Unicorn: Detect Runtime Errors in Time-Series Databases With Hybrid Input Synthesis

Zhiyong Wu
KLISS, BNRist, School of Software,
Tsinghua University, China
wuzy21@mails.tsinghua.edu.cn

Jie Liang*
KLISS, BNRist, School of Software,
Tsinghua University, China
liangjie.mailbox.cn@gmail.com

Mingzhe Wang
KLISS, BNRist, School of Software,
Tsinghua University, China
wmzhere@gmail.com

Chijin Zhou
KLISS, BNRist, School of Software,
Tsinghua University, China
ShuimuYulin Co., Ltd, China
tlock.chijin@gmail.com

Yu Jiang*
KLISS, BNRist, School of Software,
Tsinghua University, China
jiangyu198964@126.com

ABSTRACT

The ubiquitous use of time-series databases in the safety-critical Internet of Things domain demands strict security and correctness. One successful approach in database bug detection is fuzzing, where hundreds of bugs have been detected automatically in relational databases. However, it cannot be easily applied to time-series databases: the bulk of time-series logic is unreachable because of mismatched query specifications, and serious bugs are undetectable because of implicitly handled exceptions.

In this paper, we propose Unicorn to secure time-series databases with automated fuzzing. First, we design hybrid input synthesis to generate high-quality queries which not only cover time-series features but also ensure grammar correctness. Then, Unicorn uses proactive exception detection to discover minuscule-symptom bugs which hide behind implicit exception handling. With the specialized design oriented to time-series databases, Unicorn outperforms the state-of-the-art database fuzzers in terms of coverage and bugs. Specifically, Unicorn outperforms SQLsmith and SQLancer on widely used time-series databases IoTDB, KairosDB, TimescaleDB, TDEngine, QuestDB, and GridDB in the number of basic blocks by 21%-199% and 34%-693%, respectively. More importantly, Unicorn has discovered 42 previously unknown bugs.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;
• **Security and privacy → Database and storage security**.

KEYWORDS

Time-series Databases, Runtime Error, Hybrid Input Synthesis

*Jie Liang and Yu Jiang are the corresponding authors.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534364

ACM Reference Format:
Zhiyong Wu, Jie Liang, Mingzhe Wang, Chijin Zhou, and Yu Jiang. 2022. Unicorn: Detect Runtime Errors in Time-Series Databases With Hybrid Input Synthesis. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534364

1 INTRODUCTION

Along with the rapid growth in Internet of Things (IoT) deployment, time-series databases are ubiquitously used in all kinds of IoT devices. Compared to traditional relational databases, time-series databases employ complex logic to handle their low latency and time-series nature. Therefore, its security, reliability, and correctness are challenged by the complexity. To prevent vulnerabilities, a common approach is writing unit tests for the target database manually. However, unit testing is labor-consuming and cannot detect bugs at system level.

One promising approach is fuzzing, an automated software testing technique, which generates random data as program inputs. It was first developed by Miller et al. [21] in 1990s and has, since then, been widely adopted in practice for finding bugs in many critical areas, including operating systems [14, 34, 37], networking protocols [7, 19, 20, 40], third-part libraries [1, 15–17, 39]. A fuzzer exercises the target program in a loop: (1) select an input and generate candidate inputs based on it, (2) execute candidate inputs to track coverage and monitors anomalies, (3) save interesting candidate inputs which have new coverage, then go to (1). Following the fuzzing loop, fuzzers could continuously explore more and more state space of the target program.

Due to the easily adapted nature, fuzzing can continuously test whole systems with little manual effort. Prior works have successfully applied fuzzing to relational databases and discovered many vulnerabilities. For example, SQLsmith [31] constructs inputs with the abstract syntax tree (AST) model automatically and sends them to target systems for execution. It has found more than 100 bugs in PostgreSQL, SQLite, and MonetDB since 2015 [32]. However, because of the unique attributes of time-series databases, existing fuzzing strategies are hard to directly adapt to these databases. There are two major challenges as follows.

