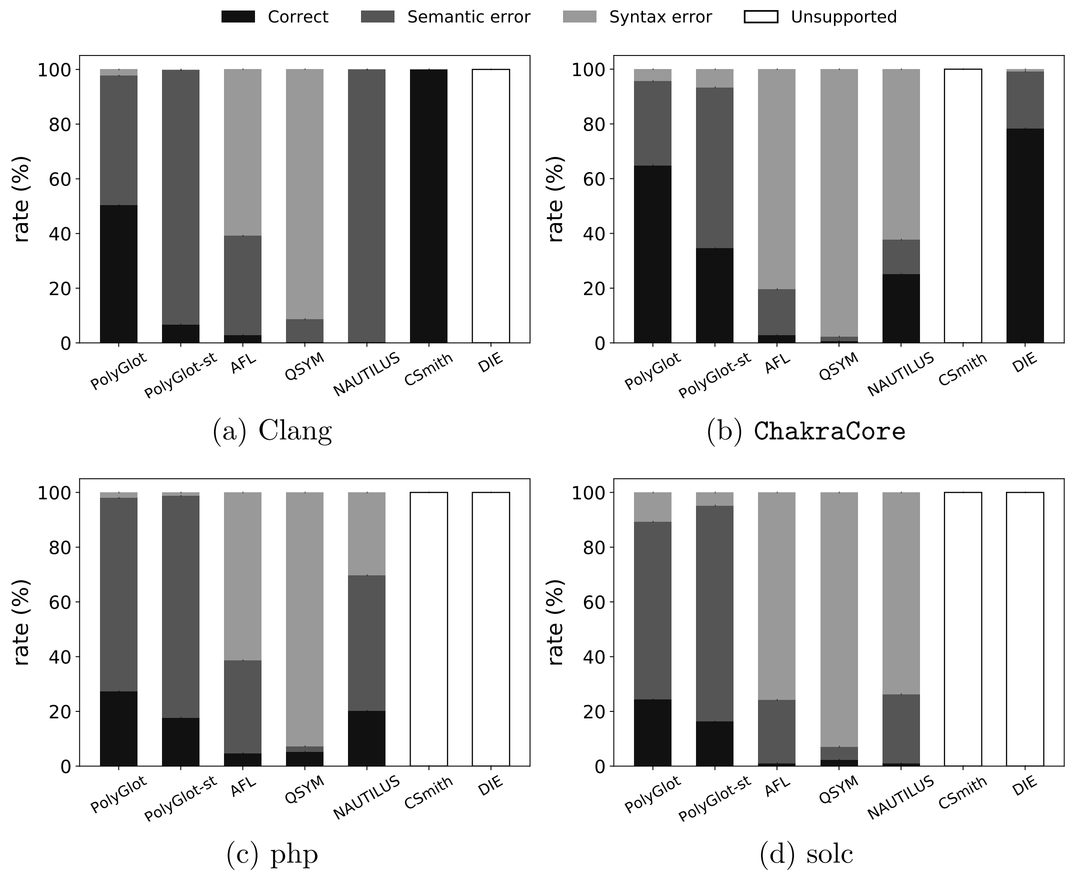


Figure 4.10: **Rate of language correctness of inputs generated by evaluated fuzzers for 24h.** "Correct" means the inputs can be successfully executed or compiled by the language processors. "Syntax error" means the inputs contain syntactic errors. "Semantic error" means the inputs are valid syntactically but not semantically. "Unsupported" mean the fuzzer is not applicable to the target.

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 Figure 4.10. 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`

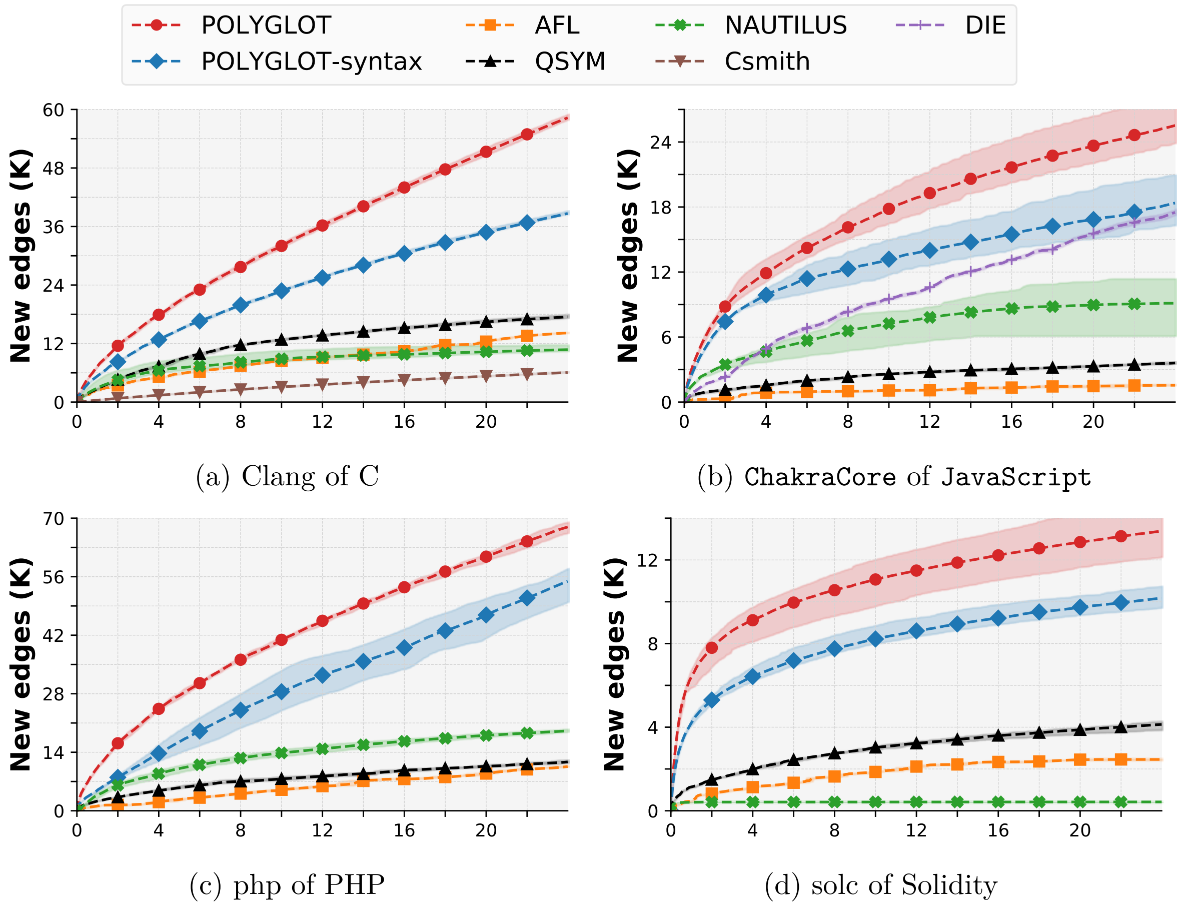


Figure 4.11: **Edge coverage found by evaluated fuzzers for 24h.** We repeat the experiments 5 times. The solid dot lines represent the mean of the result and the shadow around lines are confidence intervals for five runs.

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 Figure 4.11. The increase is higher in `Clang` and `ChakraCore` than the 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 4.5, 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.

### 4.8.4 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 4.3. `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 (Figure 4.6a), 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 Figure 4.10. 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

Table 4.4: **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 |

specific language and thus achieve higher validity in that language.

**Code Coverage against General-purpose Fuzzers.** As shown in Figure 4.11, 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% more in `solc`. If we look at the execution speed of `AFL`, `QSYM`, and `Nautilus` Table 4.5, 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.

Table 4.5: **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 | - | - |

**Code Coverage against Language-specific Fuzzers.** As shown in Figure 4.11, 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 (Figure 4.10). We analyze the results and find the following reasons. First, both `CSmith` and `DIE` execute more slowly than PolyGlot Table 4.5. 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.

# Chapter 5 Redesign Parallel Fuzzing using Microservice Architecture

In this chapter, we redesign parallel fuzzing with microservice architecture and propose the prototype $\mu$FUZZ.

## 5.1 Introduction

In recent years, fuzzing has become a popular technique for detecting security bugs in software testing [10, 50, 141, 144]. Compared to other program analysis techniques, fuzzing is efficient and requires less manual effort and prior knowledge of the target software. Its effectiveness in detecting security issues in complex and real-world programs has been demonstrated [42, 144]. This has led to a significant investment in fuzzing by many companies, such as Google's Clusterfuzz project [43], which has found over 36,000 bugs.

Researchers have proposed various optimizations to improve the internal components [66, 87] of fuzzing, including grammar-based, adaptive or unified mutators [27, 75, 134, 149], heavy program analysis techniques [9, 97, 101, 114, 143], priority adjustment algorithms [12, 124, 146], and feedback mechanisms [31, 88, 123]. The aim is to increase the performance of a single fuzzing instance.

Parallel fuzzing is another area of focus, aimed at making full use of resources and detecting more bugs in a shorter time. State-of-the-art parallel fuzzing approaches launch multiple fuzzing instances in separate processes and periodically synchronize their progress. Each instance follows the logic of a single-instance fuzzer, with its own fuzzing states and inputs, but benefits from the synchronization with other instances in exploring the program state space for bugs. Since parallel fuzzing shows its ability to shorten time cost of bug detection, companies such as Google have deployed parallel fuzzing infrastructure for continuous integration and continuous deployment (CI/CD) testing [45].

