Effective Logical Regression Testing in Database Management Systems

Alejandro Cuadron and Nicholas Thumiger

1 ABSTRACT

This paper introduces an innovative approach to regression testing in Database Management Systems (DBMSs), with a specific focus on MariaDB instances. The authors present a newly developed tool designed to compare the outputs of two different DB versions, effectively identifying potential discrepancies that may indicate regression bugs. The tool captures essential information such as the query, the current states of both databases, and the query output when a difference is detected. In addition to output comparison, the tool streamlines the development process and enhances bug detection by utilizing a novel coverage-based fuzzing technique. The authors have designed a differential coverage oracle that targets code directly modified between the two versions, aiming to provide comprehensive code coverage. This paper also addresses the implementation challenges associated with this approach, including aspects such as database instrumentation, differential testing implementation, oracle design and implementation, as well as results filtering and analysis. The authors assert that their approach effectively fills a gap in the current strategies for DBMS testing by offering a new tool and methodology for detecting regression bugs. This claim is supported by the authors' discovery of multiple verifiable real bugs in a short period of time, with noticeable differences observed when running the tool with and without the unique coverage oracle.

2 INTRODUCTION

Software systems are composed of a complex network of interconnected components, where even minor modifications can inadvertently impact seemingly unrelated code segments. Consequently, the concept of regression testing has emerged to ensure the continued proper functionality of previously developed and tested software following any changes [2].

Detecting bugs in the context of database management systems (DBMSs) presents a considerable challenge due to the intricacies of constructing an appropriate test oracle [9]. Variations in database states can lead to substantially different outputs from identical test cases, rendering automatic output validation nearly impractical. Moreover, identifying valid syntax for test cases proves to be difficult.

Automated Software Testing - ETHz, 2023, Spring Semester
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Extensive research has been devoted to creating test oracles for DBMSs, resulting in several proposed approaches. Initial strategies relied on random fuzzing and techniques such as Pivoted Query Synthesis (PQS) [9] implemented through generative tools like SQLancer. More recent methods include Query Plan Guidance (QPG), which directs automated testing towards relevant test cases [7], and coverage-based guidance employing tools like Squirrel [12] for path exploration. Although each of these methods exhibits strengths, they also possess limitations, particularly in addressing the variability of database states and guiding test cases towards bug detection. Notably, there is a lack of effective strategies for guiding test cases to detect regression bugs, highlighting a gap in the existing literature.

To bridge this gap, we have developed a novel tool capable of uncovering regression bugs between different versions of DBMSs. This tool combines techniques that have proven highly effective in identifying logical-based bugs, leading researchers to apply them to the detection of performance-based bugs as well. Notably, tools such as AMOEBA have emerged, revealing several performance bugs [6]. Consequently, leveraging these successful techniques enables efficient detection of logical bugs through regression testing across different versions. Building upon previous studies and tools such as SQLRight [7], which demonstrate that the combination of coverage-based guidance and oracle queries yields the discovery of more logical-based bugs, the tool developed in this paper consolidates existing strategies into a single differential testing tool. While this paper focuses on distinct MariaDB instances, the approach can be applied to any relational database.

The tool compares the outputs of two different DBMS versions, effectively identifying potential discrepancies. Upon detecting a difference, the tool captures the query, the current states of both databases, and the corresponding output. These discrepancies may indicate logical bugs or standard crashes in either of the DBMSs, all of which are recorded for further analysis. To enhance the bug-finding process, the tool employs a novel and highly effective coverage-based fuzzing technique.

In this paper, we present our innovative approach to regression testing in DBMSs, discuss the challenges encountered during its implementation, and provide empirical evidence of its effectiveness in detecting regression bugs. We firmly believe that our approach fills a significant void in the current strategies for DBMS testing by offering a new tool and methodology for detecting regression bugs.

3 RELATED WORK

Software testing, and specifically regression testing, is a well-studied area of research, with numerous methodologies and tools having been developed. Regression testing aims to discover new bugs that have been introduced after modifications in the software [11]. However, applying this concept to the domain of Database Management Systems (DBMSs) brings its own unique set of challenges. There has been a plethora of related works focusing on testing DBMSs, but their approaches and objectives vary.

Differential testing has been a popular method of revealing bugs in DBMSs. It involves running the same test cases on different DBMSs or different versions of the same DBMS, then comparing the outputs. If the outputs do not match, there is likely a bug [10]. However, this approach is fundamentally limited as it can only reveal bugs that cause differences in output; it cannot identify bugs that cause erroneous output which is nonetheless consistent across all DBMSs being tested.

Random testing is another common technique used to uncover bugs in DBMSs. Randoop, for example, generates unit tests for object-oriented programs randomly [8]. Yet, its effectiveness in a complex environment like DBMSs is limited due to the nature of data dependencies and transactional operations.

Pivoted Query Synthesis (PQS) [9] was later proposed to overcome the limitations. By using generative tools like SQLancer, PQS synthesizes SQL queries that are guaranteed to have specific outcomes, thus creating a reliable test oracle. Despite the advantages of PQS, it does not guide test cases to expose bugs effectively, which is a crucial aspect in the realm of regression testing.

Recent studies have made strides in addressing this limitation. Query Plan Guidance (QPG) was introduced by Jinsheng et al. [5] to guide automated testing toward more promising test cases. QPG uses the database's query execution plan to generate test cases that are more likely to reveal bugs. In a similar vein, Squirrel developed an approach using coverage-based guidance for path exploration in DBMS testing [12], enhancing the effectiveness of differential testing.

Meanwhile, work by Chong et al. [3] explored the use of symbolic execution, a technique originally applied in general software testing, for detecting bugs in DBMS. Their approach, while significantly different, shares the same goal of achieving greater accuracy and effectiveness in DBMS testing. Other research, like that of Emmi et al., proposed the use of static analysis for DBMS testing, revealing its potential in identifying tricky concurrency bugs [4].

The work by Liang et al. combined the concepts of coverage-based guidance and oracle queries, leading to an improved discovery of logical bugs with SQLRight [7]. This combined approach was later adopted to discover performance-based bugs, giving birth to tools like AMOEBA [6].

Surprisingly, the realm of regression testing in Database Management Systems (DBMSs) has received limited attention in current research. While there is a wealth of studies exploring software testing methodologies and techniques, the specific domain of regression testing in DBMSs remains largely unexplored. Despite the critical role DBMSs play in managing and storing vast amounts of data, the focus of existing research has primarily been on functional testing, performance testing, and other aspects of DBMS testing. Regression testing, which is crucial for ensuring the stability and reliability of DBMSs following modifications or updates, has not been a primary focus. As a result, there is a significant gap in the literature regarding dedicated approaches, tools, and methodologies for regression testing in DBMSs. This gap highlights the need for further research in this area to address the unique challenges and requirements of regression testing in DBMSs, ultimately enhancing the quality and dependability of these systems.

The work presented in this paper builds upon all these prior studies, with a specific focus on this research gap. While we share the overall objective of improving DBMS reliability, our approach diverges by focusing specifically on regression testing. In particular, we propose a tool for discovering regression bugs in different instances of MariaDB, employing a coverage-based fuzzing technique to enhance the bug-finding process.

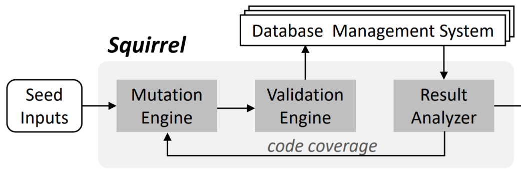

Figure 1: Design of Squirrel, used as this tool's base [12]

Our proposed tool and approach have two core contributions: First, we provide a generic test harness for differential testing of two different DBMS versions, leveraging the outputs of prior versions as a test oracle for newer versions. Second, we introduce a differential coverage metric to guide test-case generation, focusing on code that has been modified between versions. These contributions significantly extend existing work by addressing the challenge of regression bug discovery in DBMSs.

In summary, while various strategies for DBMS testing have been proposed in the literature, there remains a clear gap for techniques specifically targeted at regression testing. This paper seeks to fill this gap, providing a novel tool and approach for the detection of regression bugs in DBMSs.

