

Article

sqlFuzz: Directed Fuzzing for SQL Injection Vulnerability

Ye Yuan ^{1,2}, Yuliang Lu ^{1,2}, Kailong Zhu ^{1,2}, Hui Huang ^{1,2,*} , Yuanchao Chen ^{1,2}  and Yifan Zhang ^{1,2}

¹ College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China; 13345514795@163.com (Y.Y.); luyuliang@nudt.edu.cn (Y.L.); zhukailong@nudt.edu.cn (K.Z.); chenyuanchao@nudt.edu.cn (Y.C.); 18805612506@163.com (Y.Z.)

² Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei 230037, China

* Correspondence: huanghui17@nudt.edu.cn

Abstract: Fuzz testing technology is an important approach to detecting SQL injection vulnerabilities. Among them, coverage-guided gray-box fuzz testing technology is the current research focus, and has been proved to be an effective method. However, for SQL injection vulnerability, coverage-guided gray-box fuzz testing as a detection method has the problems of low efficiency and high false positives. In order to solve the above problems, we propose a potentially vulnerable code-guided gray-box fuzz testing technology. Firstly, taint analysis technology is used to locate all the taint propagation paths containing potential vulnerabilities as potentially vulnerable codes. Then, the source code of the application program is instrumented according to the location of the potentially vulnerable code. Finally, the feedback of seeds during the run is used to guide seed selection and seed mutation, and a large number of test cases are generated. Based on the above techniques, we implement the sqlFuzz prototype system, and use this system to analyze eight modern PHP applications. The experimental results show that sqlFuzz can not only detect more SQL injection vulnerabilities than the existing coverage-guided gray box fuzz testing technology, but also significantly improve the efficiency, in terms of time efficiency increased by 80 percent.

Keywords: SQL injection vulnerabilities; coverage guided; potentially vulnerable code guided; gray-box fuzz testing technology



Citation: Yuan, Y.; Lu, Y.; Zhu, K.;

Huang, H.; Chen, Y.; Zhang, Y.

sqlFuzz: Directed Fuzzing for SQL

Injection Vulnerability. *Electronics*

2024, 13, 2946. [https://doi.org/](https://doi.org/10.3390/electronics13152946)

10.3390/electronics13152946

Academic Editor: Valentina E. Balas

Received: 27 June 2024

Revised: 18 July 2024

Accepted: 22 July 2024

Published: 26 July 2024



Copyright: © 2024 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article

distributed under the terms and

conditions of the Creative Commons

Attribution (CC BY) license ([https://](https://creativecommons.org/licenses/by/4.0/)

[creativecommons.org/licenses/by/](https://creativecommons.org/licenses/by/4.0/)

4.0/).

1. Introduction

SQL injection vulnerabilities are widely prevalent in web applications, allowing attackers unrestricted access to the databases used by web applications without authorization, posing a serious threat to web application security. Detecting SQL injection vulnerabilities is an important research topic in the field of cybersecurity. At present, the mainstream detection methods include static analysis and dynamic analysis. Among them, static analysis refers to the analysis of web application code without executing it, mainly using abstract syntax tree, control flow chart, call flow chart, and other code syntax and semantic features, according to the characteristics of the vulnerability to find the vulnerability. Although static analysis is not subject to the specific running environment of the program and can perform full coverage analysis of code, it does not handle the dynamic nature of web applications well and introduces high false positive rates [1]. Dynamic analysis needs to actually execute the program, set up the program running environment, and debug the program on the premise of successful operation. Therefore, dynamic analysis can analyze the attributes of the program and accurately locate the vulnerable code, and the false positive rate is relatively low. Among dynamic analysis techniques, fuzz testing is a typical and mature dynamic analysis technology.

Fuzz testing is an effective vulnerability detecting technique. Currently, there are three main categories of fuzz testing techniques for the web: white box, black box, and gray box. White-box fuzz testing techniques require access to the source code and involve complex

static analysis of the source code. For example, precise symbolic execution and constraint solving require a large amount of computational resources and time [2,3]. Furthermore, the computational complexity grows exponentially as the scale of the application increases. Additionally, it faces problems such as path explosion and space explosion. Black-box fuzz testing techniques do not require the source code. In the absence of knowledge about the application's structure, vulnerabilities are triggered by sending legitimate application inputs. The greatest advantage of black-box fuzz testing techniques is their low cost, as vulnerabilities can be detected by sending a large number of test cases. However, due to the lack of understanding of the internal structure of the application, and the absence of any feedback information, black-box fuzz testing techniques cannot determine which inputs are more valuable (the definition of valuable inputs includes exploring new unobserved code paths or executing the application's business logic), leading to the sending of a large number of invalid test cases [4]. Gray-box fuzz testing is currently a popular solution. It requires access to the source code but, instead of statically analyzing the source code, it instruments the source code to obtain feedback at runtime. The fuzzer can use this feedback to identify valuable inputs and generate new test cases from them, thereby maximizing code coverage and increasing the chances of triggering vulnerabilities.

Some recent work has been carried out based on gray-box fuzz testing techniques [5–8], all of which use coverage feedback information to guide input generation. While these approaches have achieved some successes, certain methods are limited to single languages and do not detect SQL injection vulnerabilities [7,9]. Some methods rely solely on coverage as the criterion for filtering test cases, which can lead to inefficiencies in certain scenarios [8]. Additionally, some seed mutation methods in these approaches draw from techniques used for fuzz testing binary vulnerabilities, which may not be well-suited to web vulnerabilities and may thus exhibit lower efficiency [7].

webFuzz [7] is a modification of AFL designed to detect web vulnerabilities, leveraging AFL's effectiveness in detecting binary vulnerabilities. However, the authors of this work did not optimize seed selection strategies and seed mutation mechanisms specifically for web vulnerabilities. As a result, while webFuzz [7] can successfully detect XSS vulnerabilities, its efficiency remains relatively low. Witcher [8] recognized that web applications consist of many components that can affect the effectiveness of gray-box fuzz testing. Building upon AFL, Witcher [8] incorporated additional components tailored to web characteristics to increase the probability of triggering vulnerabilities. However, similar to webFuzz [7], Witcher [8] did not improve upon seed selection strategies, thus still facing efficiency challenges.

Although using coverage feedback to guide the generation of test cases has proven to be an effective strategy, we believe it can be improved in terms of efficiency. SQL injection vulnerability is a kind of taint vulnerability that is triggered when user-controlled data reaches a sink point without being processed. To trigger an SQL injection vulnerability, you must ensure that the data reaches the sink point. We consider those paths that cannot reach the sink point to be invalid paths and those that can ensure that the test case reaches the sink point smoothly to be potentially vulnerable code. The previous work was guided by the exploration of new paths and there was no sense of whether the path was vulnerable or not, and a lot of resources were devoted to the exploration of paths. If the potentially vulnerable code is perceived in the process of fuzz testing, the test target is filtered according to the potentially vulnerable code. By focusing resources on potentially vulnerable code, we believe we can improve the efficiency of fuzz testing. In addition, in the previous work, the variation in seeds generally adopts random variation. We believe that the input variation can also benefit from the perception of the potentially vulnerable code and adopt different variation modes according to the different locations of seeds.

To solve the above problems, we propose a fuzz testing technique of SQL injection vulnerability based on potentially vulnerable code guidance. By using the relevant information of the potentially vulnerable code generated by static analysis, the source code of the target web application is instrumented to increase the amount of feedback information in

the test process, improve the seed selection strategy in the existing fuzz testing technology, and improve the priority of the seeds that reach the potentially vulnerable code area. By improving the seed mutation mechanism of the existing fuzz testing technology, different granularities of mutation methods are proposed, and adaptive seed mutation strategies are adopted according to the seed arrival location. The efficiency of gray-box fuzz testing technology is improved by the above measures. The main contributions of this paper are as follows:

1. We use taint analysis to obtain codes containing potential vulnerabilities, and use potentially vulnerable codes to guide seed selection to achieve targeted effects, thereby improving the efficiency of fuzz testing.
2. In order to solve the problem of low test-case quality caused by the strong randomness of the seed mutation mode in variant fuzz testing technology for SQL injection vulnerability, we propose an adaptive seed mutation mechanism, which adopts different granularity and mutation strategies according to different seed arrival locations and performances to improve the generation quality of test cases.
3. We design, implement, and evaluate sqlFuzz, an SQL injection vulnerability fuzz testing technology guided by vulnerable paths, which efficiently detects SQL injection vulnerabilities. We evaluated our work in eight commercial web applications, and our tool detected 41 out of 50 known vulnerabilities, and our proposed technique achieved an 80% improvement in time efficiency compared to existing gray-box fuzz testing techniques.