However, two limitations exist that prevent scalability in the present parallel fuzzing architecture. First, the existing architecture is based on single-instance fuzzers whose fuzzing logic may not be well suited for parallel fuzzing. These single-instance fuzzers have a serial, synchronous loop where input generation and consumption must be done in order. If a procedure (such as test case execution) within the loop becomes blocked, for example due to file reading and writing, the whole instance is hindered. The CPU assigned to the fuzzing instance will be idle until the blocking operation is completed. With parallel fuzzing, multiple instances are run simultaneously and the synchronization process increases I/O, making instances more likely to become stuck and leading to a waste of CPU power. To fully utilize the computational power, we need to reuse idle CPU. However, simply launching more instances to handle the idle CPU, such as 32 instances on a 16-core machine, results in frequent context switches between the fuzzing instances, which decreases the fuzzing performance.

The second limitation in the current parallel fuzzing architecture is the inefficiency of synchronization. The synchronization process periodically updates the local fuzzing states of each instance to ensure they have the latest information to make globally optimal choices. However, the updates are not timely enough, meaning that each instance must rely on local information to make decisions during the time between two consecutive synchronizations. This can result in suboptimal decisions and reduced fuzzing efficacy over time. Increasing the frequency of synchronization could address this issue, but it would also come with heavy overhead and decreased fuzzing efficiency, as demonstrated in previous research [139].

To overcome the limitations of the current parallel fuzzing architecture, it is necessary to redesign the fuzzing tools to minimize the impacts of synchronization and serialization. A solution is found in the *microservice architecture* [102], which divides tasks into separate, independently functional services that can run concurrently. This allows for computing resources to be efficiently utilized, as CPU can switch to other services when one service is blocked. Additionally, each service retains its own state and only communicates minimal information with other services, allowing them to make globally optimal decisions most of the time.

We introduce $\mu$FUZZ, a parallel fuzzing framework that utilizes microservice architecture, to achieve our goals. To implement this new architecture, we divided the existing serial fuzzing loop into four microservices: corpus management, test case generation, test case execution, and feedback collection. Each microservice can run independently and schedule parallel workers. This effectively addresses the CPU idling issue since the microservices are loosely coupled, so they can replace the blocked service for execution, thus reducing idle time. Specially, our framework allows each microservice to complete its task as much as possible before switching to another one to minimize context switch overhead. Additionally, we implement an output cache mechanism to reduce the coupling between the services (e.g., test case generation and consumption services), allowing them to continue making progress even if one gets stuck. This enables consumer services to retrieve results from the cache if the producer service is blocked, allowing them to make progress even when a service is stuck.

To address the issue of untimely synchronization, $\mu$FUZZ utilizes two levels of state partitioning. First, the global state is divided among the microservices in $\mu$FUZZ according to their functionalities which allow each service to handle its states locally. For instance, the feedback collection service is responsible for evaluating code coverage and updating the coverage bitmap, so states of coverage bitmap will be stored in it. Second, within each service, individual workers handle separate parts of the service state, and the combined state of all workers constitutes the overall state of the service. These partitions eliminate the state synchronization among different services and workers and enable each service worker to use up-to-date information to make globally optimal decisions.

We implement $\mu$FUZZ in 9,534 lines of Rust code, which consists of the concurrent infrastructure (i.e.,, the asynchronous runtime) and the fuzzer. The concurrent infrastructure is built on top of Tokio [119], a well-tested asynchronous runtime library. For the fuzzer, we adopt the fork-server execution, havoc mutation, and edge coverage feedback from `AFLplusplus`-4.01c, 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 $\mu$FUZZ on popular benchmarks, including Magma [53] and FuzzBench [78]. We compared $\mu$FUZZ with the state-of-the-art parallel fuzzers, including `AFLplusplus`, `AFLEdge` and `AFLTeam`, and found $\mu$FUZZ can explore 57% more program states and 67% more bugs on average in 24 hours. Besides, our experiments show that $\mu$FUZZ can make progress in the existence of blocking I/O with its concurrent design, and its state partition helps improve code coverage by 96% and bug detection by 47%. Moreover, we evaluated $\mu$FUZZ and found three new bugs on well-tested real-world programs.

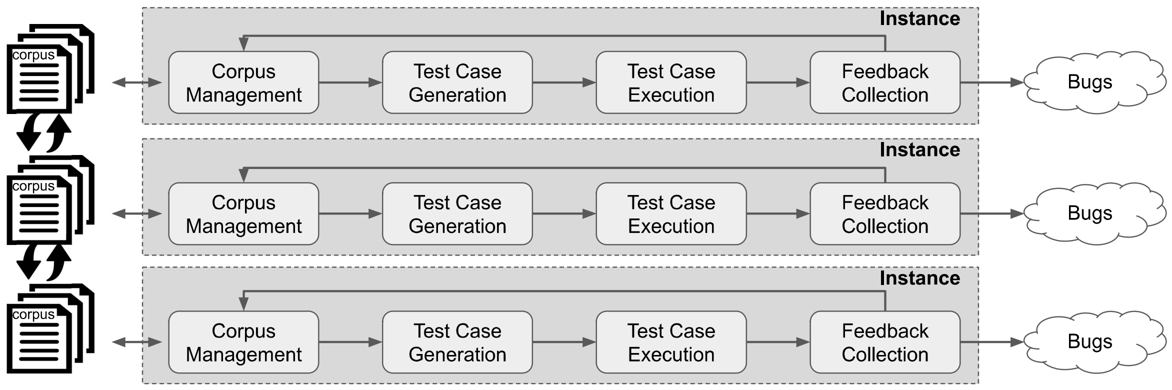


Figure 5.1: **The state-of-the-art parallel fuzzing approach.** The fuzzer spawns multiple instances and runs them in parallel. Each instance is self-contained and functionality-complete. They maintain their own local fuzzing states, such as the corpus and coverage bitmap. Most of the time, the instances work independently as if there were no other instances. Occasionally, the instances perform corpus synchronization with each other to share their fuzzing progress.

## 5.2 Problem

In this section, we provide a brief overview of the current state-of-the-art parallel fuzzing methods. We then discuss the limitations of these methods and how a microservice architecture can address these limitations. Finally, we give out our novel approach to addressing the problem.

### 5.2.1 How Existing Parallel Fuzzing Works

To improve testing of complex programs under time constraints [45,63], many fuzzers [16, 40, 54, 73, 144] support parallel fuzzing to enhance performance. In state-of-the-art approach, multiple instances of the same fuzzer run independently on multiple CPU cores and periodically synchronize the corpus (representing the fuzzing progress) with each other to catch up on the latest progress made by one instance. As shown in Figure 5.1, each instance maintains a local seed corpus. Each instance runs independently most of the time, only periodically checking and copying seeds from other instances that trigger new code.

Advanced parallel fuzzing approaches either run instances of different fuzzers to combine their capability [20,51,92] or further optimize the corpus distribution strategy by partitioning the synchronized corpus among instances to avoid duplicated fuzzing efforts [69,96,129].

**Fuzzing State.** Synchronizing the corpus can enhance fuzzing because the corpus is a crucial component of the fuzzing state, which represents the minimum information required to depict the overall fuzzing progress of an instance. This state may include the corpus, the average execution time of the test case, the seeds of the random number generator, and other relevant details.

### 5.2.2 Limitation of Existing Approaches

Existing parallel fuzzing approach involves running multiple fuzzing instances on different CPU cores, each maintaining its own local state. The instances periodically synchronize the state to catch up on each other's progress. However, this approach has two weaknesses: (1). It aggravates the problem of CPU idling due to the serial design of the underlying fuzzer. (2). It leads to suboptimal performance as the global fuzzing state cannot be synchronized to each instance efficiently and in a timely manner.

**CPU Idling due to Blocking I/O.** Existing fuzzers organize their components in a serial synchronous loop [40, 54, 144]. The fuzzing pipeline, such as in `AFLplusplus`, involves a sequence of steps (selecting a test case, mutating it, executing it, and checking the feedback) that run in a serial manner. If any of these steps, particularly the execution phase, is blocked by input/output (I/O) operations, the CPU cannot perform other tasks, leading to idle time. Therefore, such a design might suffer from performance degradation in the existence of I/O. I/O can come from two sources. First, the tested program involves 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 tasks, such as test case mutation, but wait, resulting in the CPU idling. For example, we measured the CPU usage of fuzzing `tcpdump` with `AFLplusplus` and found that the CPU usage was only 70%. We checked the system calls made by `tcpdump` using `strace` [29] and found that `tcpdump` was waiting for blocking system calls such as `poll` to return. Second, synchronizing the state between multiple instances of a fuzzer during parallel fuzzing can result in heavy file I/O and negatively impact performance. For example, in the case of `AFLplusplus`, each instance periodically checks and updates its corpus with others in a shared folder, which can lead to excessive file locking and copying and thus decrease the performance [139]. While adding more instances to the same CPU (i.e., oversubscription) may seem like a solution, it can lead to higher CPU usage but not necessarily better performance due to resource contention and excessive context switching

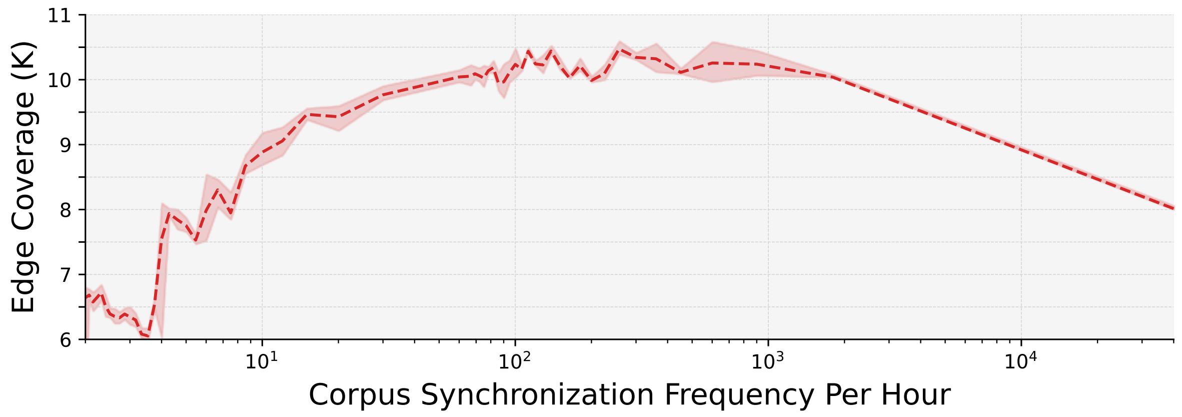


Figure 5.2: **Code coverage of 10 `AFLplusplus` instances testing `QuickJS` with different synchronization frequency in an hour.**

without a properly calibrated task scheduler.