The first challenge is generating grammatically-correct time-series queries. Time-series is the basic form to organize data for time-series

databases, but existing fuzzers are hard to generate grammatically-correct time-series queries to test. Specifically, time-series data [8, 9] represents a collection of data values observed from sequential measurements over time. To improve the efficiency to store and fetch data, time-series databases employ different strategies from relational databases to fit the time-series storage. However, lacking time-series specifications of time-series databases, existing fuzzing strategies are hard to generate grammatically-correct time-series queries. Specifically, due to the vast difference between *time-series* in IoT domain and relations in SQL, traditional relational database fuzzers (e.g. SQLsmith [31]) can hardly reach time-series logic. In addition, the queries accepted by time-series databases are *highly-structured*, the strict grammar impedes most of the seeds generated by random mutation in conventional mutation-based fuzzers (e.g. AFL [15]). As a result, designing a time-series input generation mechanism, which generates grammatically correct time-series queries, to explore the time-series logic is needed.

The second challenge is capturing exceptions handled implicitly. Crashes are used as an indication for failed tests in fuzzing, however, time-series databases utilize *implicit exception handling* to prevent crashing whole systems for usability and reliability. In other words, when anomalies do not happen in critical locations of the server, they are handled implicitly and no crashes could be triggered. For example, time-series databases usually create a new thread for each connecting client as the worker. When an exception is thrown inside the thread, the implicit handling mechanism will automatically capture it and only inform the worker with a fault message. Therefore, the server could still preserve a normal running state. However, these exceptions may contain serious bugs and they will be ignored by existing fuzzing approaches. As a result, designing an implicitly handled exception detection scheme, which directly obtains exception messages to determine whether it is an anomaly, to capture all possible bugs is required.

In this paper, we propose Unicorn to overcome the challenges through hybrid input synthesis and proactive exception detection. In order to generate grammatically-correct time-series queries, *hybrid input synthesis* combines the syntax-preserved mutation and time-series guided mutation. Specifically, we design hybrid input specification, which combines the rules to generate conventional SQLs and time-series SQLs in time-series databases. Based on the specification, Unicorn first constructs the abstract syntax tree (AST) for the original seeds and generates new time-series queries by changing the time-series nodes of AST. To detect exceptions handled implicitly, *proactive exception detection* directly captures exception information from the runtime environment and analyzes whether it is an anomaly. Specifically, instead of passively receiving the program's state, Unicorn inserts an agent into each process to proactively catch the exceptions and send them to the anomaly detector for analyzing and reporting.

For evaluation, we used Unicorn to perform fuzzing on IoTDB, KairosDB, QuestDB, TimescaleDB, TDEngine, and GridDB. We also adapted the industrial fuzzers SQLancer [25] and SQLsmith [31] for comparison. Unicorn covered 115.75% more basic blocks on average than the best results of other fuzzers. In addition, Unicorn detected 42 previously unknown bugs.

In conclusion, our paper makes the following contributions:

- We observe that current fuzzing approaches are hard to effectively test time-series databases. The two main challenges are generating grammatically correct queries and capturing exceptions handled implicitly.
- We propose hybrid input synthesis and proactive exception detection to address the aforementioned challenges. We also implement these approaches in Unicorn.
- We evaluate Unicorn on 6 popular time-series databases against state-of-the-art fuzzers SQLsmith and SQLancer. The results show that Unicorn outperforms others and 42 previously-unknown bugs are detected.

2 TIME-SERIES DATABASES

As an infrastructure for IoT data storage and analysis, time-series databases play an important role in promoting the development of Internet of Things. Generally, the time-series database is a kind of large-scale software to manipulate and manage IoT data, it handles the operation requests from various clients (including IoT devices, PC, etc.), and carries out unified management and control to ensure the security and integrity of IoT data [18]. In embedded application scenarios, time-series databases usually have the following two characteristics: 1) They employ time-series data to meet scenarios in the IoT domain, and 2) They utilize implicit exception handling to guarantee usability, namely, they limit the impacts of anomalies by handling them internally to ensure the server always runs normally.