4 APPROACH

There are two core contributions of this paper. The first is a generic test harness that can be used for the differential testing of two different database versions. Building on-top of existing work, it proposes using the outputs of prior versions of the database under test to act as the test oracle for newer versions. Secondly, this paper introduces and tests the performance of a 'differential coverage metric' that guides test-case generation to cover code that has changed between versions. This is hypothesized to improve the effectiveness of the test oracle by guiding the Fuzzer to produce test cases that specifically exercise the code that has been modified. Any regression bugs will be solely as a result of this *changed* code, and as such, it makes sense to target it.

These contributions are tested by implementing them for one specific database, namely MariaDB, and using them to detect differential bugs. They are built on-top of Squirrel and are designed such they can be easily adapted to other relational database types. The rationale behind this choice of database, and the usage of Squirrel is outlined in section 5.

A diagrammatic overview of the approach taken can be found in Figure 2. The strategy to implement this effective differential testing harness consisted of four components. Each required careful consideration and implementation on-top of Squirrel's existing work. At a high level, the seed inputs are syntactically mutated and validated by the existing Squirrel code base. The test cases are then

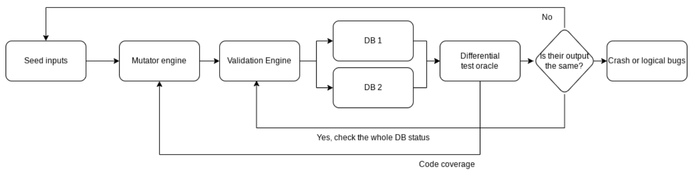

Figure 2: Implemented Methodology

passed into one or more databases, whose outputs are then collated and processed in a differential test oracle (described in section 4.3). The output of the test oracle determines whether the test case is a logical bug, a crash or valid. Finally the coverage of all the databases run is collected, processed using our coverage oracle, and fed back into the mutator to guide the next round of mutations.

4.1 Database Instrumentation

Firstly, the two versions of the database under test needed to be properly instrumented: this binary instrumentation allows the code coverage of the database to be ascertained, for each provided test case. Since one goal of the paper is to understand the effect of a differential coverage oracle, both databases need to be instrumented. Furthermore, to ensure compatibility with other databases without major code modifications, the instrumentation was to be done in such a way that no patches are required. As a result, it was decided to simply compile the database using the AFL compiler, with coverage enabled.

4.2 Differential Testing Implementation

In the first step, the two instrumented MariaDB databases are started, and a connection maintained. In this paper, versions 10.10 and 11.0 were selected, as the newer version of the two is currently in beta and has only recently been released. As a result, the chances of regression bugs existing are much higher. Despite this, the methodology here works for any arbitrary database versions as long as the same queries can be run on both.

The infrastructure of Squirrel was adapted to connect to multiple databases simultaneously; the queries are forwarded to all connected databases and the results are collated. For maximal versatility, there is no limit on the number of database versions that can be connected to, although this paper focuses only on two. The core challenge of this implementation phase is the reliability required. Since the testing could proceed for multiple days, with hundreds and thousands of test executions, memory leaks must be carefully avoided and the method in which the database connections are maintained must be reliable.

4.3 Oracle Design and Implementation

Next comes the main component of the approach: the test oracle. Currently, the Squirrel library overlays AFL++ and generates test cases using a custom mutator. These queries are then queued and managed by AFL++ which subsequently selects the next tests to run. These tests are forwarded to the multiple databases, and the results collected. A test oracle is *then* required to determine whether the outputs constitute a bug or not.

Currently, Squirrel and AFL are only designed to detect crash or hang bugs - where the test case causes the program to terminate prematurely or to take an unusually long time to execute. This paper introduces a third bug type: known as the ’differential bug’: an event that triggers when the output provided by any of the database versions under test differ. Such an oracle appears deceptively simple, as one simply needs to compare the string output of each query. However such an approach was found to be ineffective.

It is important to note that the test queries are designed to be independent and are intended to be run on an empty database. Consequently, each query consists of multiple SQL statements, and after each query the database is dropped and reset. Hence the only *output* that can be compared in the test oracle, is the output produced by a SELECT statement included in the test query. If no SELECT statement is fuzzed, then the query has no output. Furthermore, by nature of the fuzzing methodology, some statements within the test queries will yield syntax errors: even when syntactical mutation and validation is applied. When this occurs, the execution of the remaining statements will halt - including any ’SELECT’ statements that may occur later in the multi-line test query. As a result, additional queries need to be run after each test case to obtain a proper state comparison for a test oracle.

Another aspect considered is that of SQL errors. This approach compares the specific error obtained for a failed subquery and compares it with the corresponding errors across database versions. This ensures that if a test fails in the same place across two DB versions, but with a different error, that this is captured by the differential test oracle.

Consequently, for each test query, this paper collects the following information: the SQL status for each statement in the query, the received error message (if any), and a full output of the database. The latter is collected by running an additional query after the test to obtain the name of each table in the database, and to SELECT everything within them. This information is concatenated and compared across each tested DB version. If any difference is observed, this is recorded as a differential bug and is saved for further review.

One key challenge of this approach is that the order of SQL outputs is *not* guaranteed. As a result, when comparing the internal state of the database using the final SELECT statements, different output orders are categorized as a bug. This is mitigated by placing each returned row into an unordered set, and simply comparing the set. The second challenge is determining what is truly a regression bug. This is further discussed in section 4.5.

4.4 Coverage Oracle Design

The second contribution of this paper is the ’differential code coverage’ metric, used to guide the mutator into finding more relevant queries. Given that Squirrel simply runs on-top of AFL++, the existing coverage-guided fuzzing is rudimentary. It directs mutation by prioritizing inputs that increase the coverage, and mutating these inputs more often during the fuzzing process. While this is a simple approach, it has been shown experimentally to work well. Alternative methodologies are more complex and require a ’coverage aware’ mutation engine [7], which is out of the scope of this work.

This paper builds upon this by only using the edges of ’changed code’ to inform the mutation of the next test cases. As a result, an oracle needs to be developed that manipulates the coverage report to strip away the coverage of unrelated parts of code. Such an approach is very versatile, as it allows future users to further alter the coverage oracle to achieve different purposes - for instance, combining reports accross multiple DB versions.

4.5 Results Filtering and Analysis

Finally, the methodology needs a way to determine which of the detected ’differences’ are truly bugs. Initial tests encountered bugs that were simply a result of different default configuration parameters (i.e. maximum query length, and maximum timeout length) and a result of a different output order (mitigated with the map, discussed in Section 4.3). Manual effort is required to filter these out.

In addition to the spurious bug examples above, there are other categories of common bugs detected. Some ’differences’ between DB version outputs are repeatable and readily replicable: yielding logical differences in the outputs. These are classified as real bugs to be further investigated. Other ’differences’ cannot always be replicated, and depend on the load that the database is experiencing at the time the query is run. Since MariaDB is designed to be a wrapper for MySQL that performs well under high workloads and parallel processing, these are usually considered real bugs but are more challenging to replicate. These require further manual investigation to ensure that the behaviour is truly unexpected, and not an issue resulting from an external error unrelated to the test harness system or the database.

If the bug proves repeatable, he implementation generates and outputs several queries that lead to the same bug, enabling the developer to tackle the issue with a small replicable query.

5 IMPLEMENTATION

In this section, the physical implementation details and design decisions are presented. Overall, the aforedescribed methodology was implemented in C++ in a fork of the Squirrel repository. The final goal is to merge the changes presented here into the main branch, upon review from the repository maintainers.

By default, this modified version of Squirrel uses a single core. Although this is slower, it avoids a lot of the race conditions that could impact the integrity of the database during fuzzing - causing testing harness bugs that are not related to the database itself. Despite this, the single core allows the user to conduct multiple concurrent regression tests using different MariaDB versions.

Firstly, the decision to use Squirrel, as opposed to other tools such as SQLright are presented. Then, the challenges in understanding the Squirrel architecture are discussed, with a detailed explanation of this paper’s code architecture.

5.1 Choice of MariaDB and Squirrel

A suitable SQL-based DBMS was required in order to implement and test the proposed methodology. The criterion for selection required that the DB have a wide usage, an open-source status, and an active development community. These goals would ensure that our findings would assist the development of the database and make a meaningful contribution.