**Fuzzing State Not Timely Synchronized.** The instances maintain their local fuzzing states and perform periodic synchronization. During the time between two synchronizations, they continue to fuzz using the potentially outdated states and a locally optimal strategy, which may not be globally optimal. On the other hand, too frequent synchronization introduces a high overhead which can result another negative impact on performance [139].

We did a quick experiment to verify our hypothesis. The experiment used `AFLplusplus` to fuzz `QuickJS` [71], a popular JavaScript engine, using OSS-Fuzz benchmark [42]. The experiment compared fuzzing with a single instance for ten hours and ten instances in parallel for one hour The metrics measured were edge coverage and the number of interesting test cases selected for fuzzing. The results showed that when fuzzing with a single instance for 10 hours, about 13,700 new program paths were discovered and 80% of the interesting test cases were further used for fuzzing. However, when fuzzing with ten instances in parallel for 1 hour, only 6,700 new program paths were discovered, which was 49% of the coverage 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. It is believed that the decreased performance of the parallel instances is due to duplication of work on similar test cases and use of suboptimal strategies. Similar results are also found in [129].

To determine if the performance gap was caused by synchronization delay, we varied the synchronization frequency of `AFLplusplus` while measuring the effect on code coverage. More specifically, we ran `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 Figure 5.2. When the frequency was too low (i.e., untimely synchronization), the code coverage was also low, while when it was high, the code coverage dropped significantly due to the high overhead of synchronization. However, even at the best frequency, the performance was still much worse than that of single instance fuzzing. This indicates that changing the frequency of synchronization alone is insufficient to address the issue.

From the above discussion, we want a parallel fuzzing framework that supports concurrency to effectively leverages the CPU power and performs timely state synchronization with minimal overhead to ensure optimal fuzzing strategy. However, this is not feasible with the current monolithic serial approach of existing fuzzers, and a different architecture is necessary.

### 5.2.3 Microservice Architecture

We find microservice architecture [102] fits parallel fuzzing well and can potentially mitigate its current limitations. First, the microservice architecture organizes applications into a collection of small, independent services that work collaboratively and concurrently. In order to facilitate parallel fuzzing, we can break down the serial fuzzing loop and put them into these concurrent services, and thereby we can 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) if each of them focuses on a single complete functionality of fuzzing. Thus, there is no need for state synchronization between 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 fuzzing state it 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 globally optimal result.

### 5.2.4 Our Approach

We aims to design a parallel fuzzing framework that leverages concurrency to maximize CPU utilization, even with the presence of blocking I/O, while still maintaining optimal performance. This is accomplished through two steps: redesigning the fuzzing framework with microservice architecture migration and partitioning the fuzzing state. The microservice architecture brings concurrency to the framework, allowing it to fully utilize CPU power in the existence of I/O. Partitioning the fuzzing state allows each instance to use a locally optimal strategy while still achieving globally optimal performance without the need for synchronization.

**Redesign with Microservice Architecture.** We divide the traditional serial fuzzing process into four distinct services, each responsible for a specific task: corpus management, test case generation, test case execution, and feedback collection. The fuzzing state is also partitioned into these services, so that each service only requires its specific partition to make decisions, and thus can be self-contained. Although these services are interdependent, with each producing outputs for the others to consume and vice versa, we use output caching to decouple the production and consumption processes. This allows the services to run concurrently, even when one service is blocked by I/O. The cached outputs can be directly consumed by the stuck service when it becomes available, which allows us to more effectively utilize CPU power in the presence of blocking I/O.

**Partition the Fuzzing State.** The first level of fuzzing state partition has been achieved by decomposing the monolithic structure. As a result, each service now maintains its own individual fuzzing state. However, if the state is shared by multiple workers, state synchronization among them becomes necessary. To avoid this, we further partition the state among the workers within each service. We use two guiding principles in our partitioning strategy. Firstly, each partition of the state should be complete in terms of functionality, which means the worker should be able to perform its task without relying on the states of other workers. Secondly, by each worker adopting its locally optimal strategy, it is expected that the accumulation of individual results will lead to a globally optimal outcome. This enables the workers to operate independently without the need for synchronization. With only one global and distributive fuzzing state, changes to the state are directly applied to the workers' states. As a result, workers always fuzz using the most up-to-date global state and make the optimal decisions. This eliminates the issues associated with periodic synchronization, including high overhead or significant synchronization lag. For instance, the feedback collection service manages a bitmap for measuring code coverage. Instead of maintaining a single, global bitmap, we have partitioned the bitmap into multiple sub-bitmaps, with each worker managing a unique sub-bitmap. When new feedback is received from the execution service, it is sent to each worker, who only needs to examine the part of the feedback relevant to its sub-bitmap, without synchronizing to other workers.

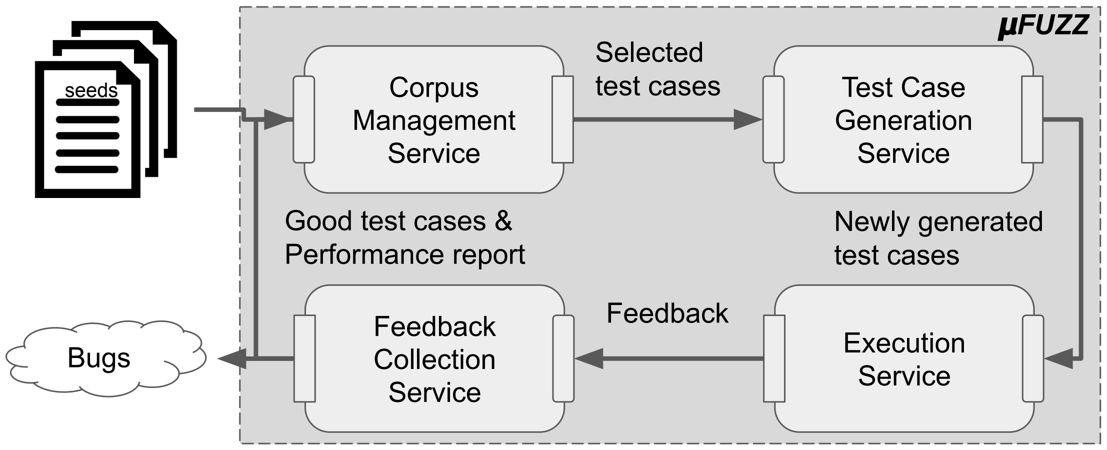

μFUZZ
seeds
Corpus Management Service
Selected test cases
Test Case Generation Service
Good test cases & Performance report
Newly generated test cases
Feedback Collection Service
Feedback
Execution Service
Bugs

Figure 5.3: **Overview of $\mu$Fuzz.** Instead of running multiple fuzzing instances and performing periodic synchronization, $\mu$FUZZ breaks the traditional monolithic architecture into a microservice one. 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. As described in Figure 5.4, inside each service, we run multiple workers to exploit parallelism.

## 5.3 Design

The overview of $\mu$FUZZ is presented in Figure 5.3. We first break down the traditional serial fuzzing loop into four services (Section 5.3.1). This division separates the fuzzer's responsibility and state into different services, so it eliminates the need for state synchronization between them. These self-contained services are the candidates for concurrency. Next, we utilize output caching to allow concurrent running of services (Section 5.3.2) while achieving maximum parallelism with load balancing (Section 5.3.3). Then, we partition the fuzzing state among workers to avoid synchronization while still getting an optimal result (Section 5.3.4). Finally, we connect the services together with zero-copy communication (Section 5.3.5) to achieve efficient parallel fuzzing.

### 5.3.1 From Monolith to Microservice

As the first step to support concurrency and reduce CPU idling due to blocking I/O, we break the monolithic serial fuzzing loop into multiple services. We use the following guidelines from the microservice architecture for the breakdown. First, each service should be micro and focus only on one core functionality of the fuzzing. 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

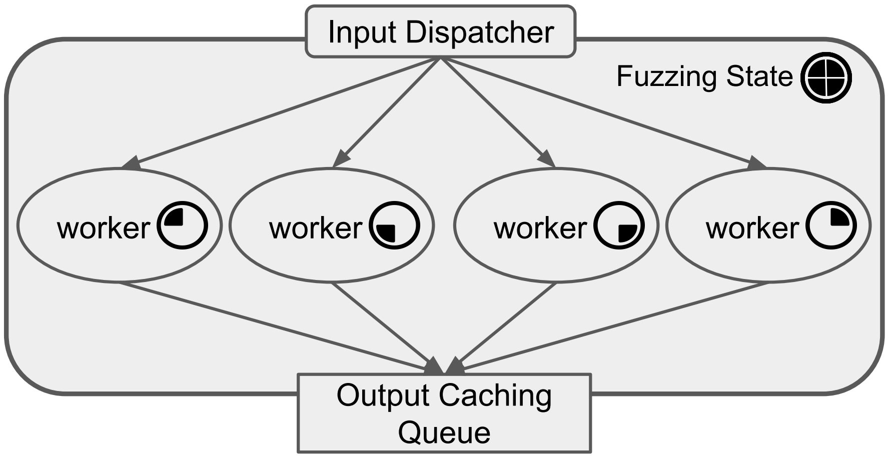


Figure 5.4: **The internal structure of a service in $\mu$Fuzz.** Each service has a fuzzing state, an input queue dispatcher, an output caching queue, and some workers. The fuzzing state is partitioned into the workers. The input queue dispatcher accepts from other services and dispatches them to the workers. The workers handle the inputs in parallel and send the results to the output caching queue. The consumer services can fetch these results whenever they are ready.

be maintained by the service. As a result, we classify four core functionalities from the fuzzing loop and 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 those finding new code coverage and those that trigger new bugs, etc. Based on the metadata of the test cases, the scheduling algorithm chooses some from the corpus that can help test case generation.