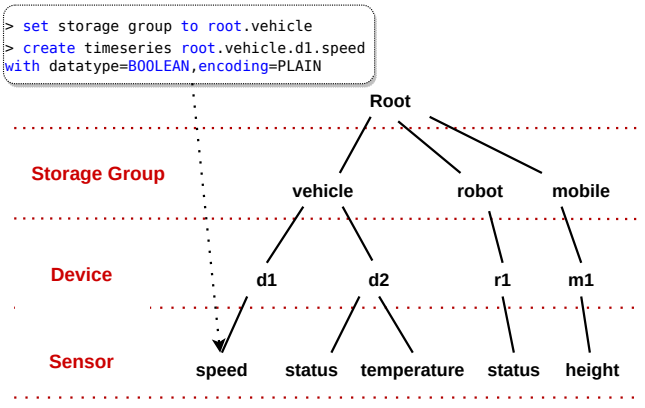

Figure 1: The time-series query along with the corresponding storage model of Apache IoTDB. The query imports new keywords related to time-series. In addition, the object has hierarchical structures because of the tree-based schema in IoTDB. IoT data is stored in a tree-based schema, and the attribute hierarchy structure has three layers. A grammatically correct object name should construct a path from the root node to a leaf node.

2.1 Employing Time-Series Data

Compared to other applications, the major characteristic of the IoT applications is *employing time-series data*. Time series data [8, 9]

represents a collection of data values observed from sequential measurements over time. Time series data are always large in data size and needed to update continuously. Due to the numerical and continuous characteristics, *time-series databases consider time-series data as a whole and use different mechanisms to deal with them.*

To efficiently store and fetch the data based on time-series data storage, time-series databases import different grammars for queries. For simplicity, the tokens in an SQL query could be divided into two groups, namely *structure* that defines the operations to perform (e.g. SQL keywords) and *data* that specify the targets of defined operations (e.g. target objects). Time-series queries differ from queries in relational databases both in structures and data.

For structure, time-series databases import new keywords and objects. We take the query in Figure 1 as an example to show the differences. The query imports two new keywords, namely `timeseries` and `storage group`. Furthermore, the object named "`root.vehicle.d1.speed`" has hierarchical structures, which are different from object names in conventional relational databases. The hierarchical structures of collecting data in IoT devices cause the differences. The figure shows the tree-based schema of IoTDB. The data are collected from three layers, and from top to bottom is the storage group layer - device layer - sensor layer. It has one root node (i.e. `ROOT`), and several leaf nodes in the sensor layer. A grammatically correct object name should construct a path from the root node to a leaf node.

For data, time-series databases attach a timestamp for each record. The data is a sequence of record values by collecting chronologically from sensors on devices. Time is one of the most important attributes, therefore, each element in the sequence is a tuple that contains the time and value attribute, which looks like `<timestamps, values>`. In addition, queries in time-series databases have strict grammar rules. Any incorrect query will be directly refused by the parser of databases.

2.2 Handling Exceptions Implicitly

Another critical characteristic of time-series databases is handling exceptions implicitly. IoT devices are widely used in many critical areas, therefore, usability and reliability are indispensable for time-series databases. However, in embedded scenarios, time-series databases are designed with complex distribution structures. Specifically, they have to handle the ingestion of tens of millions of IoT data points per second stably from hundreds of IoT device clients. Thus time-series databases generally contain hundreds of components, which have complex interactions with others during runtime. Any exception in these interactions should not cause the server to go down. As a result, *time-series databases generally use implicit exception handling*, which keeps the server running normally even if some harmless exceptions are thrown. For example, Figure 2 illustrates the implicit exception handling in Apache IoTDB [35]. To support different functionalities, IoTDB registers amounts of threads. Among them, Time-Series Storage thread, Time-Oriented Query thread, and IoT Data Management thread are three main threads in three main modules. If an exception happens in one of those threads, the *Implicit Exception Handling Mechanism* will handle the exception to keep the server running normally.

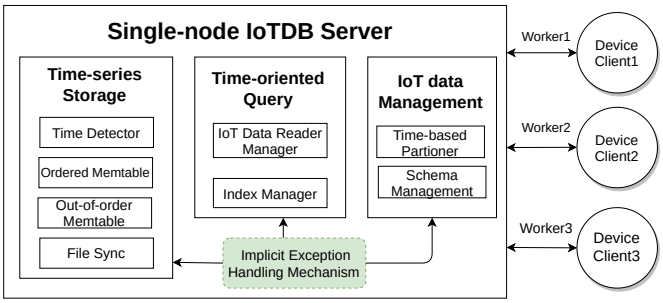

Figure 2: Implicit exception handling mechanism in a famous time-series database Apache IoTDB. In the single-node running model of IoTDB, a server is connected by lots of device clients. The server generally has three core modules, namely Time-Series Storage, Time-Oriented Query, and IoT Data Management. The interactions of the three components are frequent and complex. When an exception (e.g loss of data) happens in any component, the IoTDB will implicitly handle it to ensure the whole system running normally.