MariaDB was ultimately selected, given that it is one of the most popular open-source relational databases in use. Created by the original developers of MySQL and with the support of a more active and community-driven open-source development model, MariaDB is included in most cloud offerings and serves as the default choice in most Linux distributions [1]. A characteristic of note is that MariaDB runs *on-top* of MySQL, incorporating additional features and improvements that primarily aim to optimize performance and improve parallel data processing at scale.

As described in the literature review, no existing approach fully implements the bug-finding methodology introduced in this paper. As a result, it was important to select a suitable tool to use as a base for the required modifications. SQLancer and SQLsmith were deemed unsuitable due to their exclusive reliance on randomization and pre-designed rules for syntactical query generation. Instead, a tool was sought that already implemented coverage-guided fuzzing to improve the query generation [12]. This is still an active research direction, and hence it was out of the scope of this paper to design such a system.

In the quest to find a suitable tool, we also compared SQLRight with Squirrel, another coverage-guided fuzzer. Squirrel’s architecture, built on AFLplusplus, uses a yaml configuration file and environment variables for setup. Its native support for MariaDB and its ability to implement coverage-guided syntactical fuzzing aligned well with our specific strategies and methodologies.

SQLRight [7], with its implementation of both a test oracle and coverage-guided fuzzing, initially appeared as a viable option. However, its shortcomings, such as poor documentation, lack of out-of-the-box support for MariaDB, and reliance on an undocumented patch of the database under test for the coverage process, made it less suitable for our core goal. This goal is to create a versatile framework for testing different database versions without requiring additional modifications to the database under test, other than code instrumentation.

Despite SQLRight’s advanced features like validity-oriented mutations and oracles for detecting logical bugs, it fell short in critical areas. These shortcomings, coupled with the fact that SQLRight is built on top of Squirrel [12], led us to select Squirrel as the basis for our work.

Squirrel supports coverage-guided query generation for MariaDB natively and implements coverage-guided syntactical fuzzing. A diagrammatic outline of its operation can be found in Figure 1. However, it is not perfect. Squirrel uses AFL++ under the hood, manipulating the seeded inputs in the AFL queue by applying its own syntactical mutation engine. However, it heavily relies upon the underlying AFL++ library for the other code infrastructure, does not implement a testing oracle, and detects only crash bugs. Furthermore, the coverage retrieved from executing the instrumented database is only used to select which seed input is to be mutated and selected in the next round of fuzzing, re-using this strategy and code from the underlying AFL++ implementation.

In conclusion, despite SQLRight’s additional features and Squirrel’s reliance on AFL++, Squirrel’s architectural design, native support for MariaDB, and coverage-guided syntactical fuzzing made it the more suitable choice for our study.

5.2 Containerization

In order to test different MariaDB database versions for regression bugs, a docker container was created. Modifications were made to the existing container found within Squirrel, to ensure that Squirrel, AFL++ and two versions of MariaDB work together as expected. This section discusses the changes made to the Dockerfile and the reasons behind them.

The Dockerfile can be found in Appendix 7.1. It begins by setting up a common configuration, installing necessary packages and creating a user with sudo privileges. It then proceeds to install MariaDB server version 10.10. This is done by adding the MariaDB repository to the sources list, updating the package list, and building the dependencies for MariaDB.

The Dockerfile then clones the Squirrel repository, builds it with AFL++, and clones the MariaDB server repository, checking out version 10.10. It then builds MariaDB with AFL++ and installs it to /usr/local/mysql/.

At this point, the Dockerfile switches back to the root user to change the MariaDB repository in the sources list to version 11.0, updates the package list, and builds the dependencies for MariaDB again.

Switching back to the user, the Dockerfile checks out version 11.0 of the MariaDB server repository, builds it with AFL++, and installs it to /usr/local/mysql2/. It then changes the ownership of the installed MariaDB directories to the user and initializes the databases.

Finally, the Dockerfile sets an environment variable to ignore missing crashes, gets the map size of the AFL shared memory region, and sets it as an environment variable for running Squirrel.

The reason for these modifications is to allow testing of different MariaDB versions for regression bugs. By installing two different versions of MariaDB in separate directories, Squirrel can be run against each version to check for differences in behavior, which could indicate a regression bug. This setup also allows for easy switching between MariaDB versions for testing.

The development of the Dockerfile presented several intricate challenges, which were eventually surmounted through a comprehensive understanding of the interactions between AFL, Squirrel, and the selected DBMS.

The primary challenge was engineering a system capable of concurrently running multiple versions of MariaDB on the same machine. This task was complex due to the identical dependencies of both databases, each requiring a different version. Therefore, the installed dependencies had to be compatible not only with the specific version of MariaDB they were designed for, but also with the other version. Dependency mismatch was not an issue when comparing closely related versions, such as MariaDB 10.10 and MariaDB 11.0. However, when comparing different major releases, dependency mismatch could result in false positives or, in the worst-case scenario, a nonfunctional system.

This issue was resolved by modifying the sources.list file in the Ubuntu distribution and installing the required dependencies for both databases, thereby eliminating any arising dependency mismatches.

The second challenge pertained to the data path where MariaDB stored its data. It was crucial to ensure that the user "doBigThing" had access to the database data, as the modified Squirrel would use this user to access the data generated by the MariaDB databases and analyze each database's internal state. This was a departure from the default Squirrel, which was solely concerned with crashes and did not analyze the databases' internal state. Therefore, permission to access the DB data directory was granted to the user "doBigThing".

Subsequently, an unexpected behavior led to the crash of one of the MariaDB versions during boot-up. Examination of the logs from both MariaDB versions revealed that the issue was caused by an attempt to reuse the same UNIX socket. Even with only one client, the socket could not be reused to connect to a different DB. The initial solution was to disconnect from the DB and reconnect to the other DB, which proved effective but significantly increased the run time as it required waiting for the DB to restart before executing the query.

To optimize this strategy, we decided to increase the resources allocated to the container. This meant that both DBs would run concurrently as initially planned, but unlike the previous solution, every allocated disk space would be duplicated and assigned to each DB dynamically in the file. This approach and its implementation will be further elaborated in the implementation section.

5.3 Squirrel Architecture

Alongside the challenging work accompanying containerization, the architecture of Squirrel needed to be ascertained. An overview is provided here.

Squirrel uses the AFL shared memory API to operate, and essentially acts as the 'test harness'. When the run command is issued, Squirrel starts AFL with itself as the target to be fuzzed. Internally, it forks a process for AFL to run on, and then synchronously hooks into the AFL test cases by using the __afl_next_testcase() function. This function accesses AFLs shared memory and retrieves the desired test case. Upon receiving it from the AFL fork, it then redirects the AFL inputs to the database. After sending the query, a response is *not* requested from the database. Squirrel simply checks if the database crashed or not, returns this binary status to AFL and then unblocks - allowing the AFL fork to send the next test case. The run command is presented below, and from this it is clear that AFL is the main program that is being executed - Squirrel acts purely as a middle-ware between the database and AFL.

Listing 1: Squirrel Run Command

```
afl-fuzz -i fuzz_root/mysql_input/ -o /tmp/
    fuzz -t 60000 -- {ROOTPATH}/build/
    db_driver
```

In addition to this, Squirrel implements a mutator that hooks into an AFL callback for 'custom mutators'. This mutator is the core contribution of the Squirrel paper, and allows the prioritised queries (from AFL coverage) to be mutated according to the database syntax, using a technique introduced by the Squirrel team. For the purposes of our paper, this code is not altered and simply used as-is. The mutator is called whenever AFL modifies a test case in the queue.

In terms of coverage, Squirrel does very little. Despite making it a core component of the research paper, the Squirrel team simply uses the basic coverage feature of AFL. After redirecting the test case to the attached instrumented database, Squirrel then returns control to the AFL fork. AFL then reads the binary coverage output file from the execution of the instrumented database, and uses this information to guide which test cases to prioritise and mutate.

Consequently, the majority of the changes to meet the needs of this paper's methodology are to be placed in the synchronous code that processes the AFL test cases. For coverage, AFL itself needs to be modified.

5.4 Architectural Design

The majority of the code changes can be found inside the 'db_driver.c'. As described above, this is where the synchronous code that interfaces with AFL is situated.