**Test Case Generation Service.** It produces new test cases to fuzz the target program, either by starting from scratch or by modifying existing ones. It can use the BNF grammar to generate structured inputs or employ bit-flipping techniques to generate new variants from existing test cases.

**Execution Service.** The execution service runs the generated test cases on the target program and collects relevant feedback, such as code coverage, crashes, and timeouts.

**Feedback Collection Service.** It collects the feedback from the execution service and classifies whether they are 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 good 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. These services still need to run in a serial way due to once one service gets stuck, the overall progress stops. To address this issue and reduce CPU idling, we need to decouple the services further, as explained in the next section.

### 5.3.2 Concurrency by Output Caching

We enable the concurrent operation of services that both produce and consume data by implementing an output caching queue between them. This decouples the production and consumption of each service, allowing them to run independently. When a service generates results, it first sends them to the output queue instead of directly to the consumer service. If the consumer service is temporarily occupied, the producer service can continue generating more results that are cached in the queue. Once the consumer service becomes available again, it can retrieve the cached results directly from the output queue. This approach not only enables multiple services to run concurrently but also allows others to continue making progress even if one service is temporarily blocked.

**Congestion Control.** Output caching can result in the overproduction of outputs by a producer service, which can cause the service to monopolize CPU cores and prevent other services from consuming the cached outputs. 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 since all the CPU cores are busy running the corpus management service.

To prevent unlimited output generation and ensure that other services have the opportunity to run, we implement congestion control by setting a maximum limit on the number of cached results in the queue. When the output queue is full, the producer service will pause and allow other services to run. This keeps a dynamic balance between production and consumption and enables the fuzzing process to progress smoothly.

### 5.3.3 Parallelism by Load Balancing

To fully utilize the computation power of multiple cores, each service of $\mu$Fuzz can have multiple workers in parallel. To achieve maximum parallelism, the number of workers should be the same as 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 partitioning. We define a worker as idle if it is ready to process input but not currently processing any. Such workers notify the input dispatcher to put them into the back of the queue in order. Whenever an input arrives, the input dispatcher tries to pop an idle worker out 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 partition before dispatching. If the arrived input is larger than a threshold value and there are more than one idle workers, we partition the input evenly based on the number of idle workers and dispatch each partition to an idle worker. With the two strategies, we can achieve maximum parallelism by keeping the workload of each worker balanced dynamically.

### 5.3.4 Avoid Synchronization by State Partition

As the result in Figure 5.2 shows, if the fuzzing instances maintain their local states and rely on periodic synchronization, we end up with suboptimal performance due to either synchronization lagging or high overhead. Therefore, we avoid synchronization by partitioning the fuzzing state. Afterward, we maintain only one global state but in a distributive way.

We already perform the first level of state partition by breaking the fuzzing loop into microservices. We further perform the second level inside each service. More specifically, we partition the state among the workers in a way that each worker maintains a unique and functionality-complete partition. Every worker only needs its own part of the state to finish its job. The workers do not need state synchronization with each other. Besides, the partition enables us to achieve an overall optimal result by simply accumulating the results from the workers. In this way, the workers can run independently.

We categorize the fuzzing states into two types. The first type has a known fixed size before the fuzzing starts. For example, the feedback collection service maintains a bitmap for recording code coverage. The size of the bitmap is fixed and configured by the users. The second type has an unknown variable size. For example, the seed corpus is part of the fuzzing state and grows as we fuzz. Due to these different characteristics, we perform static partition for the first type and dynamic partition for the second, as described below.

**Static State Partition.** For the fuzzing state of fixed size, we partition it statically and evenly among the workers. Therefore, we know which worker maintains which partition in advance. When new inputs arrive, the input dispatcher partition the inputs based on the partition boundary and dispatch them to the corresponding workers. 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 maintains byte 0 to byte 99, and the second maintains byte 100 to 199). When the execution service generates new bitmap information, the information is partitioned and dispatched to different workers in the feedback collection service. By accumulation, if none of the workers in the feedback collection service find new bits, the executed test case is considered uninteresting and otherwise interesting.

**Dynamic State Partition with Tagging.** For the fuzzing state of variable size, we cannot predict the size in advance to perform static partition. Instead, we use dynamic partition: whenever a new part of the state is generated, we partition it evenly and distribute it to the workers. Since the new state is randomly distributed, the partition in each worker should have similar data distribution as the global state. This is important because most fuzzing strategies are randomized ones, which means their performance is not dependent on the size or the specific values of the underlying data but on their distribution. Therefore, a fuzzing strategy that is optimal for the global state should also be optimal for the partitioned state. For example, if the corpus management service has 25 workers and maintains a corpus of 1000 interesting test cases. 500 of the test cases are considered as good (i.e.,, they can potentially trigger more new codes), and the other 500 are bad. By uniformly random partition, each worker is expected to maintain 20 good test cases and 20 bad test cases. If the test case selection strategy for the global state is to first explore the good test cases and then the bad ones, then every worker just independently follows this strategy to achieve the same expected result.

One problem with dynamic partition is that there is no partition boundary. It is difficult for the input dispatcher to figure out how to dispatch the inputs. For example, the feedback collection service can send some performance reports of the evaluated test cases to the corpus management (e.g., the test case X generates 20 new test cases that trigger new code coverage) for fine-tuning the scheduling algorithm. The input dispatcher of the corpus management service cannot figure out which worker maintains the test case

X and should receive such inputs easily. If we maintain the global knowledge of which worker maintains which test cases, it will be too much overhead as the corpus size grows. Therefore, for dynamic state partition, we assign each worker a unique ID and tag the states it maintains with the ID. Such tagged IDs will remain in all intermediate outputs that are related to the states (i.e.,, the test cases sent out by the corpus management will contain the IDs of the workers maintaining them.). Since the number of workers can be known in advance, the input dispatcher can dispatch the inputs based on the ID.

### 5.3.5 Zero-Copy Communication

As mentioned before, we break down the fuzzing loop into several services that consume and produce data for each other. However, this can cause high communication overhead due to the large amount of data being passed. To reduce this overhead, we have designed a safe zero-copy mechanism that uses shared memory and pointer passing. This mechanism ensures safe access to data across services and requires only a constant size of data to be passed, regardless of the amount of generated output. For example, passing a pointer to 1,000 bytes of generated data only requires eight bytes on an x64 system, rather than copying the entire data.

**Pointer Passing with Shared Memory.** Instead of asking both the producer and consumer service to unnecessarily allocate memory to store and copy the data from one to the other, we create shared memory between the services and pass the pointers to the shared memory. After the shared memory is set up, 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.** Shared memory can introduce a safety issue since multiple services can access it simultaneously, leading to race conditions. To ensure safe memory access, we wrap the pointers with unique ownership. This ensures that only one service can access the underlying shared memory at a time. This is reasonable since the consumer should only access the output after the producer has finished generating it, and the producer has no need to access its output afterward.

Table 5.1: **Line of codes of different components of $\mu$Fuzz, which sum up to 9,534 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** |

## 5.4 Implementation

We implement $\mu$FUZZ in 9,534 lines of code. Table 5.1 shows the breakdown.

**Concurrent Runtime.** We use `Tokio` as the concurrent runtime of $\mu$FUZZ. The runtime is responsible for efficient task scheduling. Each worker in the services of $\mu$FUZZ is run as a task in the runtime. As users configure the number of workers, the number of total tasks is fixed. In this way, we avoid the overhead of unnecessary task creation. We maintain a double-ended 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 $\mu$FUZZ will try to put them 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. To avoid unnecessary service switching, when a service receives inputs, it processes as many of them as possible. If all the inputs are processed or the service gets stuck, $\mu$FUZZ 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.** $\mu$FUZZ 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, considering the fuzzing speed. Instead, we mutate each test case multiple times and send the new variants in bulk to the execution service to reduce the overhead. We also skip the deterministic stage and only perform the havoc stage as `AFLplusplus`'s parallel fuzzing mode does.

**Execution.** The execution service adopts the popular fork-server approach [54, 144]. Each worker in the execution service has its own fork server. When a worker receives an input to execute, it feeds the input into the fork server and requests a 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 $\mu$Fuzz in the same process to share the address space. 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, which means only one owner can operate on the underlying memory.

## 5.5 Evaluation

Our evaluation aims to answer the following questions.

- Can $\mu$Fuzz outperform state-of-the-art parallel fuzzers? (Section 5.5.2)
- Can $\mu$Fuzz's microservice architecture and state partition improve fuzzing performance? (Section 5.5.3)
- Can $\mu$Fuzz find new bugs in real-world programs? (Section 5.5.4)

### 5.5.1 Evaluation Setup

**Benchmark.** We use the state-of-the-art benchmark `Magma` [53] to evaluate $\mu$Fuzz. The measured metrics include bug detection capability and code coverage. Due to the time and resource limit, we rank the programs in `Magma` by the number of inserted bugs and test the top six programs: `Poppler`, `SQLite`, `openssl`, `sndfile`,`libxml2`, `PHP`. We use the corpus from `Magma` for all the targets and run them through `AFLplusplus`'s test case minimizers to remove redundant ones. We compare $\mu$Fuzz with three state-of-the-art fuzzers: `AFLplusplus` [54], `AFLTeam` [96], and `AFLEdge` [129]. `AFLplusplus` is the most popular fork of AFL with various improvements and is still 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 good comparison for $\mu$FUZZ's state partition. AFLEdge and AFLTeam work by integrating with existing single-instance fuzzers. Therefore, we run AFLTeam and AFLEdge on top of AFLplusplus for a fair comparison.