2. Background

2.1. SQL Injection Vulnerability

Web applications provide users with various interactive features, such as submitting forms and uploading text, among others. When a user's input is not checked and processed, it can be used by a backend server and lead to a taint-style vulnerability. SQL injection vulnerability is a typical taint-style vulnerability that allows an attacker to bypass the authentication and authorization mechanism of an application by constructing malicious SQL queries to obtain sensitive data or perform illegal operations on the database. In a normal application, data is usually entered by the user and passed to the background database in some way for corresponding queries, updates, and deletions. If the application does not filter or escape this user input properly, an attacker can insert malicious SQL statements into the input, causing the application to perform unexpected queries or actions, such as deleting tables, inserting data, or revealing sensitive information [10]. The characteristic of SQL injection vulnerability makes it necessary to ensure that the malicious input with attack payload can reach the sink point completely according to the taint propagation path in order to trigger the vulnerability. If the seed selection criterion can be taken as whether it is close to the sink point, we believe that the efficiency of the current fuzz testing can be improved.

2.2. Coverage-Guided Gray-Box Fuzz

The goal of fuzz testing is to cover as much of the program's execution state as possible, uncovering potential vulnerabilities within the program. However, due to the uncertainty of program behavior, the program state cannot be intuitively measured. Researchers therefore choose to use code coverage as a substitute metric for program state, where an increase in code coverage approximates an increase in new program states.

Coverage-guided fuzz testing first uses code coverage as feedback information to guide the generation of effective inputs. Effective inputs are those that execute code not yet traversed by the program. By mutating based on effective inputs, new inputs are more likely to traverse as much of the program's code as possible, and are more likely to uncover potential vulnerabilities within that code. AFL's continued success in industry has validated the correctness of this concept [11]. Utilizing code coverage to guide the

generation of inputs that target the coverage goal code and program target paths in fuzz testing is called directed fuzz testing. An example is AFLGo [12].

CGF is a combined concept of gray-box fuzz testing and coverage-guided fuzz testing. It acquires coverage information during program execution through lightweight program analysis tools and guides input generation based on this coverage information. Tracking the execution paths of inputs in the testing program provides accurate and comprehensive program feedback. However, due to the high performance cost associated with path coverage, in practical applications, CGF generally chooses to use coarser basic blocks as the granularity of code coverage. Since AFL introduced edge coverage into fuzz testing, researchers have found that edge coverage can provide more accurate execution information than block coverage. Edge coverage has thus become the mainstream research direction for coverage-guided fuzz testing. However, whether it is edge coverage or block coverage, CGF adjusts inputs based on their coverage information to achieve more code execution in the application and to uncover potential vulnerabilities.

3. Motivating Example

In previous fuzz testing work targeting web applications, coverage has often been used as the basis for selecting seeds. For example, in webFuzz, if a seed fails to trigger new paths after mutation, it is discarded. The priority of seeds is also based on the number of new paths triggered after mutation. The more new paths triggered after mutation, the more advantageous the seed is considered, and the higher its priority. In the context of SQL injection vulnerabilities, if user-controlled input does not reach an execution point, the vulnerability cannot be triggered. As shown in Figure 1, the user input data can only trigger the vulnerability by following path S1. The main issues with coverage-guided fuzz testing are as follows:

- Coverage-guided seed selection is often blind, leading to resource wastage and affecting the efficiency of fuzz testing. For instance, in an ideal scenario, Seed 1 can reach the endpoint of path S2 via node B1 but cannot reach the execution point at the endpoint of path S1. Seed 2, on the other hand, can reach the endpoints of paths S3, S4, and S5 via node B4. According to the coverage-guided strategy, Seed 2 would be considered more valuable due to triggering more new paths and would be assigned a higher priority than Seed 1, receiving more energy. However, in reality, Seed 2 cannot trigger paths that include the execution point; thus, allocating more resources to it would be wasteful and significantly impact the efficiency of fuzz testing. Furthermore, following the coverage-guided strategy, if Seed 1 cannot reach the execution point in subsequent mutation rounds after reaching node B1, it would be considered unable to trigger new paths and discarded, even though it is actually the seed closest to triggering the vulnerability.
- Existing seed mutation mechanisms exhibit high randomness. In previous work, regardless of where the seed reaches within the web application, the seed mutation strategy randomly selects mutation methods. This lack of targeted mutation mechanisms similarly impacts the efficiency of fuzz testing. For example, as shown in Figure 1, when an ideal seed stops at node B2 in a round, due to its proximity to the execution point, the reason the seed cannot progress is likely due to the payload being blocked by a filtering function. Therefore, the mutation strategy in the next round should primarily focus on bypassing the filtering function. Seeds that have not reached the critical code area should mainly focus on path expansion mutations and have fewer bypassing filtering function mutations. Additionally, within the same web application, since filtering functions may be uniform, the mutation strategy of well-performing seeds can guide the mutation of poorly performing seeds. Random mutation strategies are inefficient.

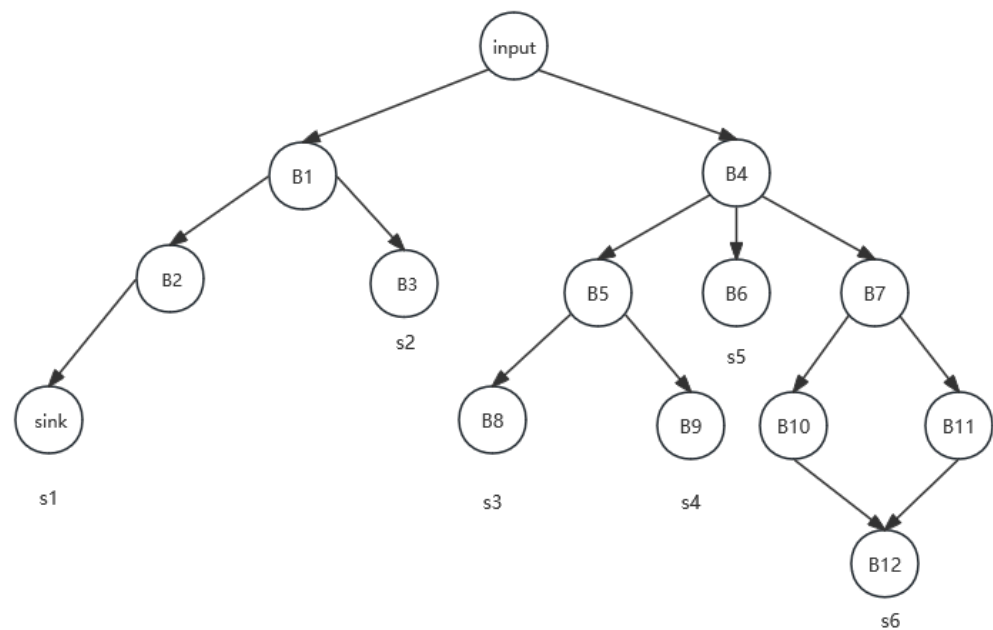


Figure 1. A code control flow graph containing a sink.

4. sqlFuzz's Design

The overall framework of the automated verification technology for SQL injection vulnerabilities based on coverage-guided fuzz testing is illustrated in Figure 2. It is divided into five main components: preprocessing, initial seed generation, seed selection, seed mutation, and vulnerability determination.