The first key implementation was the integration of a second database. This was written in such a way that an unlimited number of databases could be attached to the fuzzing instance. To implement this, the database class and configuration was modified to accept an array of database ports, names and connection information. The user is then able to specify as many databases as desired. The main relevant code files are 'db_factor.cc', 'client_mysql.cc', 'db_driver.c' and 'mysql.cc'. Note that since MariaDB runs on mysql, the Squirrel code for mysql was primarily used.

The main event loop for fuzzing can be found in 'db_factor.cc'. A pseudo-code extract from this file can be seen in Listing 2. It begins by creating and initializing a vector called databases, which holds instances of DBClient representing different databases. Each database is created based on the provided configuration. To facilitate the fuzz testing, shared memory is initialized with a specific size using the __afl_map_shm() function, allowing efficient communication between the different components of the process. Under the hood, it uses the C library function 'shm_open(shm_file_path, O_RDWR, 0600);' to mount the shared memory with a file based API. Additionally, memory is allocated for a buffer called buf with a maximum size defined as kMaxInputSize, which will store the fuzz test cases.

Listing 2: db_driver.c Main Eventloop Pseudocode

```
vector<DBClient *> databases{};
for each database in config{
  db = create_client(db_name, config);
  db->initialize(config, db_num);
  databases.push_back(database)
}

// Initialise shared memory
__afl_map_size = MAP_SIZE;
__afl_map_shm();

// Shared memory buffer
u8 *buf = malloc(kMaxInputSize);

// Start each database, if not yet on
for each database in databases{
  if not database->check_alive(){
      database.startup()
  }
}

// Start AFL fork server
__afl_start_forkserver();

while (__afl_next_testcase(buf)) {
  vector<string> results{};
  for each database in databases{

      // Setup testing environment
      database->prepare_env();

      status = database->execute(buf)
      if (status == "kServerCrash") {
          database->startup()
      }

      // Retrieve results
      result = database->get_results();
      results.push_back(result);

      database->clean_up_env();
  }

  // Use the test oracle to check results
  status = test_oracle.check(results);

  // Report that test is complete
  __afl_area_ptr[0] = 1;

  __afl_end_testcase(status);
}
```

Next, each database in the databases vector is checked if it is live. If a database is not alive, indicating that it has not been started yet, the startup() method is invoked to initiate the database. The AFL fork server is then started using the __afl_start_forkserver() function. This fork server is responsible for efficiently spawning and managing multiple parallel instances of the fuzz test execution. Then the event loop starts - continuing until there are no more fuzz test cases provided by AFL. Within each iteration of the loop, a fuzz test case is retrieved from AFL, stored in the buf buffer using the __afl_next_testcase() function, and passed to the databases. The results are collected and passed to the test oracle (described later).

When the modified Squirrel tool is started, it receives some parameters so that the multiple DB versions can be set up and run in the same machine. Those parameters were predefined in the original Squirrel repo for a single DB. However, most of them are fixed. So, it was decided to redesign the configuration file to accommodate the use of several DB. Consequently, all the different fields of the config file were modified, and the configuration parser was similarly modified. We made this modification for all of the databases supported by Squirrel, such that the code can eventually be merged back into the main Squirrel branch.

5.5 Comparison of Configuration Files

To allow the deployment of multiple MariaDB servers on a single machine, the original configuration file was modified. The comparison below details these modifications and explains the rationale behind them.

5.5.1 Original Configuration File. The original configuration file was structured as follows:

```
required: ["init_lib", "data_lib", "db", "passwd",
"user_name", "host", "sock_path", "db_prefix",
"startup_cmd"]
```

This is a list of required fields for the configuration file.

```
should_exist: ["init_lib", "data_lib"]
```

This is a list of fields which should exist in the configuration file.

```
init_lib: /home/Squirrel/data/fuzz_root/mysql_init_lib
```

This is the path to the initialisation library for MySQL.

```
data_lib: /home/Squirrel/data/fuzz_root/
```

This is the path to the global data library for MySQL.

```
db: mysql
```

This specifies the type of the database.

```
passwd: ''
```

This is the password for the database.

```
user_name: dobigthing
```

This is the username for the database.

```
host: localhost
```

This specifies the host of the database.

```
sock_path: /tmp/mysql.sock
```

This specifies the socket path for the database.

```
db_prefix: test
```

This is the prefix for the database name.

```
startup_cmd: "/usr/local/mysql/bin/mysqld
--basedir=/usr/local/mysql
--datadir=/usr/local/mysql/data
--log-error=err_log.err
--pid-file=server_pid.pid
--max_statement_time=1 &"
```

This is the command to start the database server.

5.5.2 Modified Configuration File. The configuration file was modified to facilitate the simultaneous running of multiple MariaDB servers:

```
required: ["init_lib", "data_lib", "db", "passwd",
 "user_name", "host", "sock_paths", "db_prefix",
  "ports", "startup_cmd"]
```

The "ports" field has been added to the list of required fields, and many of the fields pertaining to the database have been pluralised - accepting a list of parameters instead of a single value.

```
sock_paths: ["/tmp/mysql.sock", "/tmp/mysql2.sock"]
```

"sock_paths" is now an array to allow for multiple socket paths for different DBs.

```
ports: ["20000", "30000"]
```

A new field "ports" has been added to specify the ports for the different DBs.

```
executables: ["/usr/local/mysql/bin/mysqld",
 "/usr/local/mysql2/bin/mysqld"]
```

A new field "executables" has been added to specify the executables for the different DBs.

```
basedirs: ["/usr/local/mysql", "/usr/local/mysql2"]
```

"basedirs" now uses an array to allow directories for different DBs.

```
datadirs: ["/usr/local/mysql/data", "/usr/local/mysql2/data"]
```

"datadirs" now uses an array to allow different data directories for the DBs.

```
pid_files: ["server_pid.pid", "server_pid2.pid"]
```

"pid_files" now uses an array to allow different PID files for the different DBs.

```
startup_cmd: " --log-error=err_log.err --max_statement_time=1 &"
```

The "startup_cmd" field has been simplified, relying on the preceding fields for execution details.

5.6 Test Oracle

The next implementation detail revolved around receiving responses from the databases. The original Squirrel version does not parse the error messages or the results of SELECT and UPDATE queries, wheras this is needed in a differential testing oracle.

Squirrel works by resetting the database before every query, and then issuing one single multi-expression query to the database. It checks if the database crashes and then restarts the event loop - similar to what was presented in the prior listing. However in the modified version for differential testing, it was necessary to compare the output of each query and the database itself to ensure that each query was run as expected. Differential testing is more interested in regression and logical bugs.

To do so, the C mysql api is utilised. The pseudocode logic of this is presented in the listing below, and is sourced from 'mysql_client.cc'. The key output of this code is the output_string variable which contains the number of rows affected by each query, the specific error message encountered when executing a specific sub-expression, and the result of any SELECT queries. This means that the 'output_string' variable contains a full log of what the full query executed on the database, and is a good candidate for comparison in the differential oracle.

Furthermore, using similar logic, after each query is executed, the entire contents of the database are added to the output_string. This is achieved by making one query to find all the created tables, and another query to 'SELECT *' from each table. The result is appended to the output_string.

This output string is used inside the differential testing oracle to determine if a bug was encountered. So as not to alter the implementation logic of AFL, the oracle is very simple. It simply compares the output_string of each database, and returns a "kServerCrash" status if any of them differ from each other; even though it is not actually a server crash.

This means that AFL behaves the same way as if a crash bug was detected. The 'Crash' counter on the AFL user interface is incremented, and the input that produced the bug is stored in the crash folder. This location is specified in the RUN command of Listing 1 (by default: /tmp/fuzz). Furthermore, additional code was implemented to output the value of output_string to the file in the event of a bug. This helps the user understand why the bug may have occurred, and whether it is real. The location of this additional file is /{workspace_dir}/scripts/utils/status_logs.txt. All of this code can be found within db_driver.c.