**Environment Setup.** We perform our evaluation on three machines, each with an Ubuntu 18.04 operating system, an Intel Xeon CPU E5-2680 v3 processor with 48 visual cores and 256GB memory. For bug detection, we calculate the number of bugs with Magma, which assigns a unique bug ID to all its inserted bugs and prints a log whenever a bug is triggered. Due to the randomness in fuzzing, we further apply Magma's survival analysis to convert bug triggering time to bug survival time, which is the expected time a bug remains undiscovered. A smaller survival time indicates a fuzzer can find the bug in shorter time. We instrument the tested programs to test edge coverage with hit counters. For the code coverage and bug detection experiments, we run the fuzzers with 40 fuzzing instances on 40 cores inside docker for 24 hours and repeat the process five times. For $\mu$FUZZ, we run 40 workers for each service but still use only 40 cores which is the same as the other fuzzers. We report the average results to reduce the random noise.

### 5.5.2 Comparison against existing fuzzers

We compare $\mu$FUZZ against three state-of-the-art fuzzers to understand its strengths and weaknesses in parallel fuzzing, including the *de facto* AFLplusplus and the two most recent parallel fuzzers, AFLEdge and AFLTeam. Since $\mu$FUZZ's seed scheduling algorithm is different from AFLplusplus's, we also compare with AFLplusplus-M, which is AFLplusplus with $\mu$FUZZ's seed scheduling algorithm, to see whether performance improvement is due to our seed scheduling. We also want to understand whether increasing the synchronization frequency of existing fuzzers can improve fuzzing as a simple solution. Therefore, we add a comparison with AFLplusplus-F, which is AFLplusplus performing synchronization every 30 seconds instead of 30 minutes. We use 30 seconds because we find by experiments that it has lower overhead and achieves almost the same coverage as that of a shorter synchronization interval. We compare these fuzzers in two metrics: bug detection (the number of triggered bugs and their survival time) and edge coverage.

**Bug Detection.** As shown in Table 5.2, $\mu$FUZZ finds 12 bugs in 24 hours, while AFLplusplus, AFLEdge, AFLTeam, AFLplusplus-M, and AFLplusplus-F find only 8, 8, 6,

Table 5.2: **Bug Detection Results in 24 Hours.**

| **Targets** | **Bug ID** | $\mu$FUZZ | AFLplusplus | AFLEdge | AFLTeam | AFLplusplus-F |
|---|---|---|---|---|---|---|
| Poppler | PDF010 | 12h56m | 19h18m | 22h13m | **05h31m** | 13h13m |
| | PDF016 | **01m40s** | 16m40s | **01m40s** | 16m40s | **01m40s** |
| | PDF021 | **05h22m** | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| sndfile | SND017 | 01h24m | 03h34m | 02h51m | 02h46m | 16m45s |
| | SND020 | **01h24m** | 04h15m | 02h51m | 02h46m | 02h50m |
| libxml2 | XML002 | **12h57m** | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| | XML003 | **02h46m** | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| | XML009 | **16m40s** | 02h46m | 02h46m | 02h46m | 02h46m |
| | XML012 | **15h13m** | $\infty$ | $\infty$ | $\infty$ | 20h45m |
| | XML017 | **01m40s** | 16m40s | **01m40s** | **01m40s** | **01m40s** |
| SQLite | SQL002 | **02h46m** | 10h34m | 07h28m | $\infty$ | 03h16m |
| | SQL018 | **02h46m** | 02h52m | 05h14m | $\infty$ | 03h40m |
| **Total Bugs Found** | | **12** | **8** | **8** | **6** | **9** |

| **Targets** | **Bug ID** | AFLplusplus-M | $\mu$FUZZ-S | $\mu$FUZZ-SM |
|---|---|---|---|---|
| Poppler | PDF010 | 18h19m | $\infty$ | $\infty$ |
| | PDF016 | **01m40s** | 03h21m | 03h25m |
| | PDF021 | $\infty$ | $\infty$ | $\infty$ |
| sndfile | SND017 | **16m40s** | 02h46m | 01h25m |
| | SND020 | 02h50m | 02h46m | 01h25m |
| libxml2 | XML002 | $\infty$ | $\infty$ | $\infty$ |
| | XML003 | $\infty$ | $\infty$ | $\infty$ |
| | XML009 | 02h46m | 03h06m | 03h31m |
| | XML012 | $\infty$ | 20h07m | 22h38m |
| | XML017 | 16m40s | 02h48m | 02h46m |
| SQLite | SQL002 | 07h53m | 06h19m | 09h14m |
| | SQL018 | $\infty$ | 03h33m | 07h10m |
| **Total Bugs Found** | | **7** | **8** | **8** |

7, and 9 bugs, respectively. All the nine bugs found by other fuzzers are also covered by $\mu$FUZZ, and $\mu$FUZZ found seven of them within the shortest time. Three of the bugs (PDF201, XML002, XML003) are only found by $\mu$FUZZ. AFLplusplus finds one more bug than AFLplusplus-M, meaning that the seed scheduling of $\mu$FUZZ might not be as good as AFLplusplus's in bug detection. However, $\mu$FUZZ can still find more bugs quickly, thanks to the microservice design and state partition. We plan to adopt the advanced seed scheduling algorithm of AFLplusplus to further improve the performance

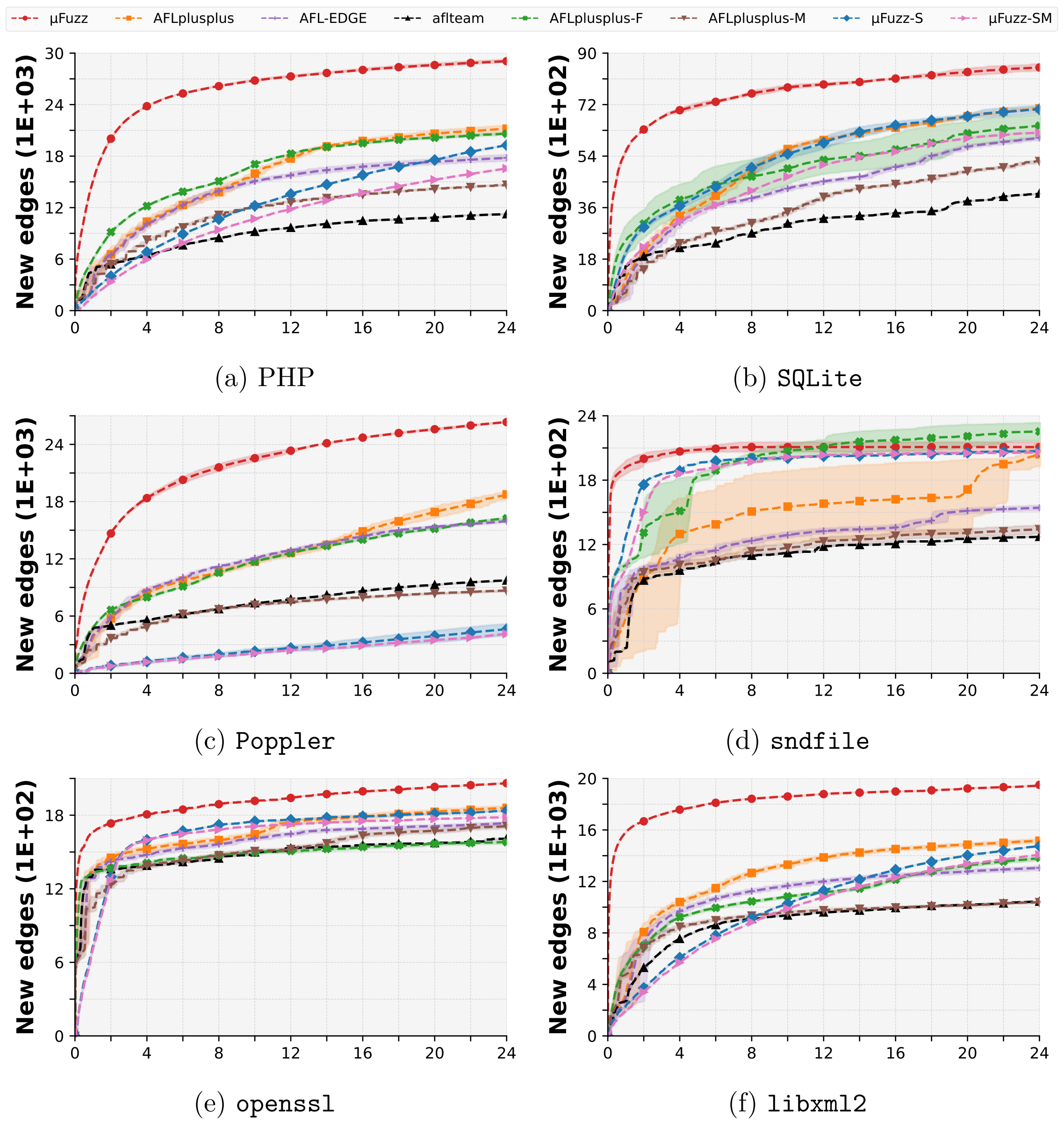


Figure 5.5: **Edge coverage found by evaluated fuzzers with 40 cores for 24h.** AFLplusplus-F and AFLplusplus-M are `AFLplusplus` with a synchronization per 30 seconds and $\mu$Fuzz seed scheduling, respectively. $\mu$Fuzz-S removes state partitioning from $\mu$Fuzz, and $\mu$Fuzz-SM further removes concurrency.

of $\mu$Fuzz. Additionally, AFLplusplus-F finds all the eight bugs found by `AFLplusplus` and one more. Out of the 8 common bugs, AFLplusplus-F finds six of them faster. This shows that higher synchronization does help improve the bug detection capability of `AFLplusplus` because the fuzzing instances can catch up with the latest progress earlier.