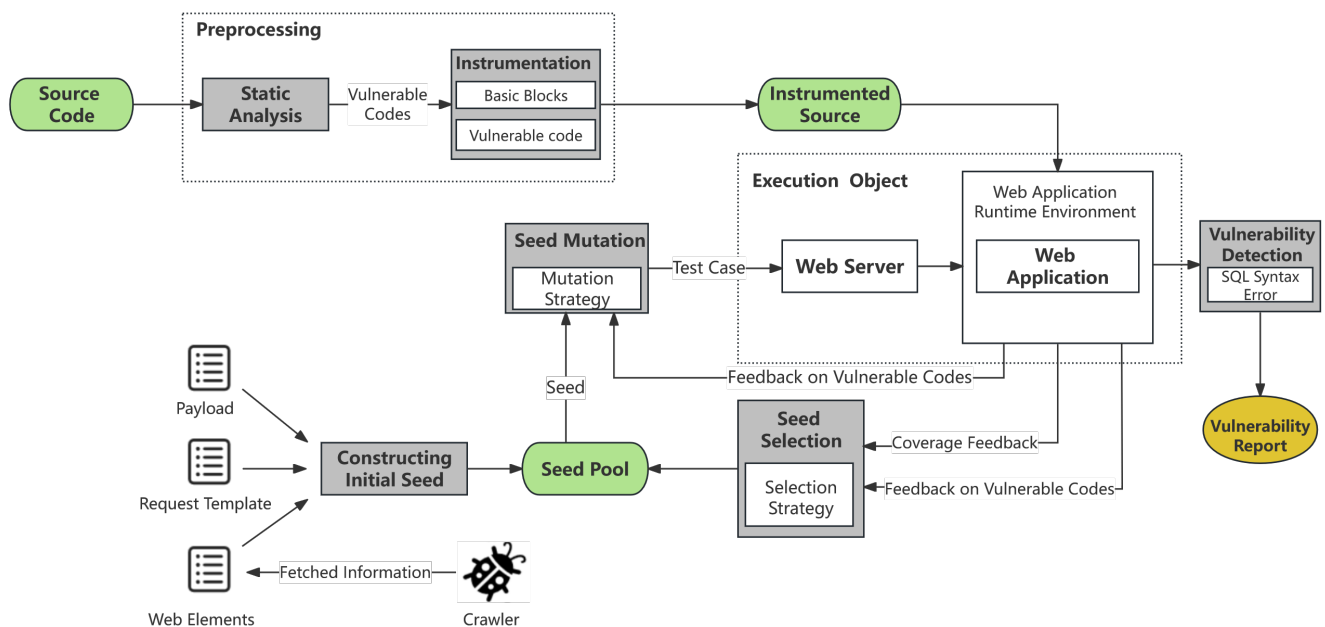


Figure 2. The overall framework of sqlFuzz.

Preprocessing includes static analysis and instrumentation. In static analysis, backward taint analysis is used to obtain information about potentially vulnerable code. In the instrumentation, the source code of the target web application is instrumented in two stages. In the first stage, all basic blocks are instrumented for feedback coverage; in the second stage, basic blocks in the potentially vulnerable code area are instrumented according

to the potentially vulnerable code information obtained from static analysis. The initial seed generation part uses the valid content of the target web page element crawled by the crawler and the payload in the payload file to construct the initial seed that conforms to the syntax of the web application according to the http request template, and then puts the generated initial seed into the seed pool. The seed selection part sets the priority of the seed according to the feedback information during the operation and assigns different energies according to the priority. The seed mutation part mutates the selected seeds, improves the diversity of seeds, and generates a large number of test cases. The vulnerability determination part determines whether the vulnerability is triggered according to the response information of the web server, and finally obtains the vulnerability report including the location information of the vulnerability and the test cases that trigger the vulnerabilities. Detailed introductions to each component will be provided below.

4.1. Preprocessing

Preprocessing, which includes static analysis and instrumentation, is an important link to improve the fuzz testing feedback information in this technical framework. First of all, the static analysis of the source code is to obtain the relevant information of the potentially vulnerable code. Secondly, instrumentation is to provide necessary guidance information for the next round of seed selection and mutation by using the position information feedback of seeds in the running stage. The instrumentation process consists of two phases. The first phase involves instrumenting all basic blocks to provide coverage feedback, thereby retaining the exploration capability of the mutation-based fuzzer within potentially vulnerable code regions. The second phase entails instrumenting the basic blocks within the corresponding regions identified through the static analysis of potentially vulnerable code-related information. This is performed to provide detailed information when seeds reach potentially vulnerable code regions, serving as the basis for subsequent seed selection and mutation. The following provides a detailed description of the key steps:

4.1.1. Static Analysis

The static analysis technique adopted in this paper is backward taint analysis. Starting from the danger function, trace each parameter that flows to the danger function. If the parameter comes from the user input, the context-sensitive data flow analysis is performed on the parameter to check whether it has been sanitized by the sanitization function. If not, the propagation path is considered to be a potentially vulnerable code. A brief description of some key steps is provided here.

- Identification of all sink points: Given that the sink points for SQL injection vulnerabilities are fixed, such as `mysqli_query()`, a regular expression-matching approach is employed for this purpose.
- Construction of the control flow graph: The source code of the web application is transformed into an abstract syntax tree (AST), which is divided into the main AST and custom function ASTs. The control flow graph is built based on the main AST. When encountering branching nodes, a new basic block is created and data flow analysis is conducted within the basic block to form a block summary. When encountering nodes for the invocation of custom functions, the control flow graph for the function is constructed based on the custom function's AST, similarly using data flow analysis to generate a function summary.
- Backward data flow analysis: Perform reverse data flow analysis on the parameters of the sink point. The specific implementation is as follows: Starting from the basic block where the sink point is located, trace the basic block upward according to CFG. According to the basic block summary, check whether the parameter has been processed by the sanitization function and, if so, end the analysis. If not, and the parameter propagates to a location that the user can control, the basic block of the entire link is recorded.
- Compilation of all basic blocks in the tainted data propagation path.

4.1.2. Instrumentation and Feedback

Instrumentation is divided into two phases. The first phase involves instrumenting all basic blocks of the web application source code to facilitate coverage feedback. Specifically, to facilitate basic block identification, the web application source code is first transformed into an abstract syntax tree. Subsequently, the abstract syntax tree is traversed to identify all basic blocks. Identification code is inserted at the beginning of each basic block. To ensure more accurate coverage calculation, we draw on AFL's instrumentation technology, using branch coverage information to calculate coverage. The branch coverage information is obtained by performing an XOR operation between the identification codes of each basic block and the previously accessed basic block's identification codes, serving as edge labels. To prevent cases where self-XOR results in zero, the edge label undergoes a right shift operation after its computation with the current basic block's identification code. Compared to calculating coverage based on the number of basic blocks, branch coverage takes into account the sequence of basic block execution, thus providing more precise results.

In specific implementation, inspired by the AFL method of providing edge coverage information, we have adapted a similar approach for web applications. Figure 3 shows the instrumented version of a function test. At the beginning of every basic block, a unique randomly generated number (the basic block's label) is XORed with the label of the previously visited block. The result of this operation represents the edge label. The edge label is used as an index in the global map array where the counter for the edge is incremented. The last statement in the stub code performs a right bitwise shifting on the current basic block label and stores the result as the label of the previously visited block. The shifting is needed to avoid cases where a label is XORed with itself thus giving zero as a result. This can happen, for instance, with simple loops that do not contain control statements in their body.

```

1  <?php
2  function test(){
3      #BEGIN instrumentation code for basic block A
4      #$_key represents Edge Callsite→A
5      $_key=BASIC_A_LABEL^$GLOBALS["_instr"]["prev"];
6      $GLOBALS["_instr"]["map"][$_key]+=1;
7      $GLOBALS["_instr"]["prev"]=BASIC_BLOCK_A_LABEL>>1;
8      #END instrumentation code
9
10     #BEGIN instrumentation code for basic block B
11     #$_key represents Edge A→B
12     $_key=BASIC_B_LABEL^$GLOBALS["_instr"]["prev"];
13     $GLOBALS["_instr"]["map"][$_key]+=1;
14     $GLOBALS["_instr"]["prev"]=BASIC_BLOCK_A_LABEL>>1;
15     #END instrumentation code
16 }
```

Figure 3. Sample of an instrumented function test for measuring edge coverage.

The second phase involves instrumenting the basic blocks within potentially vulnerable code regions of the web application source code. Specifically, the instrumentation level remains at the basic block, with instrumentation positioned at the beginning of each basic block. Through static analysis, the entire path of tainted data propagation can be obtained, and all basic blocks on this path are subject to instrumentation. The feedback information from this instrumentation includes two points: (1) whether the seed has reached the potentially vulnerable code region and specifically which potentially vulnerable code it is; (2) the distance from the seed's arrival point to the sink point along the respective path. It should be noted here that directly calculating the distance between the location of the seed and all sink points will lead to great overhead and affect the efficiency of the fuzz testing. Considering this factor, our technique adopts a method of calculating distances within the path. Specifically, it only calculates the distance from the seed's arrival point to the

sink point in the same tainted data propagation path. The specific implementation method involves first labeling each potentially vulnerable code to prevent confusion. Subsequently, starting from the sink code, the process traces backward, sequentially inserting counters into basic blocks, with the sink point assigned a position of 0, incrementing as it moves backward. A higher number indicates a greater distance from the sink point.