3 CHALLENGES IN FUZZING TIME-SERIES DATABASES

Fuzzing has achieved remarkable results in the field of software testing [4, 7, 15, 25, 31]. However, due to the two major characteristics in time-series databases, there are two major challenges for fuzzing time-series databases, namely *generating grammatically-correct time-series queries* and *capturing exceptions handled implicitly.*

To demonstrate the challenges of performing fuzzing in time-series databases, we present a bug causing damage to the *database integrity* of Apache IoTDB. As one of the most important features of databases, *database integrity* requires that the data stored in the database is logically consistent, correct, and valid. It reflects that

```
create timeseries root.st0.device0.senor0 with
      datatype=INT64,encoding=REGULAR ;
--Step 1: set the TTL for storage group
set ttl to root.st0 1000 ;
--Step 2: Insert records
insert into root.st0.device0(time,senor0) values
      (2021-02-25T18:01:01.000+08:00,1200);
insert into root.st0.device0(time,senor0) values
      (2021-02-25T18:01:02.000+08:00,1200);
-- Step 3: flush temp content and unset the TTL
flush ;
unset ttl to root.st0;
-- Step 4: delete the data and reinsert records
delete from root.st0.device0;
insert into root.st0.device0(time,senor0) values
      (2021-02-25T17:36:01.000+08:00,1200);
insert into root.st0.device0(time,senor0) values
      (2021-02-25T17:36:02.000+08:00,1100);
insert into root.st0.device0(time,senor0) values
      (2021-02-25T17:36:03.000+08:00,1000);
...
--Step 5: flush temp content and query the data in
      root.st0.device0
flush;
select * from root.st0.device0;
```

Listing 1: The simplified time-series query that triggers the IndexOutofBoundsException.

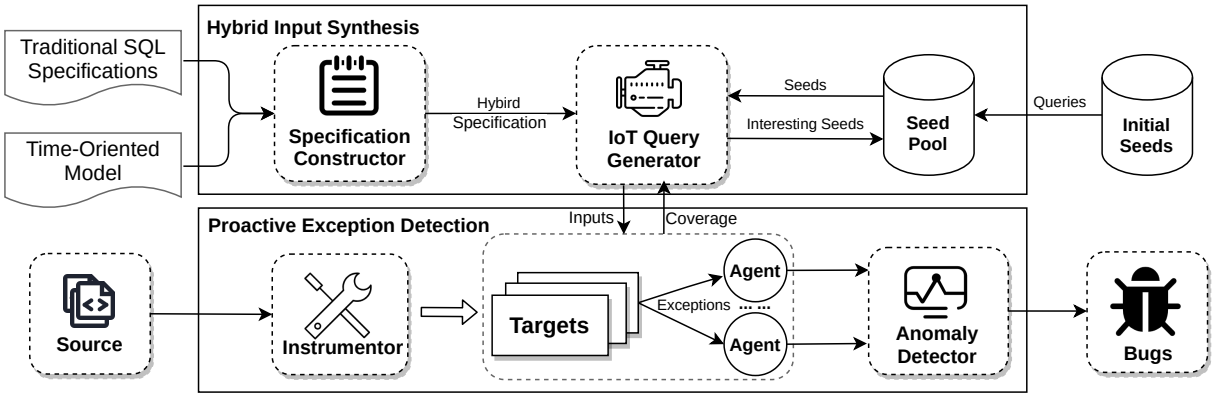

Figure 3: Overview of Unicorn. It contains two parts, namely *hybrid input synthesis* and *proactive exception detection*. Hybrid input synthesis first constructs the hybrid specification with the collected traditional SQL specifications and time-oriented models. Then, the IoT Query Generator generates time-series queries by mutating seeds based on hybrid specification. The queries which hit new behaviors will be preserved to the seed pool for further mutation. Proactive exception detection uses agents to collect exceptions. The agents are managed by the anomaly detector and they will send exception messages to the detector. Finally, the anomaly detector deduplicates and reports bugs along with their triggered input and call stack traces.

the server of a database can always return correct results for each query from a client. The bug violates the integrity, which causes the loss of data. Consequently, the IoT devices that rely on these data perform abnormally and a large amount of real valuable data stored in IoT devices is lost in the production environment.