**Code Coverage.** As shown in Figure 5.5, on average $\mu$Fuzz identifies 23%, 45%, 102%,

Table 5.3: **Line of source codes and the number of inserted bugs of the tested six targets.** Larger code spaces usually result in more complex programs.

| **Target** | `Poppler` | `SQLite` | `openssl` | `sndfile` | `libxml2` | `PHP` |
|---|---|---|---|---|---|---|
| **LOC** | 342K | 320K | 695K | 66K | 457K | 1,488K |
| **Bug Num** | 22 | 20 | 20 | 18 | 17 | 16 |

88%, and 33% more new edges than `AFLplusplus`, `AFLEdge`, `AFLTeam`, AFLplusplus-M, and AFLplusplus-F respectively on the six targets. We can see from Table 5.3 that the improvement of $\mu$Fuzz against the second-best fuzzer is related to the complexity of the targets. If the programs have larger code bases and more program states to explore , $\mu$Fuzz can achieve much higher code coverage than the second-best fuzzer (e.g., 40% more in `Poppler` and 37% more in `PHP` than `AFLplusplus`). Otherwise, if the programs are small and fuzzers can achieve saturated coverage in a short time, then the improvement of $\mu$Fuzz is not apparent (10% more in `openssl` than `AFLplusplus` and 6% less in `sndfile` than AFLplusplus-F). Another thing we see is that `AFLplusplus` has higher code coverage than AFLplusplus-M, which shows that the simple scheduling algorithm of $\mu$Fuzz is not as good as `AFLplusplus`'s in input space exploration. This means the improvement of $\mu$Fuzz does not come from its scheduling algorithm but its new architecture and state partition. We notice that `AFLEdge` and `AFLTeam` have worse performance than `AFLplusplus`. We carefully investigated their source code and execution status and found that they both run their partition algorithms at a fixed time interval (i.e., every hour). The algorithms aggregate the fuzzing progress of all their instances (e.g., the corpus of all instances) and then perform partitioning, which requires heavy analysis. However, since we are running the experiment with 40 instances and the total size of the corpus is large, the analysis can take 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 and the partition results might not be optimal anymore. Interestingly, we notice that the coverage of AFLplusplus-F 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 input space is not well explored. An instance might find a bunch of interesting test cases, but cannot explore all of them timely. Under this situation, faster synchronization allows other instances to catch up with the progress and help explore the interesting test cases. A small corpus also allow the instances to synchronize with low overhead. However, as the code coverage is about to be saturated, there are not as frequent progress updates as in the beginning and AFLplusplus-F still synchronizes frequently. Every time an instance wants to synchronize with another's corpus, it has to walk through the directory to see whether there are any new test cases. Since the corpus has grown bigger, such operations become more expensive, so the fuzzing speed of AFLplusplus-F goes down, resulting in a slower increase in code coverage. If we further check the bug triggering time in Table 5.2, we can see that seven out of the 9 bugs found by AFLplusplus-F are within the first four hours. This is when faster synchronization is still beneficial for fuzzing.

Overall, $\mu$Fuzz outperforms the three compared parallel fuzzers `AFLplusplus`, `AFLEdge`, `AFLTeam`, and their variants in both bug detection and code coverage. The fuzzing effectiveness of $\mu$Fuzz comes from both its concurrent design and state partition.

### 5.5.3 Contribution of Microservice Architecture and State Partition

We compare $\mu$Fuzz with $\mu$Fuzz-S, which is $\mu$Fuzz with blocking synchronization but without state partition, to understand their contributions in parallel fuzzing. Since $\mu$Fuzz-S introduces blocking I/O, we further compare it with $\mu$Fuzz-SM, which is $\mu$Fuzz-S without concurrency and runs services synchronously. More specifically, every worker in $\mu$Fuzz-S maintains a copy of the global state. Whenever there are state updates, $\mu$Fuzz-S will propagate the updates to all its workers synchronously. The workers cannot handle new inputs until the state update is finished.

**Bug Detection.** $\mu$Fuzz successfully identifies 12 bugs in the targets in 24 hours, while both $\mu$Fuzz-S and $\mu$Fuzz-SM finds only 8. As shown in Table 5.2, $\mu$Fuzz finds all the bugs that are found by $\mu$Fuzz-S and $\mu$Fuzz-SM much faster, spending 46.6% and 53.7% less of the time respectively. Although $\mu$Fuzz-S and $\mu$Fuzz-SM found the same eight bugs, $\mu$Fuzz-S found five of them faster. We checked the fuzzing speed of these fuzzers to investigate the cause of the performance gap. We found that although $\mu$Fuzz-S has blocking synchronization, its fuzzing speed has no significant difference from that of $\mu$Fuzz, which has no blocking synchronization. This is because the concurrency allows $\mu$Fuzz-S to run other services during the state updates; thus its overall progress will not be blocked. However, the fuzzing speed of $\mu$Fuzz-SM is only 91% of $\mu$Fuzz-S on average. This means that in $\mu$Fuzz-S, when some services get blocked during synchronization, the other services cannot take over the CPU and utilize the computation power. Therefore, we can have two conclusions: First, with concurrency, the fuzzer continues to make progress in the existence of blocking I/O and thus well utilizes the CPU power. Second, with state partition, $\mu$FUZZ can find more bugs faster.

**Code Coverage.** As shown in Figure 5.5, $\mu$FUZZ finds 96% and 117% more edge than $\mu$FUZZ-S and $\mu$FUZZ-SM respectively in the six targets on average. From Figure 5.5, we find that the coverage increase of $\mu$FUZZ is much faster than $\mu$FUZZ-S until the coverage is saturated. We checked the execution status and found that workers in the corpus management service of $\mu$FUZZ-S tend to select duplicated test cases for mutation. This is because the workers have the exact same state as each other and use the same scheduling algorithm. Such duplication can slow down fuzzers' exploration. $\mu$FUZZ-SM has the worst performance because it not only suffers from the aforementioned duplication but also slower fuzzing speed due to blocking I/O.

Overall, $\mu$FUZZ outperforms $\mu$FUZZ-S and $\mu$FUZZ-SM in both bug detection and code coverage. Both the microservice architecture and state partition contribute to $\mu$FUZZ's strengths. The concurrent design allows $\mu$FUZZ to make progress in the existence of blocking I/O. State partition allows all the workers to work independently and still achieves an overall optimal result.

### 5.5.4 Identified New Bugs

Since we do not propose any new fuzzing strategy but help the existing strategies parallel more efficiently, we only ran some preliminary experiments to fuzz some programs from OSS-Fuzz for a short time. Yet we still find three new bugs in these well-tested programs, showing that $\mu$FUZZ 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 three identified bugs include one logical error and two memory-corruption, of which one has been fixed and one acknowledged by the developers at the time of writing. We are confident in $\mu$FUZZ since it has better performance than existing fuzzers according to our evaluation.

# Chapter 6 Discussion

In this chapter, we discuss several limitations of our presented works and provide potential solutions for them.

## 6.1 Squirrel

**DBMS-Specific Logic.** Although our design of Squirrel is DBMS-agnostic, we find that utilizing program-specific features improves testing result. First, each DBMS implements a dialect of SQL, which is either almost the same as the official one, like `SQLite` [55], or significantly different in many features, like `PostgreSQL` [99]. Due to this reason, Squirrel works well on `SQLite` (51 bugs), but only triggers few bugs in `PostgreSQL`, `MySQL` and `MariaDB`. We can implement more accurate grammar of different SQL dialects to improve our fuzzing efficacy. Second, DBMSs may adopt extra checks before executing the query. For example, we find that `PostgreSQL` requires type correctness between all operands, and comparisons between integers and floating points are not allowed. `SQLite` does not check anything but will automatically perform type-casting during the execution, while `MySQL` only gives a warning for mismatched types. We can implement the type consistency relation in our semantics-guided instantiation for testing `PostgreSQL`.

**Alternative Feedback Mechanisms.** Recent software testing practice widely adopts code coverage to guide mutation-based fuzzing [40, 73, 144]. However, during our evaluation, we find potentially harmful code coverage that hinders the generation of semantics-correct queries. Especially, at the beginning of testing grammar-incorrect queries trigger many new branches in the fault-handling code. The coverage-based feedback guides Squirrel to focus on these inputs instead of the original semantics-correct queries. Recent works for fuzzing language compilers and interpreters mention a similar observation [52, 137]. We can investigate this problem and develop solutions to mitigate it, like dropping inputs that trigger new branches in short executions.

## 6.2 PolyGlot

**Inconsistent Grammar.** PolyGlot generates test cases according to the provided grammar. However, it still generates syntactically incorrect test cases because a BNF grammar is usually a superset of the real grammar that DBMSs and 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 [46, 47]. Alternatively, we can try machine learning techniques to infer the accurate input grammar from test cases [38, 72, 135].

**Specific format BNF grammar.** Currently, PolyGlot relies on a BNF grammar which must be conflict-free for LR(1) parser, one-token lookahead, since Bison is an LR(1) parser. Unfortunately, not all BNF grammars are conflict-free for LR(1) parser. For this reason, it is time-wasting to eliminate conflicts in BNF grammar. As a solution, we can adopt other parsers which use more than 1 token of lookahead,e.g., LL(k) parser. Thus, we can construct an unambiguous parser from BNF grammar.