In summary, the sets of feedback from the two phases of instrumentation are independent of each other. The second phase of instrumentation aims to identify seeds that have reached the vulnerable area, serving as prioritized seeds for the next round, and assigning different energy levels based on the distance to the sink point. The first phase of instrumentation aims to maintain the fuzzer's exploration of new paths. Because initial seeds may not directly reach the potentially vulnerable code area, it may take several rounds of mutation for some initial seeds to reach the potentially vulnerable code area. Therefore, it is essential to retain coverage-guided strategies as a secondary strategy for seed selection.

4.2. Constructing Initial Seed

The main goal of this phase is to generate a high-quality initial seed at the beginning of the system that conforms to the syntax of the web application and triggers vulnerabilities as much as possible.

The input content of this stage includes three parts: 1. the effective information of the target web application crawled by the crawler; 2. payload; 3. http request template. First of all, the valid information crawled by the crawler includes the url of the target web application and various elements in HTML. Because ensuring the effectiveness of the initial seed first needs to meet the http protocol specification, the url is an important part of it, and various elements in HTML can be used as parameters for GET requests or POST requests. Then, payload is an important part of the test case, and its role is to enable the test case to trigger vulnerabilities. Finally, the http request template is the basis for generating an initial seed that conforms to the format specification. The request template is a pre-constructed template file in accordance with the format of the http request. Both the crawler elements and the payload will be filled into the template file to construct the initial seed that conforms to the format specification. This stage will generate a large number of correctly formatted initial seeds to be placed in the seed pool.

4.3. Seed Selection

During the fuzz testing process, each round of testing requires selecting a seed from a large pool of candidate seeds for mutation to generate test cases. Previous work has demonstrated that a good seed selection strategy can significantly improve fuzz testing efficiency, helping to discover more vulnerabilities more quickly. With a good seed selection strategy, the fuzzer can either cover more code, making it easier to trigger vulnerabilities and reduce wasteful repeated execution of paths, thus saving computational resources, or optimize the selection of seeds that cover deeper, more potentially vulnerable code, aiding in the faster identification of hidden vulnerabilities. Based on this idea and in combination with the practicality of SQL injection vulnerabilities, this section proposes a seed selection rule that prioritizes seeds reaching potentially vulnerable code areas. Through guidance, the seeds are concentrated in code areas that are likely to trigger vulnerabilities, with seeds reaching these areas being assigned more energy. Test cases generated from the initial seeds are sent to the web application and, based on the feedback from the instrumentation code, it is determined whether the seed has reached the potentially vulnerable code area. Seeds that directly reach the vulnerable area are marked as top priority and placed into the seed pool. Additionally, seeds that have not reached the vulnerable area but have discovered new paths are also placed into the seed pool and marked as secondary priority. Seeds that neither reach the potentially vulnerable code area nor discover new paths are filtered out. The following provides a detailed description of the seed selection rule:

The seed selection rule proposed in this section categorizes the priority of seeds into two levels based on feedback information during runtime. The top priority corresponds to seeds that have reached the vulnerable area, while the secondary priority corresponds to seeds that have not reached the vulnerable area but have explored new paths. The remaining seeds are filtered out. Our technique establishes a more refined selection strategy for the top priority. Specifically, first, priority is determined based on numerical values, where a lower value indicates closer proximity to the execution point and thus a higher priority. Second, if a node is part of different propagation paths, the second phase of instrumentation can be repeated and, when providing feedback on depth information, priority is ultimately calculated based on the smallest numerical value. It is important to note that, even for seeds reaching different vulnerable paths, the direct comparison of numerical values is feasible. This is because, first, the distance calculation strategy adopted in the instrumentation phase is based on path-internal calculations and, second, distance calculation starts from the sink point, with smaller numerical values indicating closer proximity to the sink point.

Figure 4 illustrates a sample program's control flow graph, where s1 and s2 represent paths within the potentially vulnerable code area, requiring two-phase instrumentation for all basic blocks on these paths. On the other hand, as s3 and s4 do not contain sink points, some of the basic blocks in them require only the first phase instrumentation. When comparing priorities, several scenarios may arise: (1) Seeds with feedback from the second phase of instrumentation take precedence over those with feedback only from the first phase of instrumentation; for example, a seed reaching node B1 in this round takes precedence over a seed reaching node B4. (2) For seeds reaching different nodes within the same vulnerable path, priority is determined based on distance values, with a smaller distance value indicating higher priority; for example, a seed reaching node B3 takes precedence over a seed reaching node B2. (3) For seeds reaching nodes within different vulnerable paths, priority can also be determined by comparing distance values; for example, a seed reaching node B1 takes precedence over a seed reaching node B2, as explained earlier. (4) Some nodes exist in multiple vulnerable paths, allowing for instrumentation in different paths. When calculating distance values, comparison is made directly within the same path and, when comparing with different paths, the smallest distance value for that node is taken into account. For example, for a seed at the source point, if compared with a seed that stopped at B2, the distance value is taken as 3. If compared with seeds from hazardous path nodes other than s1 and s2, take the minimum value of 2.

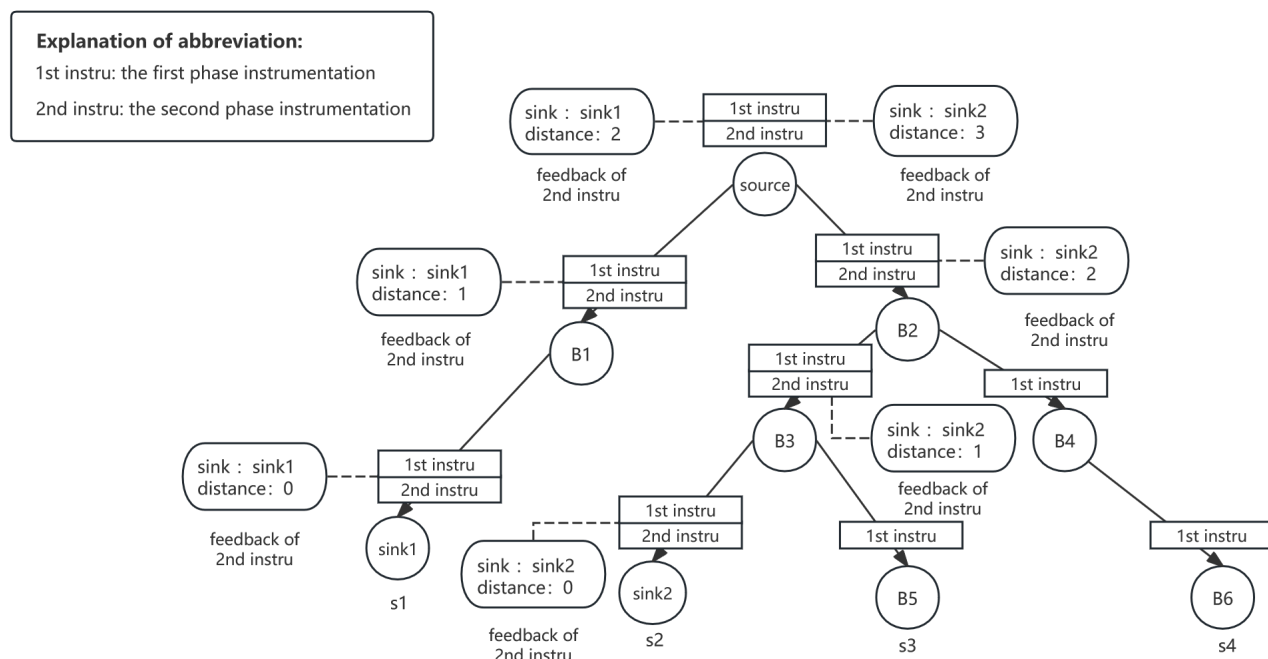


Figure 4. A sample program's control flow graph.

4.4. Seed Mutation

4.4.1. Mutation Methods

We have designed two primary mutation methods: crossover mutation and bypass mutation.