Listing 1 shows the simplified seed to trigger this bug. The Apache IoTDB performs the actions as follows with the seed:

(1) Set the TTL (i.e. Time To Live) for the existing time-series storage group. The database will delete old data which exceeds TTL automatically and periodically.
(2) Insert some out-of-date records, which will be automatically deleted because of the TTL.
(3) Flush the temporary data, then unset the TTL from the storage group when overrun the TTL.
(4) Delete the time-series data in the storage group and then insert some valid time-series records.
(5) Flush the temporary data again and then execute select statements to query the data in the storage group.

According to the database integrity, the server should have returned some time-series records inserted in Step 4 to the client after executing the statement "select * from root.st0.device0". However, the client just received an error message "Msg:500, bitIndex < 0 :-2147483648", because of the time-series data loss in IoTDB.

The bug can hardly be detected by existing fuzzers. There are two major challenges for detect this anomaly. First, this bug was triggered with minuscule symptoms. Test oracles are prerequisites for testing, which determines whether the test target runs normally under a test case. A basic oracle for fuzzing is detecting whether an input runs normally without triggering any crash of the server, e.g.SQLsmith [31]. However, this bug about the database integrity does not crash the server directly, and even the client continues to run normally after receiving the error message. It would be missed by the detector of existing fuzzers.

In addition, this bug could only be triggered when IoTDB is in a specific state within the context of schema change, which contains some time-series operations. The query in Listing 1 has time-series characteristics, which is different from conventional queries in relational databases both in structure and data. However, lacking time-series specifications, existing fuzzing strategies are hard to generate time-series queries. As a result, the logic to deal with the main attribute – time-series, is hard to be tested in time-series databases by existing fuzzing strategies. On the other hand, the queries accepted by time-series databases are *highly-structured*, which increases the difficulty for fuzzers to generate grammatically correct queries. For example, conventional mutation-based fuzzers (e.g. AFL [15]) utilize random mutation to generate new seeds, which could easily wreck the delicate structures in SQL queries. Consequently, the inputs which triggered the bug are difficult to be generated by existing fuzzers.

4 SYSTEM DESIGN

Figure 3 presents the overview of Unicorn has two parts, namely *hybrid input synthesis* and *proactive exception detection*. *Hybrid input synthesis* addresses the challenge of generating grammatically correct time-series queries, while *proactive exception detection* addresses the challenge of detecting exceptions during the running of time-series databases.

4.1 Hybrid Input Synthesis

To perform fuzzing on time-series databases, grammatically correct time-series queries need to be generated automatically. However, existing fuzzers can hardly generate them due to the highly structured time-series input specification. Specifically, conventional mutation-based fuzzers (e.g. AFL) are hard to guarantee the grammatical correctness for highly structured specifications while existing generation-based fuzzers (e.g. SQLsmith) are hard to adapt to time-series features in the IoT domain. To address the problem,

we propose hybrid input synthesis. It designs the *hybrid input specification* to describe time-series features and utilizes *time-series mutation* to ensure grammatical correctness.

4.1.1 Hybrid Input Specification Construction. Hybrid Input Specifications describes the rules to construct a highly structured time-series query. It consists of traditional SQL specifications and time-oriented models. Traditional SQL specifications describe the grammar rules of relational queries, which can be obtained straightforwardly from the Internet. Time-oriented models describe time-series keywords and syntax rules. Traditional SQL specifications contain many basic descriptions of the SQL queries, which supplies the skeleton for queries. On the other hand, time-oriented models abstract time-series features, which supply the rules to generate time-series elements. Combining traditional SQL specifications and time-oriented models, hybrid input specifications could generate suitable time-series queries. Figure 4 shows an upper-level abstraction example of the hybrid input specification. The time-series query statements of time-series databases can be divided into two parts, namely general skeleton, which represents *traditional SQL specifications* (covered with red), and time-series elements, which reflect *time-oriented model* (covered with green).

A time-series query based on hybrid input specification is constructed as follows: first, UNICORN takes the traditional SQL input specification as the general skeleton; second, it fills the skeleton with time-series elements according to the time-oriented model. For example, in Figure 4, the `CREATE` statements of TimescaleDB can be constructed by adding a time-series element to the general skeleton of PostgreSQL as an attribute.