Listing 3: mysql_client.cc Result Processing Code

```
output_string = ""
do {
    // Retrieve the next result
    if (not first execution)
        status = mysql_next_result(db_con);

    aff_rows = mysql_affected_rows(db_con);
    result = mysql_store_result(db_con);

    // Check for errors
    err = mysql_errno(db_con);
    if (err) {
         err_str = mysql_error(db_con);
         output_string += err_str;
    }

    output_string += aff_rows;
    if (!result)
        continue;

    // Fetch query result rows
    while (row = mysql_fetch_row(result)) {
        lens = mysql_fetch_lengths(result);
        for (i : mysql_num_fields(result)) {
            output_string +=row[i];
        }
    }

    mysql_free_result(result);
    more = mysql_more_results(&(*connection)
        );
} while (more);
```

Future work can add an additional status type called 'kDifferentialBug' and handle this correctly within AFL's implementation. However this is more challenging, and is out of the scope of this project.

The main challenge of this implementation was not the complexity of the code itself - this was straightforward. Instead the complexity came from understanding the existing functionality and interaction between AFL and Squirrel, and properly injecting code to test out the methodology of this paper. Due to time constraints, the implementation is not production worthy, however it properly achieves the goals set out at the start of the project and achieves good results.

5.7 Coverage Oracle

A key implementation difficulty was the coverage oracle approach presented in this work. The initial plan was to modify the coverage report for each test case to include only the lines that were changed between the two versions of the database. As discussed in the architectural section, AFL does not produce a human readable coverage file upon each execution of the instrumented binary. Instead, it outputs a binary map, corresponding to each edge of the code.

This means there are two possible implementation options. The first was to manipulate the binary coverage report for each test in order to only include the coverage of the lines that were changed between the two DB versions. This can be done either by modifying the file that is generated, or by modifying the AFL coverage map that is in shared memory. This is stored within the shm_map shared object. This approach is very unfeasible, as it requires corresponding each location in the binary map with the location in code - which requires an indepth understanding of the compiler and the coverage option.

The alternate option would be to adjust the inner workings of AFL to ignore certain parts of the binary map when parsing it. Again this would require a high level understanding of how code regions in the compiled binary map to sections in the coverage map. Given the complexity of this endeavour, and the time constraints of the project, this was excluded from the scope of the project. Instead, a simpler but equally effective approach was formulated.

AFL instead offers a way to selectively instrument sections of code. The coverage of an edge can only be determined in instrumented code. As a result, if only the *changed* code is instrumented, this achieves the goal set out in the methodology section of this paper. In other words, if only non-instrumented code is run (i.e code that is exactly the same between database versions), then the coverage is listed as 0. The coverage only increases when the instrumented code is run.

To achieve this selective instrumentation, there are two main approaches. The first is to insert the macros __AFL_COVERAGE_ON() and __AFL_COVERAGE_OFF() into the source-code of the databases undertest. This is a fine grained approach that would offer the highest amount of control over what is instrumented. However, this is much less versatile and more difficult to implement - as the functions that are changed between versions need to be individually annotated with these macros. Furthermore, a stated goal of this paper is to make the work as versatile as possible, and to avoid editing the source code of the database under test.

As a result, the second approach was used. Here, the compiler option AFL_LLVM_ALLOWLIST can be set to a file containing a list of files. This tells the AFL compiler, when compiling the database under test, to only instrument the specific files listed. Using github, a list of *changed* files between the two versions can be generated. The code for this is presented below. Then, through this specific *whitelist* compilation command, it was possible to only include those specific changed files in the coverage instrumentation. As a result, AFL receives a full unmodified coverage report from the database under test, but it only reports the edge coverage of the whitelisted files. Although this approach is not polished, and not easy to extrapolate to other projects, it provided the best compromise between implementation complexity and results.

Listing 4: Whitelist File Generation

```
git diff --name-only mariadb-10.10.4 mariadb
    -11.0.1 | grep -E '\.(c|h)$' > whitelist
    .txt
```

A core disadvantage of the approach is that it is not possible to *combine* coverage reports from the different DB's under test. In this case, AFL only receives the *differential* coverage from the newer database, and there is no scope to do some pre-processing before AFL uses it to select the next test case.

The other disadvantage is that it is not fine grained. If only a single line is changed in a particular file between two versions, then the entire file is instrumented regardless. This turned out not to be a problem while testing the program. This is because there are 24088 code files in the newest version of MariaDB, and only 398 were modified between the two versions under test. This reduces the coverage space to only 1.6% of the code base - and this helpfully guides the fuzzer to cover the code that is most likely to produce bugs.

Further work can look into creating a script to optimally instrument the database for custom coverage oracles.

Note that this coverage methodology is included in the Dockerfile shown in Appendix 7.1, and included inside the provided docker image that accompanies the submission of this report.

5.8 Fuzzing Process

The task of identifying bugs through the process of fuzzing presented several challenges which required us to adapt our approach on the fly and offered opportunities for learning and improvement for future iterations.

The code behind the fuzzing approach is based entirely upon AFL++ and the original Squirrel's mutation engine. The mutation engine uses an intermediate representation (IR) designed to maintain SQL queries in a structural and informative manner. To generate syntactically correct queries, type-based mutations are performed on IR, including statement insertion, deletion and replacement. To mitigate semantic errors, each IR is analyzed to identify the logical dependencies between arguments, and queries are generated that satisfy these dependencies. AFL then uses a hybrid metric that combines edge coverage and hit counts to guide test case selection. In our case, this was based upon our modified instrumented version described in a previous section. When AFL discovers a new test case that triggers new coverage, it saves it as part of the test case queue.

Unfortunately, our initial efforts using the approach above, coupled with our unique coverage metric, weren't as successful as we'd hoped due to a lack of refinement in our fuzzing method.

A significant early setback arose from an issue with the CPU governor. The CPU governor is a system or subsystem that modulates the operating frequency of the central processing unit(s) (CPUs) in response to the current processing demand and power constraints. A team member was utilizing a laptop with a new architectural design from Intel which led to false crashes in the American Fuzzy Lop (AFL). After resolving this issue, we proceeded with fuzzing.

In order to verify our approach, the system was initially tested against itself by using the same DB version, in this case, MariaDB 10.10, in a duplicated setup. This proved to be quite effective as it led to the identification of errors in our implementation, such as the Unix socket issue. In essence, we were running differential testing on our own code! Additionally, we discovered that the program was prone to severe memory leaks as indicated by the Address Sanitizer (ASAN). After a few days of operation, the program crashed due to exhausting the available RAM and Swap space on the laptop.

Despite our best efforts to rectify the arising issues in our code, the memory leak bug proved intractable when the system is run in 'debug' mode, with vscode attached. When run separately as an independent docker container, the system is able to run uninterrupted for many consecutive days on a laptop equipped with 32GB of RAM, which was a significant achievement. A limit to the runtime has not yet been discovered - the maximum time we ran the fuzzer was 6 days for test purposes.

Initially, our strategy was somewhat naive, letting the service run for several days in the hope of unearthing bugs. All crashes were logged to /tmp/fuzz, but we neglected to capture the internal state of both DBs or the cause of the error, resulting in a rather unsatisfactory bug report. A number of bugs were detected and seemed legitimate, with most discoveries occurring shortly before the testing suite crashed.

Upon restarting the service, we attempted to replicate the bugs by using the queries that seemed to have caused them as initial seeds. However, the testing software did not report any crashes. Consequently, our strategy shifted: two fresh DBs were spun up and the suspected bugs were run against these DBs directly, using the latest stable version of the MariaDB client. This was achieved programmatically by using a script we wrote called "are_these_bugs.py". This script, which is included in the submission zip file, takes a suspected buggy query as input, and runs them on both database versions configured. It then outputs the differences between the DB outputs (if any).

These early bugs could not be replicated when run against the DBs, prompting us to discard them on the suspicion that the errors lay within our testing harness. Despite these challenges, it was essential to conduct a significant amount of bug testing on the testing service. It was decided that the internal state of the DB at the time of the error and the error message itself should be captured. This was implemented, and the testing software was run once more for several days. The software found no bugs but still crashed after running continuously for a few days. When restarted, it again found no bugs.

Following significant computational effort and yielding no results, we considered resolving the memory issues. However, we hypothesized that the issue might stem from our reliance on seed data that Squirrel founders considered optimal. We thought modifying the content of these seeds might offer a more successful approach.