1. Crossover Mutation

We draw inspiration from the crossover operation in genetic algorithms, which continuously exchanges information between seeds to generate a large number of new seeds. Crossover mutation is divided into two levels of granularity: coarse-grained and fine-grained. Coarse-grained mutation operates at the parameter level of the HTTP request, while fine-grained mutation operates at the functional unit level of the seed. When constructing initial seeds, the seeds are already divided into functional units. Based on different granularities, the main operational methods include the following categories:

(a) Coarse grained

- Parameter swapping
- Parameter combination concatenation

Firstly, the construction of test cases is mainly based on elements extracted from the web pages by web crawlers. There can be multiple combinations of relationships between URLs and parameters. Therefore, by exchanging and combining parameters, the exploration scope of the fuzzer can be greatly expanded. Secondly, different test case parameters often have interdependencies. By combining and concatenating parameters, the probability of traversing paths can be increased. For example, if test_case1 has a parameter id = 1 and test_case2 has a parameter name = li, only by combining them as id = 1&name = li can the next node be reached.

(b) Fine grained

- Exchange of the same functional unit
- Injection attack type unit concatenation combination

The principles of crossover include the following: firstly, content exchange occurs only between the same functional units and not across functional units. This is because the division of functional units is to ensure that the seeds comply with the syntax rules of the web application. This principle ensures that the seeds still comply with the syntax rules after crossover. Secondly, a test case exchanges only one functional unit. This is to avoid the risk of excessive crossover leading to an excessively large scale of test cases and a decrease in overall quality.

2. Bypass Mutation

Web applications typically use input filters to defend against SQL injection attacks, and these filters are generally located within the application's code. Typically, the filters attempt to block inputs containing one or more of the following:

- SQL keywords, such as SELECT, AND, INSERT, etc.
- Specific individual characters, such as quote marks or hyphens
- Whitespace

If you want to exploit the vulnerability, you need to find a way to bypass the sanitization functions, in order to pass malicious input to potentially vulnerable code.

4.4.2. Mutation Strategy

After selecting suitable seeds, mutations are applied to enhance the diversity of the seeds, generating a greater number of test cases. For targeted web application fuzz testing, seed mutation serves two primary purposes: first, to enhance the ability of the seeds to bypass web application filtering mechanisms, thereby improving their traversal capabilities;

second, to increase code coverage, meaning that the test cases generated by the seeds can trigger more new paths. In order to make seed mutation more targeted, this section proposes an adaptive mutation strategy, selecting different mutation methods based on the different areas the seeds reach within the web application.

The adaptive mutation strategy primarily selects the corresponding mutation method based on the different areas the seeds reach within the web application. This section presents two main mutation methods, crossover mutation and bypass mutation. These two mutation methods serve different purposes. Crossover mutation primarily expands the exploration scope of the seeds and addresses potential issues related to multiple parameter traversal. Bypass mutation, on the other hand, primarily assists the seeds in bypassing the various filtering rules set by the web application for SQL injection vulnerabilities. The adaptive mutation strategy adjusts the mutation method based on the location information and performance of the seeds. It mainly involves the following two strategies:

- Seeds reaching vulnerable areas primarily undergo bypass mutation and fine-grained crossover mutation, while the remaining seeds primarily undergo coarse-grained crossover mutation.
- For seeds that successfully trigger vulnerabilities and seeds close to the sink points, the fuzzer automatically collects these seeds as high-performing seeds. Poorly performing seeds will undergo mutation using the same methods as these high-performing seeds. This is mainly due to the consideration that the same web application may adopt fixed filtering strategies for SQL injection vulnerabilities. Therefore, the mutation strategy of high-performing seeds holds a certain reference value.

4.5. Vulnerability Determination

For a fuzz testing tool, the effectiveness and precision of the vulnerability determination part directly determine the quantity of false negatives and false positives in the experimental results. In binary fuzz testing, this part monitors the system's runtime behavior to determine the presence of vulnerabilities by detecting abnormal system behavior, such as crashes. Inspired by this, we design a sensitive detection method for SQL injection vulnerabilities.

Firstly, the principle of the existence of SQL injection vulnerabilities is due to the ineffective processing of user input, which allows it to be executed as code. Attackers can input data containing SQL keywords into user input fields on web forms, causing the database to execute unconventional code, thereby illegally and unrestrictedly accessing data stored in the backend database. Based on this, when inputting improperly formatted data, such as adding an extra single quote, it leads to a syntax error because the input is not being treated as data, impacting the server-side code execution logic. For example, assuming a PHP application executes `mysqli_query($con, $query);` `$query = "SELECT username FROM tbl WHERE user_id = '$id'"`, if the fuzzer's test case includes the parameter `id = 1'`, and the web application does not handle the input in any way, it would result in a syntax error because the additional single quote makes the execution statement non-compliant with syntax rules. The specific implementation is as follows:

We use the BeautifulSoup library to parse HTML responses, and then employ web crawling to capture the key phrase "You have an error in your SQL syntax" to determine the presence of an SQL injection vulnerability. In some real-world web applications, error messages may be configured not to be displayed in the HTML response. To address this issue, during static analysis, all sink points are first identified, such as `mysqli_query()`, and then, during instrumentation, a detection function is added after each sink point. After instrumentation, when a syntax error is triggered, even if the original web application does not return an error message, the artificially added error function will present the error information in the HTML response.

5. Evaluation

Based on the aforementioned method, this section has implemented a prototype system called sqlFuzz for automatically validating SQL injection vulnerabilities using coverage-guided fuzz testing. In this section, the performance of this system is evaluated through experiments. This section aims to address the following two questions through experimental evaluation:

RQ1: Does sqlFuzz demonstrate effective detection of vulnerabilities in real-world programs?

RQ2: Do the seed selection and mutation mechanisms in sqlFuzz effectively enhance the efficiency of fuzz testing?

5.1. Setup

We selected eight open-source web applications as our evaluation dataset as shown in Table 1. The choice of these eight PHP applications was primarily based on two considerations: firstly, all eight applications are commercial and widely used, and available for free download from source code websites; secondly, many vulnerabilities in these applications have been assigned CVEs, allowing us to manually verify their exploitability, facilitating our analysis of false positives and false negatives. These eight web applications collectively contain 50 known vulnerabilities. When selecting the evaluation targets, we aimed to choose relatively recent releases and the latest versions of the applications to ensure that the evaluation results are more practically valuable. This is because newer applications are more likely to be downloaded and used by users, increasing the value of discovering potential zero-day vulnerabilities.

In terms of the experimental environment, the target web applications were all deployed on virtual machines. The virtual machines were configured with an Intel Core i7-10750 (2.60 GHz) processor and 8 GB of memory, and the operating system was Ubuntu 18.04. The PHP version was set to PHP 5.6 or PHP 7.2 based on the PHP version supported by each web application.

Table 1. Overview of the main characteristics of the applications.

Application	Size	Version	Release Date	Language (S)	Lines of Code	Numbers (Files)	Numbers (Vulnerabilities)
Best POS Management System	40 MB	1.0	February 2023	PHP/XML	155,259	2057	6
Online Food Ordering System	37.8 MB	2.0	January 2023	PHP/XML	131,173	1810	6
Raffle Draw System	149 KB	1.0	December 2022	PHP	770	18	4
Pizza Ordering System	24 MB	1.0	February 2023	PHP/XML	131,413	1812	8
Online Traffic Offense Management System	67.3 MB	1.0	February 2023	PHP/JS/XML	546,033	1967	3
Vehicle Service Management System	64.6 MB	1.0	September 2021	PHP/JS/XML	545,768	1976	8
Eduauth	24.7 MB	1.0	February 2023	PHP/XML	139,159	1248	3
Judging Management System	4.17 MB	1.0	December 2022	PHP	48,570	109	12

5.2. Evaluation of Vulnerability Detection

In order to verify question 1, this experiment chooses sqlFuzz to compare with static analysis technology RIPS [13] and coverage-guided gray-box fuzz testing technology webFuzz. The main purpose of the comparison experiment with RIPS is to verify whether sqlFuzz, as a dynamic analysis technology, is better than static analysis technology RIPS in terms of false positive rate. The main purpose of the comparison experiment with webFuzz is to verify whether sqlFuzz can detect more vulnerabilities than the current state-of-the-art gray-box fuzz testing technology. Three techniques were used to test the same dataset in the same environment. Two fuzz testing techniques were used to test each web application individually for 50 h, while static analysis techniques were used to test the end. The experimental results are shown in Tables 2 and 3.