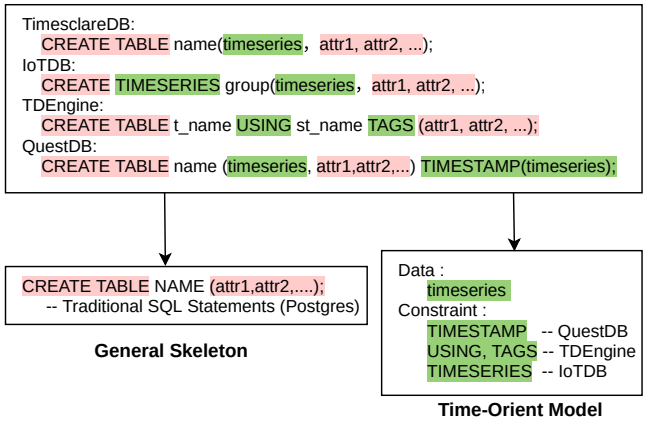

Figure 4: An example of hybrid input specification for time-series databases. The `CREATE` statements for QuestDB, IoTDB, and TDEngine can be generated from the general skeleton and the time-oriented model. The general skeleton is from PostgreSQL's specification. The time-oriented model is defined with the features of time-series databases.

4.1.2 Time-Series Mutation. With the hybrid input specification, UNICORN mutates queries by changing the structure and data which have time-series features. Algorithm 1 illustrates the process of time-series mutation. UNICORN first translates the original seed into the abstract syntax tree (AST) for better understanding and analysis. Compared to the source raw query, AST provides a more structured and precise format for analyzing the query's construction and logic. Then UNICORN identifies the time-series data nodes and time-series structure of the AST based on the specification. Then UNICORN mutates queries by randomly choosing from following two mutation methods: ① Modify time-series data. The mutation method mainly modifies the data in an AST, such as groups, devices, sensors. It also changes time-series value to generate new queries. ② Modify time-series structure. It changes the structure of the AST by removing, modifying, or adding nodes on the basis of the hybrid input specifications.

Algorithm 1: Time-Series Mutation.

```
Input  : Q: Time-series query,
         P: Grammar parser
Output : S: New time-series query
Function TimeSeriesParseMutate(parser,query):
    Tree = ConstructAST(Q, P) ;
    foreach node ∈ Tree.nodes do
        DNodes, SNodes = classifyNodes(node, Tokens) ;
    end
    while ifcontinueMutate() do
        Method = mutateStructureOrData() ;
        if Method = Struct then
            S = SNodeToMuate(SNodes) ;
        else
            S = DNodeToMuate(DNodes) ;
        end
    end
End Function
```

Take Figure 5 as an example, the query "select COUNT(*) from root.vehicle group by ([2,50],20ms)" is firstly translated into an AST. Secondly, UNICORN identifies the time-series data nodes and time-series structure nodes. Specifically, `prefixPath`, `LRBRACKET`, and `timeInterval` nodes belong to time-series data nodes, and the `selectElements`, `fromClause`, and `specialClause` belong to time-series structure nodes. Then, assume UNICORN chooses mutating new query by modifying the structure. It removes the specialClause node and all its successor nodes (removed nodes are covered by red) to generate a new AST. Finally, the new AST is translated into a new time-series query "select COUNT(*) from root.vehicle".

4.2 Proactive Exception Detection

As mentioned in Section 2, time-series databases use implicit exception handling to guarantee robustness and fault tolerance, which makes it difficult to detect internal anomalies with traditional crash oracles. To overcome this challenge, we design *proactive exception detection* to detect internal anomalies. It proactively captures the error messages from the runtime environment by inserting *agents* into each process and analyzing the exceptions reported from each agent with *anomaly detector*.

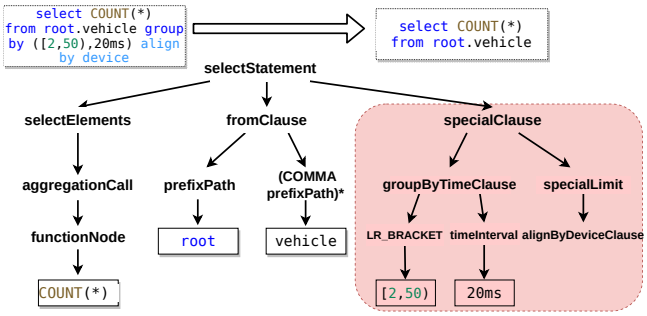

Figure 5: An example of time-series mutation for time-series databases. Unicorn chooses mutating structures and deletes `specialClause` node and all its successor nodes.