**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.

## 6.3 $\mu$Fuzz

**Collaborative Fuzzing.** The current implementation of $\mu$FUZZ does not support collaborative fuzzing, which combines all kinds of different fuzzers to get a higher overall fuzzing performance. This is because $\mu$FUZZ now assumes that each service maintains one united global state and each worker inside the service maintains one partition of the state. However, different fuzzers can have different fuzzing states for the same functionality and $\mu$FUZZ cannot distinguish workers of different states inside a service. For example, grammar fuzzers might maintain a corpus in the form of abstract syntax trees (AST) instead of a binary stream. Suppose we combine a grammar-based mutation along with a bit-level mutation in $\mu$FUZZ. In that case, the input dispatcher of the test case generation service might wrongly dispatch an AST test case to a worker of bit-level mutation. We plan to try two ways to support collaborative fuzzing with $\mu$FUZZ. First, we can tag both the data of service communication and the workers. Moreover, we restrict the input dispatcher to only dispatch inputs to workers with matching tags. In this way, a service can maintain different types of workers and different states. Another way is that we can use $\mu$FUZZ as a base fuzzer for existing collaborative fuzzing approaches. For example, we want to combine 20 instances of a bitflip fuzzer and 10 instances of a grammar fuzzer. Then we can implement both fuzzers in $\mu$FUZZ, and then use them as the base fuzzers in the existing collaborative fuzzing framework such as ClusterFuzz [43].

**Distributed Fuzzing.** Currently, $\mu$FUZZ is implemented as a multithreaded program, which allows all threads to share the same memory space so that they can share data efficiently. $\mu$FUZZ can easily be extended to support distributed fuzzing in two ways. One way is to run one $\mu$FUZZ on each machine and perform state synchronization by connecting services with remote procedure calls (RPC), which is the state-of-the-art approach. For example, we can run $\mu$FUZZ on different machines and connect their corpus management to synchronize the corpus and connect their feedback collection to synchronize the code coverage bitmap. More importantly, although state synchronization over the network can cause slow I/O, $\mu$FUZZ will not be affected by this kind of I/O thanks to its concurrency design. If one service need to wait for the network communication to complete, the other services can still run and make individual progress. Another way is to utilize the microservice architecture to run different services on different machines and communicate over the network with adding input caching to each service. In this way, each service can keep fetching the result from other services into the input cache and running the workers to consume the inputs concurrently. As long as we warm up each service by filling enough results in the input cache before fuzzing, each service can run without waiting for network I/O.

# Chapter 7 Conclusion

In this thesis, we present three of our works to enhance the effectiveness and efficiency of fuzzing. In the first two works, we focus on improving the quality of test cases that require syntactic and semantic correctness. Specifically, in the first work, we investigate current challenges in testing Database Management Systems and propose SQUIRREL to generate valida SQL queries. We define an IR that represents the syntactic components and carries the necessary semantic information. Then we build a dependency graph according to the semantic information and perform a semantic instantiation to validate the test cases. We evaluated SQUIRREL on four popular DBMSs: `SQLite`, `MySQL`, `MariaDB` and `PostgreSQL`, and found 51 bugs in `SQLite`, 7 in `MySQL` and 5 in `MariaDB`. SQUIRREL achieves at least 3.4 times improvement in semantic correctness than that of current mutation-based and generation-based fuzzers, and triggers up to 12 times of improvement in code coverage than current mutation-based fuzzers. The results show that SQUIRREL is effective in testing database management systems.

Followed by the first work, we propose POLYGLOT which can achieve both generic applicability and high quality for generic language processors fuzzing. We neutralize the differences in languages by casting their specification to the specification of IR. Then we perform constrained mutation and semantic validation on the IR statements to generate valid test cases. In our evaluation, We applied POLYGLOT on 21 processors of 9 languages and successfully identified 173 new bugs. Besides, our evaluation shows POLYGLOT is more effective in testing language processors than existing fuzzers with up to 30 times improvement in code coverage.

Our third work focuses on improving efficiency in parallel fuzzing. We identified two weaknesses in state-of-the-art parallel fuzzing: the CPU idle issue of blocking I/O and untimely state synchronization. To address these two weaknesses, we propose $\mu$FUZZ. $\mu$FUZZ adopts the microservice architecture, which enables each service to run concurrently and reduces the burden of synchronization. The loosely coupled structure of microservices allows $\mu$Fuzz to replace the blocked service and schedule parallel workers effectively, improving CPU utilization. $\mu$Fuzz has two levels of state partition that eliminate the need for state synchronization among different services and workers and allow each service worker to make globally optimal decisions. Our evaluation shows $\mu$Fuzz is more effective in parallel fuzzing than existing fuzzers with 57% improvement in code coverage and 67% improvement in bug detection on average in 24 hours. Besides, $\mu$Fuzz finds three new bugs in well-tested real-world programs.

# Bibliography

[1] Byron Acohido. Small Banks and Credit Union Attack Set for Tuesday. https://www.usatoday.com/story/cybertruth/2013/05/06/ddos-denial-of-service-small-business-cybersecurity-privacy/2139349/, May 2013.

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

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[4] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring Deep State Spaces via Fuzzing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1597–1612, 2020.

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

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

[7] Hardik Bati, Leo Giakoumakis, Steve Herbert, and Aleksandras Surna. A Genetic Approach for Random Testing of Database Systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB07, pages 1243–1251, 2007.

[8] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. A Taint Based Approach for Smart Fuzzing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 818–825, 2012.

[9] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing structure while fuzzing. In *USENIX Security Symposium*, 2019.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, 2017.

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

[12] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. *IEEE Transactions on Software Engineering*, 45(5):489–506, 2017.

[13] Eliot Van Buskirk. Facebook Confirms Denial-of-Service Attack. https://www.wired.com/2009/08/facebook-apparently-attacked-in-addition-to-twitter/, August 2009.

[14] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[15] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1531–1531. IEEE Computer Society, 2022.

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

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

[18] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. Savior: Towards bug-driven hybrid testing. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1580–1596, 2020.

[19] Yaohui Chen, Dongliang Mu, Jun Xu, Zhichuang Sun, Wenbo Shen, Xinyu Xing, Long Lu, and Bing Mao. Ptrix: Efficient hardware-assisted fuzzing for cots binary. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 633–645, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1967–1983, 2019.

[21] Catalin Cimpanu. Google Chrome Impacted by New Magellan 2.0 Vulnerabilities. https://www.zdnet.com/article/google-chrome-impacted-by-new-magellan-2-0-vulnerabilities/, December 2019.

[22] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 725–730, New York, NY, USA, 2014. Association for Computing Machinery.

[23] Kyle Dewey, Jared Roesch, and Ben Hardekopf. Fuzzing the rust typechecker using clp (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 482–493, 2015.

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

[25] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1497–1511, 2020.

[26] Ren Ding, Yonghae Kim, Fan Sang, Wen Xu, Gururaj Saileshwar, and Taesoo Kim. Hardware support to improve fuzzing performance and precision. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2214–2228, 2021.

[27] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, and Yan Shoshitaishvili. Favocado: Fuzzing the Binding Code of JavaScript Engines Using Semantically Correct Test Cases. In *Network and Distributed System Security Symposium*, 2021.

[28] Will Estes. Flex, the fast lexical analyzer generator. https://github.com/westes/flex/, 2020.

[29] esyr, Dmitry V. Levin, and Elvira Khabirova. strace is a diagnostic, debugging and instructional userspace utility for linux. https://github.com/strace/strace, 2022.

[30] Usama M. Fayyad. Data Science Revealed: A Data-Driven Glimpse into the Burgeoning New Field. https://fayyad.com/data-science-revealed-a-data-driven-glimpse-into-the-burgeoning-new-field/, 2011.

[31] Andrea Fioraldi, Daniele Cono D'Elia, and Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2829–2846. USENIX Association, August 2021.

[32] Lorenzo Franceschi-Bicchierai. Hacker Tries To Sell 427 Milllion Stolen MySpace Passwords For $2,800. https://www.vice.com/en_us/article/pgkk8v/427-million-myspace-passwords-emails-data-breach, May 2016.

[33] Free Software Foundation. Gnu bison. https://www.gnu.org/software/bison/, 2020.

[34] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594, 2020.

[35] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[36] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, 2018.

[37] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *2009 IEEE 31st International Conference on Software Engineering*, pages 474–484, 2009.

[38] 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)*, pages 50–59, 2017.

[39] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, jun 2008.

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

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

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

[43] Google. ClusterFuzz - Scalable fuzzing infrastructure. https://github.com/google/clusterfuzz, 2019.

[44] Google. 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.

[45] Google. ClusterFuzzLite. https://google.github.io/clusterfuzzlite/, 2022.

[46] Rahul Gopinath, Björn Mathis, Matthias Höschele, Alexander Kampmann, and Andreas Zeller. Sample-free learning of input grammars for comprehensive software fuzzing. *ArXiv*, abs/1810.08289, 2018.

[47] Rahul Gopinath, Björn Mathis, and Andreas Zeller. Inferring input grammars from dynamic control flow. *arXiv preprint arXiv:1912.05937*, 2019.

[48] Hunter Gregal. 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.

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

[50] Samuel Groß. FuzzIL: Coverage Guided Fuzzing for JavaScript Engines, 2019. https://saelo.github.io/papers/thesis.pdf.

[51] Emre Güler, Philipp Görz, Elia Geretto, Andrea Jemmett, Sebastian Österlund, Herbert Bos, Cristiano Giuffrida, and Thorsten Holz. Cupid: Automatic fuzzer selection for collaborative fuzzing. In *Annual Computer Security Applications Conference*, pages 360–372, 2020.

[52] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[53] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A ground-truth fuzzing benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3), December 2020.

[54] Marc Heuse, Heiko Eißfeldt, Andrea Fioraldi, Dominik Maier, and Jana Aydinbas. Aflplusplus. https://github.com/AFLplusplus/AFLplusplus, 2022.

[55] Richard D Hipp. SQLite. https://www.sqlite.org/index.html, 2020.

[56] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, page 38, USA, 2012. USENIX Association.

[57] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 36–50. IEEE, 2022.

[58] Troy Hunt. The 773 Million Record "Collection #1" Data Breach. https://www.troyhunt.com/the-773-million-record-collection-1-data-reach/, January 2020.