The testing software was run for several days until it crashed. Each time, a new seed was introduced, but this approach yielded no results. We were uncertain as to how to modify the seeds to increase the bug count, and at times it appeared that the testing software's performance was deteriorating.

Instead of altering the content of the seeds, we decided to increase their quantity until it appeared to be detrimental to the testing infrastructure. This approach led to some intriguing bug discoveries, which will be detailed later. However, the number of bugs found was not sufficient for the successful completion of the project. Interestingly, this strategy identified over one hundred bugs. Unfortunately, when we tested the validity of these bugs, the system was unable to confirm their replicability.

Subsequent testing suggested that a more effective approach would be to dynamically change the contents of the seeds when a potential bug was identified. The service could then explore the replicability of the bug and validate its legitimacy by discovering similar bugs or those arising from the same root cause.

After implementing dynamic seed adjustment and restoring the default seed values, the service was restarted. After several days of operation without crashes, the system reported numerous bugs. These bugs were initially assumed to be more replicable because some had the same root cause, as indicated by the modifications made to the testing software.

The system identified several bugs that seemed legitimate and replicable in MariaDB 11 against MariaDB 10.10. We decided that our method for testing the validity of these bugs was insufficient, prompting the development of a new confirmation method. Each bug belonging to a unique category would be tested against a fully operational release version of a syntactically similar DBMS. MySQL was chosen for this purpose due to its compatibility with MariaDB and its wider array of features and customization options.

Given a suspected 'buggy' input that causes output differences between the two tested versions of MariaDB, we needed a way to determine which version is erroneous. For this, the query is converted in a similar query that is valid in the newest version of MySQL. If the behaviour of the output is the same as the old version of mariaDB, then we know that we have identified a regression bug. If the output is the same as the new MariaDB version, then the old version of MariaDB is the buggy version, and has been since resolved.

The testing process revealed that many of the identified bugs were not truly replicable, even though they initially appeared so. For instance, a bug was suspected in the conversion of a string to an integer in MySQL, which led to a division by zero scenario as shown in Figure 3. This was later identified as a configuration difference in the MySQL interpreter. MariaDB 11 uses a non-string to integer conversion; meaning, when comparing a string to an integer, an error is returned instead of converting the string to an integer and returning zero if the string cannot be converted. Conversely, MariaDB 10 does perform this string to integer conversion by default. When this scenario arose, both DBMS returned a different result. However, this does not imply that it is a regression bug, as string to integer conversion is a configurable variable that can be enabled or disabled according to user needs. Interestingly, the default value for MariaDB 11 differs from that of MariaDB 10.

```
CREATE TABLE v0 ( v1 TEXT , v2 CHAR ( 16 ) ) ;
INSERT INTO v0 ( v1 ) VALUES ( 16 ) ;
UPDATE v0 SET v1 = 15 WHERE
( SELECT v2 FROM v0 AS v3 )
IN ( SELECT CASE
WHEN v1 % 0 != -128 THEN v2
WHEN 28 THEN 'x'
WHEN 16 THEN 'x'
ELSE 80 / 0
END
FROM v0 ) ;
INSERT INTO v0 SELECT * FROM v0 WHERE 0 > 0 ;
```

Figure 3: Bug in String to Integer Conversion and Division by Zero (MariaDB)

A second *example* suspected bug was related to the return order of SELECT queries, as shown in Figure 4. Upon further investigation into the MariaDB documentation and release notes, it was determined that the order of the SELECT statement is not guaranteed, but the content is. As such, as long as the content remains consistent, the order of returned elements does not matter. Therefore, this was not deemed to be a real bug. Note that despite our code designed get around this non-guaranteed order fact, that uses a *map* to compare outputs of SELECT statements, this bug was not prevented. This is because the select statment is inside the subquery, and is not affected by our code. There is no easy way to programmatically filter out false bugs like this.

```
CREATE TABLE v0 ( v1 INT ) ;
INSERT INTO v0 ( v1 ) VALUES ( 56 ) ;
CREATE TEMPORARY TABLE v0_tmp1 AS SELECT * FROM v0;
UPDATE v0 SET v1 = 76 WHERE
( v1 BETWEEN
( SELECT v1
FROM v0_tmp1 AS v2
WHERE v1 > 96 OR -1 ^ 56 + 71 ^ 0 * 41557549.000000
GROUP BY v1 )
AND v1 , v1 , v1 )
< ( 45 , 2 , 44 ) ;
INSERT INTO v0 ( v1 ) VALUES ( 0 ) , ( 84 ) ;
CREATE TEMPORARY TABLE v0_tmp2 AS SELECT * FROM v0;
CREATE TEMPORARY TABLE v0_tmp3 AS SELECT * FROM v0;
UPDATE v0 SET v1 = 'x' WHERE
( v1 = -1 )
OR ( v1 = ( SELECT v1 AS v3
FROM v0_tmp2
WHERE v1 IN ( 3 , v1 NOT LIKE 'x' , 38 , 22 ) ) )
OR ( ( v1 , v1 ) IN ( SELECT v1 , v1
FROM v0_tmp3 ) )
OR ( v1 = 2147483647 ) ;
```

Figure 4: Suspected Bug in Order of Returned Elements in SELECT Statement

A detailed record of the bugs that appeared legitimate but were eventually determined not to be, can be found in the complete project submission (the file is listed in the README). Each entry follows this pattern: firstly, the query, followed by the return of each subquery as well as the internal state of the whole database at the end of the execution of all the subqueries, regardless of whether the DB crashed or not. This data extraction is replicated for the subsequent DB.

6 RESULTS

Overall, the methodology was highly successful. After 2 days of fuzzing, once the changes in the previous section were implemented, more than 500 bugs were detected. Of these, there were 9 'unique' bugs - essentially 9 categories of inputs that yield similar types of bugs. In this section, these bugs are discussed and the performance of our methodology is evaluated.

To prove the whole correctness of the system, two different database versions were picked, MariaDB version 11.0 (release candidate) and MariaDB version 10.10. These versions were picked because the latter has been extensively modified and parched as more bugs were discovered and the former includes a major revision of MariaDB. Hence, a higher chance of finding a regression bug.

After running our tool for several days, it encountered several bugs. Those bugs can be classified into different types:

- **Performance bugs:** This type of bug is hard to replicate as it is dependent on external factors such as memory or CPU usage. Some of these bugs are included in the appendix.
- **Regression logical bugs:** This is the type of bug that the tool aims to find. A regression bug refers to a bug that was not present in a previous version of the software but appears in a newer version. These bugs will be discussed in the next section.

Although this tool is able to identify performance regression bugs (given that both MariaDB instances are under the same load before and after the query and they both have access to the same amount of RAM), we decided not to focus on this aspect, as the goal of this project was to find logical regression bugs.

When investigating the 9 unique bugs detected, it was important to determine if they truly were regression bugs or not. This is challenging, because often times it is not clear what the correct output of the query is meant to be. As a result, for each potential bug, the query was converted into MySQL syntax and run on the newest version of MySQL. The idea behind this is that if the bug is a real regression, then the new version of MySQL should be consistent with the old version of MariaDB. It is unlikely that the same bug is introduced to MariaDB and MySQL. Following this procedure, there were two bugs that ended up being true regression bugs. These are discussed in the following section.

6.1 Unveiling Regression Bugs

In this section, we delve into the specifics of two regression bugs that our tool has effectively identified. These bugs serve as illustrative examples of the challenges inherent in SQL query handling and demonstrate the efficacy of our tool in detecting complex bugs.

6.1.1 Regression Bug 1. The first of these bugs is detailed below and depicted in Figure 5. It manifests in an intricate SQL query composed of five distinct SQL commands.