In comparison experiments with RIPS, the evaluation criterion is mainly the false positive rate (FPR), which is the ratio of false alarms to all alarms. The false positive rate can obviously reflect the control ability of experimental technology over false positives. The higher the false positive rate, the weaker the control ability. In the comparison experiment with webFuzz, the evaluation criterion is mainly the recall rate (R), which refers to the ratio of the number of true vulnerabilities (TP) to the actual number of vulnerabilities (i.e., the sum of TP and false negatives (FN)). The recall rate can directly reflect the ability of experimental technology to detect real vulnerabilities and, the higher the value, the more real vulnerabilities are detected. The specific formula is as follows:

$$FPR = \frac{FP}{TP + FP} \quad (1)$$

$$R = \frac{TP}{TP + FN} \quad (2)$$

Table 2. The comparative experimental results of RIPS and sqlFuzz.

Application	RIPS			sqlFuzz		
	TP	FP	FPR	TP	FP	FPR
Best POS Management System	0	33	1	5	3	0.38
Online Food Ordering System	1	26	0.96	5	2	0.29
Raffle Draw System	0	0	0	4	1	0.2
Pizza Ordering System	2	26	0.93	7	2	0.22
Traffic Offense	1	58	0.98	3	1	0.25
Vehicle Service	4	74	0.95	6	3	0.33
Eduauth	0	0	0	3	0	0
Judging Management System	0	0	0	8	2	0.2
TOTAL	8	217	-	41	14	-

Table 3. The comparative experimental results of webFuzz and sqlFuzz.

Application	webFuzz			sqlFuzz		
	TP	FP	R	TP	FP	R
Best POS Management System	3	3	0.5	5	3	0.83
Online Food Ordering System	3	4	0.5	5	2	0.83
Raffle Draw System	3	1	0.75	4	1	1
Pizza Ordering System	4	1	0.5	7	2	0.88
Traffic Offense	1	2	0.33	3	1	1
Vehicle Service	4	2	0.5	6	3	0.75
Eduauth	3	1	1	3	0	1
Judging Management System	7	2	0.58	8	2	0.67
TOTAL	27	16	-	41	14	-

We have analyzed the experimental results:

In terms of false positives, sqlFuzz significantly outperforms the static analysis tool RIPS. The reasons are as follows: Firstly, RIPS, as a static analysis technology, is inferior to dynamic analysis technology in false positive control from principle analysis, which is also proved by the experimental results. Then, sqlFuzz has a false positive rate as high as 0.38 in some test web applications, which is a relatively high false positive rate as a fuzzing test tool. The analysis shows that the main reason for the high false positive is that the vulnerability determination rules are not perfect. sqlFuzz believes that the sign of triggering sql injection vulnerability is the occurrence of syntax errors, and this judgment standard

is also adopted by many academic studies in the same field. However, this judgment logic is not completely objective, and non-vulnerability bugs can also cause syntax errors. This aspect will also serve as the next direction to improve the technology. Finally, looking at the overall situation, sqlFuzz controls the false positives better. We believe that the main reason is that, by directing the seeds towards potential taint execution points, the probability of generating low-quality seeds becomes lower.

In terms of real vulnerability detection capability, sqlFuzz detected the largest number of real vulnerabilities among the three vulnerability detection tools; a total of 41 real vulnerabilities were detected; the improved webFuzz detected 27 and RIPS detected the least number of real vulnerabilities, 8. We analyze the reasons why sqlFuzz performs better than webFuzz, and the main reason may be that webFuzz takes a coverage-guided strategy, which may not necessarily find all execution points in a limited time. sqlFuzz obtained all potential vulnerability execution points and taint propagation paths as comprehensively as possible through static analysis. Under the guidance of the target, these potential vulnerabilities can be verified more quickly and efficiently.

5.3. Ablation Experiment

sqlFuzz uses static analysis techniques to identify potentially vulnerable code areas in web applications and employs instrumentation to provide feedback to the fuzzer, thereby guiding the seed generation process. This is mainly applied in the seed selection and seed mutation stages. In order to clearly demonstrate the impact of the vulnerability-guided approach on the efficiency of fuzz testing, a set of ablation experiments was designed in this study. Under the same experimental conditions, different configurations of sqlFuzz were used to test the same dataset. To fully understand how vulnerability-guided methods enhance efficiency, four different configurations of sqlFuzz were set up: the complete version of sqlFuzz; sqlFuzz-ns, which avoids feedback from instrumentation during seed selection; sqlFuzz-nm, which avoids feedback from instrumentation during seed mutation; and sqlFuzz-null, which has no feedback from instrumentation.

Method: Testing was conducted on the four versions of sqlFuzz in the same environment and dataset. Within 50 h, the number of vulnerabilities detected by each tool was recorded every 10 h. The experiment was conducted simultaneously on eight commercial web applications, and the results represent the total number of vulnerabilities detected across the eight applications. The experimental results are shown in Figure 5.

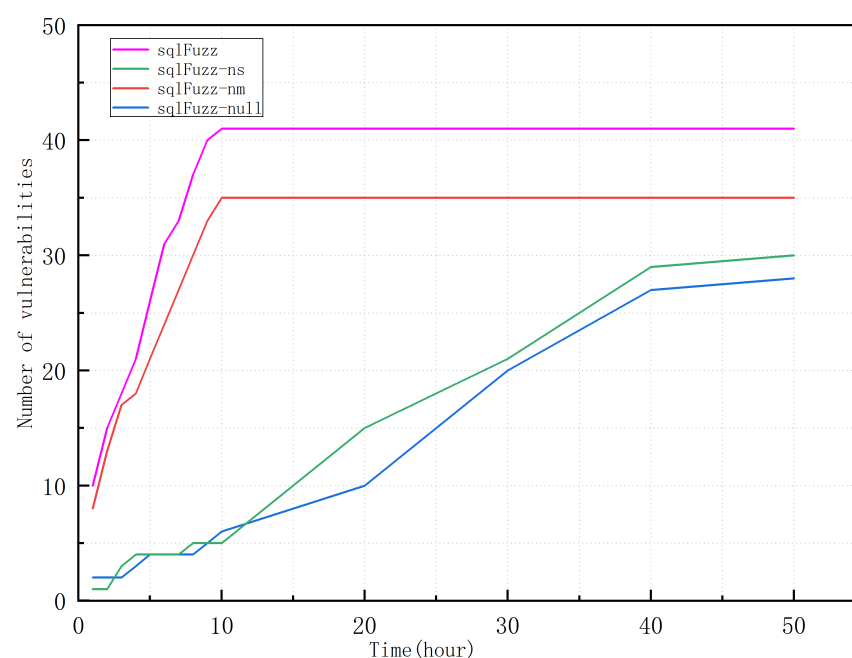


Figure 5. Experimental results graph for efficiency improvement assessment.

The experimental results indicate that: (1) In terms of the total number of vulnerabilities detected, sqlFuzz detected the most vulnerabilities, finding 41 out of 50 known vulnerabilities. Following this, sqlFuzz-nm detected 35 vulnerabilities, sqlFuzz-ns detected 30 vulnerabilities, and, lastly, sqlFuzz-null detected 28 vulnerabilities. From the data, it is evident that, firstly, vulnerability-guided methods can help uncover more vulnerabilities, as has been demonstrated in vulnerability detection capability assessment experiments. (2) In terms of efficiency, sqlFuzz and sqlFuzz-nm have a fast vulnerability detection speed and basically complete the detection task in the first 10 hours, while the sqlFuzz-ns and sqlFuzz-null vulnerability detection speed is slow. This result fully proves that the method based on potentially vulnerable code orientation has a significant effect on the efficiency of fuzz testing.

We have analyzed the experimental results:

Firstly, we analyzed the reason for the disparity in the number of vulnerabilities detected between sqlFuzz and sqlFuzz-null. Following the ablation experiment, we conducted supplementary experiments on sqlFuzz-null, continuing fuzz testing after 50 h, and, still, only a few vulnerabilities were discovered. The supplementary experiments indicate that the fundamental reason affecting the number of vulnerabilities discovered is an efficiency issue, meaning that, within a limited time, the vulnerability discovery capability of sqlFuzz-null is significantly weaker than that of sqlFuzz.