Figure 6 shows the overview of proactive exception detection. Generally, programs are dependent on the runtime environment, which might be a Virtual Machine (VM) or an Operation System (OS). All runtime errors of the programs will be transformed and handled in the runtime environment. Traditional fuzzers detect anomalies by monitoring whether the program crashes while it is running, which will ignore the exception with minuscule symptoms that are handled implicitly in time-series databases. In our design, Unicorn uses *agents* to directly collect all exception information generated by the program from the runtime environment, including those handled implicitly by time-series databases. In addition, Unicorn also contains an *anomaly detector*, which is used to manage all agents from each process, as well as record and analyze the detailed information of exceptions.

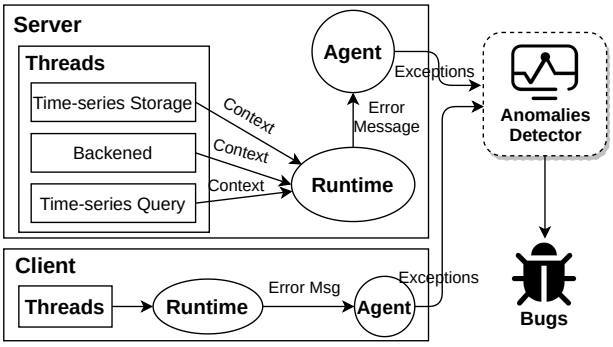

Figure 6: The design of proactive exception detection. Unicorn uses agents to capture exceptions both in server and clients. All agents are managed by an anomaly detector, which is also used to report the detected bugs. To collect the exception message, the server and clients are run in a common runtime environment. For example, the backend thread, time-series query thread, and the storage thread all share the same runtime environment. When exceptions happen, the agent will receive the message from the environment and report them to the anomaly detector.

4.2.1 Agent. As Figure 2 shows, time-series databases register a large number of threads to support different functions when the server is running, such as cache monitoring threads and temporary data dispatching threads. To prevent crashing the whole system, time-series databases usually handle all exceptions in each thread with the implicit exception handling mechanism. However, those implicitly handled exceptions may reflect bugs in workers, which may cause memory consumption or unexpected results.

Agent is designed to capture error messages from the runtime environment during the target database running. One agent is used to follow one process of time-series databases by instrumenting on exception handling code. To capture the exception message from the runtime environment, Unicorn firstly locates the exception handling code and gets all custom exceptions in time-series databases, which are used to distinguish the unchecked exceptions. Then, Unicorn scans each exception handler, and instruments those that don't throw the captured exception but implicitly handles it, to capture the error messages. For example, Listing 2 details the instrumentation code in an exception handler of Apache IoTDB. It first checks the exception's type and the logger. If the exception belongs to unchecked exceptions or it is a serious assertion, the detailed information of the exception (e.g. stack trace) will be recorded and sent to *agents* after the execution of the seed. Based on the instrumentation, *agents* can directly get the exception messages from the runtime environment.

```
public class MergeTask implements Callable<Void> {
    ...
  @Override
  public Void call() throws Exception {
    try {
      doMerge();
    } catch (Exception e) {
      logger.error("Runtime exception {}",
            taskName, e);
      if (e instance of runtime error) ;
      add e into errorFeedback of Mem.class to send ;
      record e.getStackTrace() and e.getCause() ;
      abort();
    }
    return null;
  }
  ...
}
```

Listing 2: The automatically instrumented code in an exception handler of Apache IoTDB.

4.2.2 Anomaly Detector. Agents in the different runtime environments are managed by *anomaly detector*. It also records and reports anomalies. Commonly, time-series databases start with different processes. And servers and clients may have different processes which run in various runtime environments. For example, KairosDB generally starts three processes in three individual virtual machines, namely the client process, the background storage process (i.e. Cassandra Server), and the main server process(i.e. KairosDB). Each agent in three runtime environments may report error messages. Consequently, Unicorn employs *anomaly detector* to collect exception messages from agents in multiple runtime environments. Specifically, Unicorn first configures unique agents for each virtual machine. Then, all the exceptions along with

their detailed error messages captured by agents from runtime environments are sent to the anomaly detector.