(1) **CREATE TABLE v0 (v1 INT):** This command initiates the process by creating a table titled "v0", which contains a singular integer column called "v1". This forms the groundwork for the subsequent commands.
(2) **INSERT INTO v0 (v1) VALUES (60):** The second command introduces a new row into the "v0" table, assigning the value of 60 to the "v1" field. At this point, the table contains one row with a single integer value.
(3) **UPDATE v0 SET v1 = 5 WHERE ... :** This command is more complex, updating the value of "v1" to 5 in rows of the "v0" table, but only if they meet a certain condition. This condition is encapsulated within a series of nested SELECT statements that include GROUP BY and HAVING clauses. Additionally, an IN operator is used with several conditions. The inherent complexity of this command, with its nesting and multiple conditions, potentially triggers the bug.
(4) **INSERT INTO v0 SELECT * FROM v0 WHERE 127 > v1:** This command copies all rows from "v0" where "v1" is less than 127, and inserts them back into the "v0" table. This effectively doubles the number of rows in the table for values of "v1" less than 127.
(5) **SELECT v1, v1 FROM v0 WHERE NOT ('x' = 'x' AND v1 = 99):** The final command in this sequence selects rows from "v0" under the condition that "v1" is not equal to 99, or that 'x' is not equivalent to 'x'. Given that 'x' always equals 'x', this effectively filters out rows where "v1" is equal to 99.

MariaDB 11.0 (release candidate) returned an error, while MariaDB 10.10 returned two rows and two columns with the integer value 60 in every position.

In order to validate that this anomaly indeed constitutes a bug, the query was executed on the most recent release of MySQL, leading to inconsistent outcomes. When run on MariaDB 11.0 (release candidate), the system returned an ERROR 1292 (22007) at line 5, indicating a 'Truncated incorrect DECIMAL value: 'x'. Contrasting, the result generated by MySQL, as depicted in the appendix, was congruent with the output from MariaDB 10.10. This discrepancy substantiates the presence of a regression bug between MariaDB 11.0 and MariaDB 10.10.

6.1.2 Regression Bug 2. The second of these bugs is explained below and illustrated in Figure 7.2. This bug also occurs in a complex SQL query, this time composed of seven separate SQL commands.

(1) **CREATE TABLE v0 (v1 INT, v2 INT, v3 INT, v4 INT):** This initial command creates a table "v0" with four integer columns, "v1", "v2", "v3", and "v4".
(2) **INSERT INTO v0 (v2) VALUES (27):** The second command inserts a row into the "v0" table with a "v2" value of 27.
(3) **CREATE TEMPORARY TABLE v0_tmp1...:** This command creates a temporary table "v0_tmp1" as a copy of the current state of table "v0".
(4) **UPDATE v0 SET v1 = 58 WHERE ...:** This complex command updates the "v1" value in the "v0" table to 58 for rows where certain conditions are met, related to the other columns in the table and their relation to values in the temporary table.
(5) **CREATE TEMPORARY TABLE v0_tmp2 AS SELECT * FROM v0:** Similar to the third command, this creates another temporary table "v0_tmp2" as a copy of the updated "v0" table.
(6) **INSERT INTO v0 SELECT v2 + 8 FROM v0_tmp2 AS v5, v0 AS v6 NATURAL JOIN v0_tmp2 NATURAL JOIN

```
-- First, we create a table v0 with a single
     integer column v1
CREATE TABLE v0 (
    v1 INT
);

-- Then we insert a row into v0 with a value
     of 60 in column v1
INSERT INTO v0 (v1)
VALUES (60);

-- Next, we update the values in column v1
    to 5 for rows in v0 that meet a certain
    condition
UPDATE v0
SET v1 = 5
WHERE (
    SELECT v1
    GROUP BY v1, v1, v1
    HAVING (v1 = 5 AND (v1 = v1 OR v1 = v1))
) IN (
    SELECT
        'x' IN (
            (
                v1 = 'x' AND v1 = 16,
                v1
            ) NOT IN (
                SELECT v1, v1
            ),
            77,
            NULL,
            74
        )
    FROM v0
    GROUP BY v1
);

-- We then duplicate rows in v0 where the
    value in column v1 is less than 127
INSERT INTO v0
SELECT * FROM v0
WHERE 127 > v1;

-- Finally, we select the rows from v0 that
    match a certain condition
SELECT v1, v1
FROM v0
WHERE NOT ('x' = 'x' AND v1 = 99);
```

Figure 5: The first identified SQL query leading to a regression bug

v0_tmp2: This command inserts new rows into "v0", selecting a value of "v2" plus 8 from the temporary table and joined records.

(7) **SELECT v3, v4 FROM v0 WHERE NOT ('x' = 'x' AND v2 = 0):** The final command selects rows from "v0" where 'x' is not equivalent to 'x' or "v2" is not equal to 0. Given that 'x' always equals 'x', this effectively filters out rows where "v2" is equal to 0.

In this one, MariaDB 11 (release candidate) returns an error "ERROR 1054 (42S22) at line 4: Unknown column 'v3' in 'order clause'". However, MariaDB 10.10 returns a different type of error, ERROR 1066 (42000) at line 6: Not unique table/alias: 'v0'

In order to validate that this anomaly indeed constitutes a bug, the query was executed on the most recent release of MySQL, leading to inconsistent outcomes. When run on MariaDB 11.0 (release candidate), the system returned an "ERROR 1054 (42S22) at line 4: Unknown column 'v3' in 'order clause'". Contrasting, the result generated by MySQL, as depicted in the appendix, was congruent with the output from MariaDB 10.10. This discrepancy substantiates the presence of a regression bug between MariaDB 11.0 and MariaDB 10.10.

6.2 Efficacy of the Methodology

Given the speed at which new bugs were found, this appears to be an effective approach to detect regression logical bugs. However, it is difficult to determine if the differential coverage oracle played a role in this success.

As a result, a further test was run. Fuzzing was also run for 2 days without the differential coverage - simply with normal full coverage. In this time, no bugs were detected. This is not a conclusive result, as fuzzing is a highly randomised process, and it is possible that between runs, even the same methodology will result in different levels of success. Overall, however, it is believed that the differential coverage oracle makes a significant improvement in the speed of bug detection.

Further investigation can be conducted in the future into how the coverage oracle can be manipulated to better find bugs.

REFERENCES

[1] Mariadb, 2009.
[2] Anirban Basu. Software quality assurance, testing and metrics.
[3] Z. Chong et al. Detecting database bugs via symbolic execution. *arXiv preprint arXiv:2001.04174*, 2023.
[4] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded scheduling. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 411–422. ACM, 2015.
[5] Z. Jinsheng et al. Guiding test case generation for database systems with query plan coverage. In *International Conference on Software Engineering*, 2023.
[6] Vijayakumar L et al. Amoeba: Effective discovery of communication patterns in collaborative applications. In *International Symposium on Software Testing and Analysis*, 2021.
[7] X Liang et al. Detecting logical errors in sql queries for large-scale databases. In *International Conference on Software Engineering*, 2022.
[8] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816. ACM, 2007.
[9] Manuel Rigger, Zhendong Le, Daniel Moldovan, Alexander J Summers, Peter M O'Hara, Shuai Chen, Thomas Mytkowicz, and Kathryn S McKinley. Testing database systems with pivoted query synthesis. In *Proceedings of the VLDB Endowment*, volume 13, pages 2722–2735. VLDB Endowment, 2020.
[10] Donald R Slutz. Massive stochastic testing of sql. In *Proceedings of the 24th VLDB Conference*, 1998.

```
-- First, we create a table v0 with four
    integer columns v1, v2, v3, and v4
CREATE TABLE v0 (
    v1 INT,
    v2 INT,
    v3 INT,
    v4 INT
);

-- Then we insert a row into v0 with a value
     of 27 in column v2
INSERT INTO v0 (v2)
VALUES (27);

-- We create a temporary table v0_tmp1 which
     is a copy of table v0
CREATE TEMPORARY TABLE v0_tmp1 AS
SELECT * FROM v0;

-- We update column v1 to 58 in the rows of
    v0 that match a certain condition
UPDATE v0
SET v1 = 58
WHERE (
    SELECT v4
    UNION SELECT v2
    ORDER BY v3
) IN (
    SELECT v1 FROM v0_tmp1
    WHERE
        16 > v4
        OR (v2 > 127 AND v3 < 35)
        OR (v1 BETWEEN 16 AND 10)
        OR (v1 <= 0)
);

-- We create another temporary table v0_tmp2
     which is a copy of the updated v0
CREATE TEMPORARY TABLE v0_tmp2 AS
SELECT * FROM v0;

-- We insert new rows into v0, with values
    computed from v0_tmp2 and other rows of
    v0
INSERT INTO v0
SELECT v2 + 8
FROM v0_tmp2 AS v5, v0 AS v6
NATURAL JOIN v0_tmp2
NATURAL JOIN v0_tmp2;

-- Finally, we select the rows from v0 that
    match a certain condition
SELECT v3, v4
FROM v0
WHERE NOT ('x' = 'x' AND v2 = 0);
```

Figure 6: The second identified SQL query leading to a regression bug

[11] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[12] Xingjun Zhong, Shuai Chen, Zhendong Su, Zhi Chen, and Dan Zhang. Squirrel: Testing database management systems with language validity and coverage feedback. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 689–703, 2020.