Secondly, the setup and comparison between sqlFuzz-ns and sqlFuzz-nm were primarily aimed at understanding in which stage vulnerability-guided methods effectively enhance the efficiency of fuzz testing. From the experimental results, it is clear that the performance of sqlFuzz-nm is significantly better than that of sqlFuzz-ns; therefore, vulnerability-guided methods have a greater impact on seed selection. The reason for this is that, in the vulnerability-guided fuzz testing, guiding the seed selection essentially allocates the energy of fuzz testing to areas more likely to trigger vulnerabilities, while guiding seed mutation is mainly aimed at better bypassing sanitization functions in web applications. When seeds are guided to potentially vulnerable code areas, the energy of the fuzzer is focused on breaking through nodes in the propagation path and, with the assistance of instrumentation feedback, more energy is allocated the closer it gets to the sink points. This feedback mechanism maximizes the efficiency of fuzz testing.

6. Discussion

6.1. Limitation

sqlFuzz primarily focuses on SQL injection vulnerabilities, but there are still many other types of vulnerabilities in web security that are worth attention, for example, XSS vulnerabilities, upload vulnerabilities, command execution vulnerabilities, file inclusion vulnerabilities, and other similar vulnerabilities. These vulnerabilities all belong to the category of tainted data vulnerabilities and share many similarities in analytical approaches. Therefore, in theoretical terms, sqlFuzz can be expanded to encompass the detection of these vulnerabilities.

sqlFuzz is unable to perform fuzz testing on Single Page Applications (SPAs). These types of web applications heavily rely on JavaScript, and the server responses are usually not presented in HTML format. Because sqlFuzz does not execute client-side JavaScript code, which in SPA applications is fully responsible for the creation and rendering of HTML documents, sqlFuzz is therefore not applicable in such scenarios.

6.2. Future Work

1. In the preprocessing phase of our work, we need to conduct static analysis on the target web application in order to identify the potential locations of vulnerabilities. In this phase, we have drawn inspiration from the static analysis technology RIPS. Despite being the most advanced static analysis technology currently available, RIPS tends to generate a relatively high number of false positives and false negatives due to its use of regular expression matching to locate sinks and its lack of support for object-oriented

features. We have already begun addressing some of these issues in our other work [14]. Comprehensive improvement in static analysis technology to enhance the accuracy of program analysis and reduce false positives and false negatives is a crucial means of advancing guided fuzz testing techniques, and it can be considered a focal point for future work.

2. The components of the initial seed are mainly the valid elements of the target web application crawled by the crawler and the dictionary of payload obtained online. As a result, the limitation of the initial seed is great. Whether the vulnerability can be triggered depends on whether the dictionary contains the payload that can trigger the vulnerability. Current large models are extremely capable of learning and can learn rules to directly generate well-formed statements. In the next work, we can try to use large models to generate initial seeds with more variety and range, thereby improving the quality of the initial seeds.

In summary, gray-box fuzz testing technology remains a focal point of current research in the fuzz testing field, with the goal of developing more comprehensive and efficient techniques in the future. This includes enhancing static analysis techniques to pinpoint as many potential vulnerable code locations as possible, and improving seed generation techniques to increase the likelihood of triggering vulnerabilities with test cases.

6.3. Related Work

Fuzz testing has become an indispensable vulnerability discovery tool for security researchers in recent years, serving as a significant driver behind the surge in vulnerability disclosures. However, fuzz testing tools exhibit a certain degree of blindness and randomness in various stages such as seed generation, selection, mutation, and feedback, leaving substantial room for improvement in vulnerability discovery efficiency. Existing research aimed at enhancing code coverage in fuzz testing can be categorized into the following areas: improvements in mutation strategies, enhancements in seed scheduling strategies, and advancements in program feedback.

6.3.1. Improving Mutation Strategies

1. Improvements to the mutation strategy itself

The issues with the current mutation strategy in fuzz testing, namely the time-consuming nature of deterministic mutation strategies and the overly random nature of non-deterministic mutation strategies, are currently being addressed through two main approaches. First, there is the improvement in deterministic mutation strategies, with the current predominant method being to skip this stage altogether [15,16]. However, bypassing deterministic mutation strategies directly can increase the inherent randomness of non-deterministic mutation strategies. Second, there is the enhancement of optimal mutation operation selection within non-deterministic mutation strategies.

In response to the issue of AFL's random mutation lacking target guidance, Lemieux et al. [16] proposed FairFuzz. FairFuzz first identifies path branches that are infrequently hit by input during the fuzz testing process, labeling them as "rare" branches. Subsequently, FairFuzz determines which bytes in the input are associated with these rare branches and restricts changes to these bytes during mutation, ensuring that the mutated test cases can hit the rare branches and thereby enabling thorough testing within these rare branches.

In response to the issue of low path discovery efficiency caused by fuzzers like AFL using a fixed distribution of mutation operations for operation selection, Lv et al. [17] proposed a mutation algorithm based on Particle Swarm Optimization (PSO). This algorithm considers the scheduling of mutation operations as a problem of finding the optimal probability distribution for mutation operations. However, this method only addresses the selection of mutation operations, neglecting the choice of the number of mutation operation combinations during the random mutation process. To address this, Wu et al. [18] introduced Havoc-mab. Havoc-mab models the selection of the number of mutation op-

eration combinations and the mutation operations during the random mutation phase as a two-layer slot machine model and dynamically adjusts the mutation strategy using the UCB1 algorithm to improve path coverage.

Lv et al. [19] observed that historical mutation data in fuzz testing can enhance the generation of effective test cases during the mutation phase. However, existing fuzzers have not effectively utilized this information. To address this, they proposed a new history-guided mutation framework that captures historical byte-level mutation strategies to guide subsequent mutation processes, thereby increasing the number of generated test cases that can improve coverage.

2. Enhanced mutation strategy

Due to the presence of path constraints such as magic numbers [20], checksums [21,22], and conditional statements [23] in the program, existing mutation strategies struggle to generate test cases that satisfy these constraints, thus failing to cover paths protected by these constraints (resulting in low code coverage). Current solutions to this issue mainly include: (1) symbolic execution, which uses constraint solvers to solve complex path constraints; (2) taint analysis, which can establish the relationship between input bytes and branch constraints, enabling targeted mutation.

Stephens et al. [24] first combined selective concolic execution with fuzz testing to address the challenge of fuzz testing failing to pass through path constraints such as programmatic validity checks. They used fuzz testing to explore program space and selective concolic execution to resolve path checks that were difficult for the fuzzer to navigate, thereby achieving the effective testing of the target program.

6.3.2. Improving Mutation Strategies

Seed scheduling strategy is an important component of fuzz testing, consisting of two parts: seed selection and energy allocation. American Fuzzy Lop (AFL) [25] selects seeds from the seed queue in order of edge number value, executing the optimal seed for each edge (i.e., short execution time and small size). AFLFast [26] selects seeds based on the priority of seeds that have been selected fewer times in the seed queue and seeds that have generated fewer test cases when exploring the same path previously. Its basic idea is to model fuzz testing using a Markov chain, where the execution paths of inputs are seen as states, and the transitions between states are seen as new inputs generated through the mutation phase. It selects seeds by calculating the transition probabilities between all states. In addition, some research work selects seeds based on uncovered branches along the test case execution path [21,27], the number of path executions [16], and control flow graphs [28]. The energy of a seed represents its execution time during the non-deterministic mutation phase, and the seed's energy is controlled by the seed's score.

In fuzz testing, AFL lacks a scientifically grounded theoretical model for energy allocation, resulting in the generation of a large number of test cases that execute the same paths in the target program. To address this, Böhme et al. [26], based on modeling fuzz testing using a Markov chain, proposed an algorithm for monotonic energy allocation. This allows seeds that execute low-frequency paths (i.e., executed fewer times) to be allocated more energy in fuzz testing. Compared to the monotonic energy allocation algorithm proposed by Böhme et al. [26], the adaptive scheduling energy allocation algorithm introduced by Yue et al. [29] is capable of effectively reducing energy consumption while achieving high path coverage in fuzz testing.

However, the coverage-guided fuzz testing mentioned above tends to allocate excessive energy to code areas that are less likely to contain bugs. To address this, Böhme et al. [26] utilized the distance-based differentiation of input reaching the target code areas and employed a simulated annealing-based energy allocation strategy. This approach effectively directs a significant amount of energy expenditure toward covering target points (potential vulnerability areas).

6.3.3. Improving Program Feedback

Program feedback is used to assess the quality of test cases and guide the seed scheduling and mutation process in fuzz testing. Due to the limited bitmap space for tracking code coverage in AFL and the use of a fixed hash formula to compute edge hash values, it is easy for different edges to have the same hash value (i.e., hash collisions). To address this, Gan et al. [30] reduced hash collisions by providing a larger bitmap space and using different hash formulas for different edges.