[59] Bo Jiang, Ye 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*, ASE 2018, pages 259–269, New York, NY, USA, 2018. Association for Computing Machinery.

[60] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems. *Proc. VLDB Endow.*, 13(1):57–70, sep 2019.

[61] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo 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.

[62] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[63] Thijs Klooster, Fatih Turkmen, Gerben Broenink, Ruben ten Hove, and Marcel Böhme. Effectiveness and Scalability of Fuzzing Techniques in CI/CD Pipelines. *arXiv preprint arXiv:2205.14964*, 2022.

[64] Vidita Koushik. ALERT: SQLite database Remote Code Execution Vulnerability. https://www.secpod.com/blog/sqlite-database-remote-code-execution/, August 2019.

[65] Doug Laney. 3-D Data Management: Controlling Data Volume, Velocity and Variety. Technical report, META Group, February 2001.

[66] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. Constraint-guided directed greybox fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3559–3576, 2021.

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

[68] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. Steelix: Program-state Based Binary Fuzzing. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[69] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang 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*, pages 809–814, 2018.

[70] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. Pata: Fuzzing with path aware taint analysis. In *IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, Los Alamitos, CA, USA*, pages 154–170, 2022.

[71] Darren Liew. QuickJS. https://github.com/ldarren/QuickJS, 2020.

[72] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1044–1051, 2019.

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

[74] Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. A Framework for Testing DBMS Features. *The VLDB Journal*, 19(2):203–230, April 2010.

[75] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized mutation scheduling for fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, Santa Clara, CA, August 2019. USENIX Association.

[76] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. 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.

[77] MariaDB Foundation. MariaDB. https://www.mariadb.org/, 2009.

[78] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. Fuzzbench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1393–1403, 2021.

[79] Microsoft. Processing a SQL Statement. https://docs.microsoft.com/en-us/sql/odbc/reference/processing-a-sql-statement?view=sql-server-ver15, 2017.

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

[81] Barton P. Miller, Louis Fredriksen, and Bryan So. An Empirical Study Of The Reliability Of UNIX Utilities. *Commun. ACM*, 33(12):32–44, December 1990.

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

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

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

[85] MYSQL. MySQL. https://www.mysql.com/, 1995.

[86] Mysql. MySQL Customers. https://www.mysql.com/customers/, 2020.

[87] Cheolwoo Myung, Gwangmu Lee, and Byoungyoung Lee. {MundoFuzz}: Hypervisor fuzzing with statistical coverage testing and grammar inference. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1257–1274, 2022.

[88] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 787–802, 2019.

[89] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1683–1700, 2021.

[90] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew 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*, CCS '21, page 351–365, New York, NY, USA, 2021. Association for Computing Machinery.

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

[92] Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. Collabfuzz: A framework for collaborative fuzzing. In *Proceedings of the 14th European Workshop on Systems Security*, pages 1–7, 2021.

[93] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo 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.

[94] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. T-fuzz: Fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 697–710, 2018.

[95] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, 2016.

[96] Van-Thuan Pham, Manh-Dung Nguyen, Quang-Trung Ta, Toby Murray, and Benjamin IP Rubinstein. Towards systematic and dynamic task allocation for collaborative parallel fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1337–1341. IEEE, 2021.

[97] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with SymCC: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198. USENIX Association, August 2020.

[98] Sebastian Poeplau and Aurélien Francillon. Symqemu: Compilation-based symbolic execution for binaries. In *Network and Distributed System Security Symposium*, 2021.

[99] PostgreSQL. PostgreSQL. https://www.postgresql.org/.

[100] PostgreSQL. PostgreSQL Clients. https://wiki.postgresql.org/wiki/PostgreSQL_Clients, 2020.

[101] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *Network and Distributed System Security Symposium*, 2017.

[102] Chris Richardson. Microservice architecture. https://microservices.io/, 2022.

[103] Chris Rohlf and Yan Ivnitskiy. Attacking clientside JIT compilers. *Black Hat USA*, 2011.

[104] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2597–2614, 2021.

[105] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL:Hardware-Assisted feedback fuzzing for OS kernels. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 167–182, 2017.

[106] Andreas Seltenreich. SQLSmith. https://github.com/anse1/sqlsmith, 2016.

[107] Andreas Seltenreich. SQLsmith Description. https://github.com/anse1/sqlsmith#description, 2020.

[108] Kostya Serebryany. Sanitize, Fuzz, And Harden Your C++ Code, 2016.

[109] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, 2019.

[110] Dongdong She, Abhishek Shah, and Suman Jana. Effective seed scheduling for fuzzing with graph centrality analysis. *arXiv preprint arXiv:2203.12064*, 2022.

[111] Congxi Song, Xu Zhou, Qidi Yin, Xinglu He, Hangwei Zhang, and Kai Lu. P-fuzz: a parallel grey-box fuzzing framework. *Applied Sciences*, 9(23):5100, 2019.

[112] Sqlite. Well-Known Users of SQLite. https://www.sqlite.org/famous.html, 2020.

[113] Jonathan Stempel and Finkle. Jim. Yahoo Says All Three Billion Accounts Hacked in 2013 Data Theft. https://www.reuters.com/article/us-yahoo-cyber/yahoo-says-all-three-billion-accounts-hacked-in-2013-data-theft-idUSKCN1C82O1, October 2017.

[114] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Krügel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium*, 2016.

[115] Michael Stonebraker, Sam Madden, and Pradeep Dubey. Intel "Big Data" Science and Technology Center Vision and Execution Plan. *ACM SIGMOD Record*, 42(1):44–49, 2013.

[116] The Tcpdump team. Tcpdump. https://www.tcpdump.org/, 2010.

[117] Ken Thompson. Reflections on Trusting Trust. *Commun. ACM*, 27(8), August 1984.

[118] TIOBE Software BV. Tiobe index for august 2020. https://www.tiobe.com/tiobe-index/, 2020.

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

[120] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*, pages 581–601. Springer, 2016.

[121] W3schools. SQL Keywords Reference. https://www.w3schools.com/sql/sql_ref_keywords.asp, 2020.

[122] W3schools. SQL Operators. https://www.w3schools.com/sql/sql_operators.asp, 2020.

[123] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 1–15, 2019.

[124] Jinghan Wang, Chengyu Song, and Heng Yin. Reinforcement learning-based hierarchical seed scheduling for greybox fuzzing. In *Network and Distributed Systems Security (NDSS) Symposium 2021*, 2021.

[125] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735, 2019.

[126] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu 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*, PLDI 2022, page 1010–1024, New York, NY, USA, 2022. Association for Computing Machinery.

[127] 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*, pages 497–512, 2010.

[128] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *Network and Distributed System Security Symposium*, 2020.

[129] Yifan Wang, Yuchen Zhang, Chenbin Pang, Peng Li, Nikolaos Triandopoulos, and Jun Xu. Facilitating parallel fuzzing with mutually-exclusive task distribution. In *International Conference on Security and Privacy in Communication Systems*, pages 185–206. Springer, 2021.

[130] Wikipedia contributors. List of programming languages. https://en.wikipedia.org/wiki/List_of_programming_languages, 2017.

[131] Wikipedia contributors. Data type. https://en.wikipedia.org/wiki/Data_type, 2020.

[132] Wikipedia contributors. Translator (computing) — Wikipedia, the free encyclopedia. "https://en.wikipedia.org/w/index.php?title=Translator_computing&oldid=969277386", 2020.

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

[134] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One fuzzing strategy to rule them all. In *Proceedings of the International Conference on Software Engineering*, 2022.

[135] Zhengkai Wu, Evan Johnson, Wei Yang, Osbert Bastani, Dawn Song, Jian Peng, and Tao Xie. 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*, pages 488–498, 2019.

[136] Peng Xu, Yanhao Wang, Hong Hu, and Purui Su. Cooper: Testing the binding code of scripting languages with cooperative mutation. *Proceedings 2022 Network and Distributed System Security Symposium*, 2022.

[137] Roy Xu and Sai Vegasena. Vasilisk. https://blog.osiris.cyber.nyu.edu/2019/12/22/vasilisk/, December 2019.

[138] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2313–2328, New York, NY, USA, 2017. Association for Computing Machinery.

[139] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328, 2017.

[140] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 2313–2328, New York, NY, USA, 2017. Association for Computing Machinery.

[141] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'11, New York, NY, USA, 2011.

[142] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. 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*, pages 435–450, 2021.

[143] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, August 2018.

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

[145] ZeroDayInitiative. 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.

[146] G Zhang, P Wang, T Yue, X Kong, S Huang, X Zhou, and K Lu. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. In *Network and Distributed Systems Security (NDSS) Symposium 2022*, 2022.

[147] Kunpeng Zhang, Xi Xiao, Xiaogang Zhu, Ruoxi Sun, Minhui Xue, and Sheng Wen. Path transitions tell more: Optimizing fuzzing schedules via runtime program states. *arXiv preprint arXiv:2201.04441*, 2022.

[148] Lei Zhao, Yue Duan, Heng Yin, and J. Xuan. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[149] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao 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, November 2020.

[150] Xu Zhou, Pengfei Wang, Chenyifan Liu, Tai Yue, Yingying Liu, Congxi Song, Kai Lu, and Qidi Yin. Unifuzz: Optimizing distributed fuzzing via dynamic centralized task scheduling. *arXiv preprint arXiv:2009.06124*, 2020.

[151] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: A survey for roadmap. *ACM Comput. Surv.*, 2022.

# Vita

## Rui Zhong

Rui Zhong obtained his Ph.D. from the College of Information Sciences and Technology at The Pennsylvania State University in 2023 under the supervision of Dinghao Wu. He specializes in software security, specifically in identifying and exploiting vulnerabilities. Aside from academia, Rui Zhong is a bug hunter and competitive hacker in Capture the Flag events, and actively contributes to enhancing software security.