7 APPENDIX

7.1 Modified Dockerfile for running Squirrel with two DBMS

```
FROM mcr.microsoft.com/devcontainers/base:jammy
LABEL maintainer="Squirrel"

# common config
RUN apt-get update && apt-get -y install make cmake build
    -essential vim sudo git \
    clang libmysqlclient-dev ninja-build pkg-config clang
        -format \
    libpq-dev libyaml-cpp-dev lld python3-fire

RUN mkdir -p /home && \
    groupadd dobigthing && \
    useradd -l -K UMASK=0000 -d /home -g dobigthing
        dobigthing && \
    chown dobigthing:dobigthing /home

RUN     echo "dobigthing:dobigthing" | chpasswd &&
    usermod -a -G sudo dobigthing
RUN chmod +w /etc/sudoers && \
    echo "%dobigthing   ALL=(ALL:ALL)NOPASSWD:ALL" >> /
        etc/sudoers && \
    chmod -w /etc/sudoers

# install mariadb-server
RUN apt-get -y install apt-transport-https curl software-
    properties-common gnutls-dev
RUN curl -o /etc/apt/trusted.gpg.d/
    mariadb_release_signing_key.asc \
    'https://mariadb.org/mariadb_release_signing_key.asc'
RUN echo "deb https://ftp.osuosl.org/pub/mariadb/repo
    /10.10/ubuntu/ jammy main" >> /etc/apt/sources.list
RUN echo "deb-src https://ftp.osuosl.org/pub/mariadb/repo
    /10.10/ubuntu jammy main" >> /etc/apt/sources.list
RUN apt-get update && apt-get -y build-dep mariadb

USER dobigthing
WORKDIR /home

RUN git clone https://github.com/Dobigthing666/Squirrel.
    git && \
    cd Squirrel && git submodule update --init && \
    cmake -S . -B build -DCMAKE_BUILD_TYPE=Release && \
    cmake --build build -j && \
    cd AFLplusplus/ && LLVM_CONFIG=llvm-config-14 make -
        j20

RUN git clone https://github.com/MariaDB/server.git
    mariadb && cd mariadb && git checkout 10.10
RUN mkdir bld2 && cd bld2 && \
    CC=/home/Squirrel/AFLplusplus/afl-clang-fast CXX=/
        home/Squirrel/AFLplusplus/afl-clang-fast++ cmake
        ../mariadb/ && \
    make -j20 && sudo cmake --install . --prefix /usr/
        local/mysql/

USER root
RUN sudo rm /etc/apt/sources.list && sudo touch /etc/apt/
    sources.list
RUN curl -o /etc/apt/trusted.gpg.d/
    mariadb_release_signing_key.asc \
    'https://mariadb.org/mariadb_release_signing_key.asc'
RUN echo "deb https://ftp.osuosl.org/pub/mariadb/repo
    /11.0/ubuntu jammy main" >> /etc/apt/sources.list
RUN echo "deb-src https://ftp.osuosl.org/pub/mariadb/repo
    /11.0/ubuntu jammy main" >> /etc/apt/sources.list
RUN apt-get update && apt-get -y build-dep mariadb

USER dobigthing
RUN cd mariadb && git checkout 11.0 && cd .. && \
    mkdir bld && cd bld/ && \
    CC=/home/Squirrel/AFLplusplus/afl-clang-fast CXX=/
        home/Squirrel/AFLplusplus/afl-clang-fast++ cmake
        ../mariadb/ && \
    make -j20 && sudo cmake --install . --prefix /usr/
        local/mysql2/

RUN sudo chown dobigthing:dobigthing /usr/local/mysql/ -R
     && \
    cd /usr/local/mysql/ && \
    scripts/mysql_install_db --user=dobigthing

RUN sudo chown dobigthing:dobigthing /usr/local/mysql2/ -
    R && \
    cd /usr/local/mysql2/ && \
    scripts/mysql_install_db --user=dobigthing

# Get map size and save it to /tmp/mapsize
RUN AFL_I_DONT_CARE_ABOUT_MISSING_CRASHES=1
RUN AFL_DEBUG=1 __AFL_SHM_ID=1234 /usr/local/mysql/bin/
    mariadbd 2>&1 | tail -n 1 | cut -d"," -f8 | cut -d"
    " -f 3 > /tmp/mapsize
WORKDIR /home/Squirrel/data
ADD ./data ./data
WORKDIR /home/Squirrel/scripts/utils
ENTRYPOINT AFL_MAP_SIZE=$(cat /tmp/mapsize) python3 run.
    py mariadb ../../data/fuzz_root/mysql_input/
```

7.2 Second query to MySQL to check if the discrepancy was a bug

```
-- First, we create a table v0 with four
    integer columns v1, v2, v3, and v4
CREATE TABLE v0 (
    v1 INT,
    v2 INT,
    v3 INT,
    v4 INT
);

-- Then we insert a row into v0 with a value
     of 27 in column v2
INSERT INTO v0 (v2)
VALUES (27);

-- We create a temporary table v0_tmp1 which
     is a copy of table v0
CREATE TEMPORARY TABLE v0_tmp1 AS
SELECT * FROM v0;

-- We update column v1 to 58 in the rows of
    v0 that match a certain condition
UPDATE v0
SET v1 = 58
WHERE (
    SELECT v4
    UNION SELECT v2
    ORDER BY v3
) IN (
    SELECT v1 FROM v0_tmp1
    WHERE
        16 > v4
        OR (v2 > 127 AND v3 < 35)
        OR (v1 BETWEEN 16 AND 10)
        OR (v1 <= 0)
);

-- We create another temporary table v0_tmp2
     which is a copy of the updated v0
CREATE TEMPORARY TABLE v0_tmp2 AS
SELECT * FROM v0;

-- We insert new rows into v0, with values
    computed from v0_tmp2 and other rows of
    v0
INSERT INTO v0
SELECT v2 + 8
FROM v0_tmp2 AS v5, v0 AS v6
NATURAL JOIN v0_tmp2
NATURAL JOIN v0_tmp2;

-- Finally, we select the rows from v0 that
    match a certain condition
SELECT v3, v4
FROM v0
WHERE NOT ('x' = 'x' AND v2 = 0);
```

7.3 First query to MySQL to check if the discrepancy was a bug

```
-- First, we create a table v0 with a single
     integer column v1
CREATE TABLE v0 (
    v1 INT
);

-- Then we insert a row into v0 with a value
     of 60 in column v1
INSERT INTO v0 (v1)
VALUES (60);

CREATE TEMPORARY TABLE v0_tmp1 AS

-- Next, we update the values in column v1
    to 5 for rows in v0 that meet a certain
    condition
UPDATE v0
SET v1 = 5
WHERE (
    SELECT v1
    GROUP BY v1, v1, v1
    HAVING (v1 = 5 AND (v1 = v1 OR v1 = v1))
) IN (
    SELECT
        'x' IN (
            (
                v1 = 'x' AND v1 = 16,
                v1
            ) NOT IN (
                SELECT v1, v1
                ),
                77,
                NULL,
                74
            )
    FROM v0_tmp1
    GROUP BY v1
);

CREATE TEMPORARY TABLE v0_tmp2 AS

-- We then duplicate rows in v0 where the
    value in column v1 is less than 127
INSERT INTO v0
SELECT * FROM v0_tmp2
WHERE 127 > v1;

-- Finally, we select the rows from v0 that
    match a certain condition
SELECT v1, v1
FROM v0
WHERE NOT ('x' = 'x' AND v1 = 99);
```

7.3.1 *Authors:* Alejandro Cuadron and Nicholas Thumiger,,