Meanwhile, due to AFL's context-insensitive edge coverage, it is unable to differentiate the same branch executed in different contexts. To address this, Chen et al. [31] resolved this issue by extending edge coverage to context-sensitive branch counting.

Furthermore, when a fuzzer encounters difficult-to-cover paths, code coverage alone cannot guide the fuzzer to explore these paths, as code coverage only indicates whether the current path has been executed. To address this, Aschermann et al. [32] designed a set of source code annotation primitives (a few lines of patch code) that testers can use to intervene in the fuzzer's feedback mechanism, enabling the fuzzer to explore these hard-to-reach paths.

7. Conclusions

SQL injection vulnerability detection is essential to web security. Fuzz testing technology is an important means for detecting SQL injection vulnerability. Aiming at the problems of blind seed selection guided by coverage and the low quality of generated test cases due to the great randomness of the seed variation mechanism in current fuzz testing technology for SQL injection vulnerability, a fuzz testing technology based on potentially vulnerable code guidance was proposed. The technology first uses static analysis technology to determine the taint propagation path and mark it as a potentially vulnerable code area, then instruments the potentially vulnerable code, and finally uses the feedback information to guide seed selection during operation, improving the priority of the seeds that have reached the potentially vulnerable code area and giving more energy. In addition, we designed an adaptive seed variation mechanism, and adopted different seed mutation methods according to different locations of seeds to improve the pertinence of seed mutation. Based on the above techniques, we implemented the sqlFuzz prototype system and used this system to analyze eight modern PHP applications. The experimental results show that sqlFuzz not only improves the efficiency of SQL injection vulnerability fuzz testing technology but also detects more SQL injection vulnerabilities with a lower false positive rate.

Author Contributions: Conceptualization, Y.Y. and Y.L.; Methodology, Y.Y. and K.Z.; Validation, Y.L.; Investigation, Y.C. and Y.Z.; Resources, Y.L.; Data curation, K.Z.; Writing—review & editing, Y.L. and H.H.; Supervision, H.H.; Project administration, H.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Johnson, B.; Song, Y.; Murphy-Hill, E.; Bowdidge, R. Why don't software developers use static analysis tools to find bugs? In Proceedings of the 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 672–681.
2. Alhuzali, A.; Eshete, B.; Gjomemo, R.; Venkatakrishnan, V. Chainsaw: Chained automated workflow-based exploit generation. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 641–652.

3. Artzi, S.; Kiezun, A.; Dolby, J.; Tip, F.; Dig, D.; Paradkar, A.; Ernst, M.D. Finding bugs in web applications using dynamic test generation and explicit-state model checking. *IEEE Trans. Softw. Eng.* **2010**, *36*, 474–494. [\[CrossRef\]](#)
4. Seal, S.M. *Optimizing Web Application fuzzing with Genetic Algorithms and Language Theory*; Wake Forest University: Winston-Salem, NC, USA, 2016.
5. Pham, V.T.; Böhme, M.; Santosa, A.E.; Căciulescu, A.R.; Roychoudhury, A. Smart greybox fuzzing. *IEEE Trans. Softw. Eng.* **2019**, *47*, 1980–1997. [\[CrossRef\]](#)
6. Gauthier, F.; Hassanshahi, B.; Selwyn-Smith, B.; Mai, T.N.; Schlüter, M.; Williams, M. Backrest: A model-based feedback-driven greybox fuzzer for web applications. *arXiv* **2021**, arXiv:2108.08455.
7. van Rooij, O.; Charalambous, M.A.; Kaizer, D.; Papaevripides, M.; Athanasopoulos, E. webfuzz: Grey-box fuzzing for web applications. In Proceedings of the Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, 4–8 October 2021; pp. 152–172.
8. Trickel, E.; Pagani, F.; Zhu, C.; Dresel, L.; Vigna, G.; Kruegel, C.; Wang, R.; Bao, T.; Shoshitaishvili, Y.; Doupé, A. Toss a fault to your witcher: Applying grey-box coverage-guided mutational fuzzing to detect sql and command injection vulnerabilities. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–25 May 2023; pp. 2658–2675.
9. Zhao, J.; Lu, Y.; Zhu, K.; Chen, Z.; Huang, H. Cefuzz: An directed fuzzing framework for php rce vulnerability. *Electronics* **2022**, *11*, 758. [\[CrossRef\]](#)
10. Clarke, J. *SQL Injection Attacks and Defense*, 2nd ed.; Tsinghua University Press: Beijing, China, 2014; pp. 7–8.
11. AFL. Available online: <https://afl-1.readthedocs.io/en/latest/> (accessed on 15 June 2024).
12. AFLGo. Available online: <https://github.com/aflgo/aflgo> (accessed on 15 June 2024).
13. Dahse, J.; Holz, T. Simulation of Built-in PHP Features for Precise Static Code Analysis. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2014; Volume 14, pp. 23–26.
14. Yuan, Y.; Lu, Y.; Zhu, K.; Huang, H.; Yu, L.; Zhao, J. A Static Detection Method for SQL Injection Vulnerability Based on Program Transformation. *Appl. Sci.* **2023**, *13*, 11763. [\[CrossRef\]](#)
15. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M. {AFL++}: Combining incremental steps of fuzzing research. In Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20), Online, 11 August 2020.
16. Lemieux, C.; Sen, K. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, Montpellier, France, 3–7 September 2018; pp. 475–485.
17. Lyu, C.; Ji, S.; Zhang, C.; Li, Y.; Lee, W.H.; Song, Y.; Beyah, R. {MOPT}: Optimized mutation scheduling for fuzzers. In Proceedings of the 28th USENIX Security Symposium (USENIX Security 19), Santa Clara, CA, USA, 14–16 August 2019; pp. 1949–1966.
18. Wu, M.; Jiang, L.; Xiang, J.; Huang, Y.; Cui, H.; Zhang, L.; Zhang, Y. One fuzzing strategy to rule them all. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 1634–1645.
19. Lyu, C.; Ji, S.; Zhang, X.; Liang, H.; Zhao, B.; Lu, K.; Beyah, R. EMS: History-Driven Mutation for Coverage-based Fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 24–28 April 2022.
20. Li, Y.; Chen, B.; Chandramohan, M.; Lin, S.W.; Liu, Y.; Tiu, A. Steelix: Program-state based binary fuzzing. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Paderborn, Germany, 4–8 September 2017; pp. 627–637.
21. Rawat, S.; Jain, V.; Kumar, A.; Cojocar, L.; Giuffrida, C.; Bos, H. VUzzer: Application-aware Evolutionary Fuzzing. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017; Volume 17, pp. 1–14.
22. Wang, T.; Wei, T.; Gu, G.; Zou, W. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In Proceedings of the 2010 IEEE Symposium on Security and Privacy, Berkeley/Oakland, CA, USA, 16–19 May 2010; pp. 497–512.
23. Chen, P.; Liu, J.; Chen, H. Matryoshka: Fuzzing deeply nested branches. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 499–513.
24. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In Proceedings of the NDSS, San Diego, CA, USA, 21–24 February 2016; Volume 16, pp. 1–16.
25. American Fuzzy Lop. Available online: <https://lcamtuf.coredump.cx/afl/> (accessed on 15 June 2024).
26. Böhme, M.; Pham, V.T.; Roychoudhury, A. Coverage-based greybox fuzzing as markov chain. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 1032–1043.
27. Zhang, K.; Xiao, X.; Zhu, X.; Sun, R.; Xue, M.; Wen, S. Path transitions tell more: Optimizing fuzzing schedules via runtime program states. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 1658–1668.
28. She, D.; Shah, A.; Jana, S. Effective seed scheduling for fuzzing with graph centrality analysis. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 23–25 May 2022; pp. 2194–2211.
29. Yue, T.; Wang, P.; Tang, Y.; Wang, E.; Yu, B.; Lu, K.; Zhou, X. {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 2307–2324.
30. Gan, S.; Zhang, C.; Qin, X.; Tu, X.; Li, K.; Pei, Z.; Chen, Z. Collafl: Path sensitive fuzzing. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–23 May 2018; pp. 679–696.

31. Chen, P.; Chen, H. Angora: Efficient fuzzing by principled search. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; pp. 711–725.
32. Aschermann, C.; Schumilo, S.; Abbasi, A.; Holz, T. Ijon: Exploring deep state spaces via fuzzing. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18–21 May 2020; pp. 1597–1612.

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.