Next, the anomaly detector analyzes whether the exceptions are bugs based on their types and error messages. First, exceptions could be divided into unchecked exceptions and checked exceptions, and unchecked exceptions are determined as bugs by Unicorn. Specifically, an unchecked exception is an exception that occurs at the time of execution. *Unchecked exceptions could always be regarded as anomalies*, because the target program cannot reasonably be expected to recover from them or to handle them in any way [5, 22]. For example, typical unchecked exceptions include pointer exceptions (e.g. access an object through a null reference), indexing exceptions (e.g. attempting to access an array element through an index that is too large or too small), and arithmetic exceptions (e.g. dividing by zero). However, to ensure the server of time-series databases always running normally, unchecked exceptions should also be handled implicitly by the target system. Consequently, Unicorn also captures them for potential bugs.

The other type of exception is the checked exception. The checked exceptions may not represent a bug when they are thrown because generally, the target program could handle them reasonably. Even though, to avoid missing bugs, the *anomaly detector* still records the detailed error message and database logs for manual verification.

5 IMPLEMENTATION

We implement the Unicorn with 8,657 lines of code. The fuzzer is implemented in Rust using the Tokio asynchronous framework [3]. As Figure 3 shows, Unicorn has two parts, namely hybrid input synthesis and proactive exception detection.

Hybrid Input Synthesis. Hybrid input specification is mainly implemented like *ANTLR* format. We collect a number of traditional input specifications from the Internet to build the general skeleton library in *ANTLR* format. The time-oriented models for each time-series database are also constructed in *ANTLR* format. The time-series mutation is implemented in 1,423 lines of code. For generalization, we implement a converter in 1,075 lines of Python code to make *YACC* format compatible with our specification. We instrument databases to collect coverage feedback. For Java databases, we implement it on *ASM* framework. And for C/C++ databases, we implement it based on Clang.

Proactive Exception Detection. Detection is based on *agent* and *anomaly detector*. For VM-based time-series databases, the agent is implemented in 2,147 lines of Java code with ASM framework. For OS-based databases, the agent is implemented in 1,646 lines of C++ and Rust code and leverages *ptrace* for sub-processes error detection. In addition, we implemented the anomaly detector in Unicorn with 1,345 lines of Rust code.

6 EVALUATION

In this section, we evaluate the effectiveness of Unicorn in terms of coverage and bug discovery against state-of-the-art database fuzzers—SQLsmith and SQLancer. Besides, we also evaluate the effectiveness of two parts of Unicorn, namely hybrid input synthesis, as well as proactive exception detection.

6.1 Evaluation Setup

Target Time-Series Databases. To evaluate the generality of Unicorn, six popular and widely-used open-source time-series databases are selected as the targets, namely IoTDB, KairosDB, QuestDB, TimescaleDB, TDEngine, and GridDB. Table 1 shows the features of those target databases.

Table 1: Features of Chosen Time-series Databases

Target	Language	Schema	Grammar	Dependency
IoTDB	Java	tree-based	SQL-like	N/A
KairosDB	Java	tag-based	Http API	Cassandra
QuestDB	Java	column-based	SQL-like	N/A
TDEngine	C++	table-based	SQL-like	N/A
TimescaleDB	C	relational	SQL-like	PostgreSQL
GridDB	C++	tag-based	SQL-like	N/A

Compared Fuzzers For the performance comparison, we have also adapted the blackbox fuzzers SQLsmith and SQLancer, which are widely used to test relational databases in industry, to those time-series databases. Other fuzzers such as Squirrel [38] are preliminary evaluated in Section Section 7 because of the limitation of their language support.

- **SQLsmith** [31] is one of the state-of-the-art generation-based DBMS Fuzzer. It continuously generates a large amount of syntactically correct database inputs with the pre-built abstract syntax tree model and sends them to the target database server. If the database server crashes, a bug is very likely to be detected.
- **SQLancer** [25] is a fuzzer for hunting logic bugs in DBMS. It detects logic bugs by constructing invariant oracles [26–28] and checking whether results violate semantic logic. For example, it synthesizes queries to fetch a random row from an existing relational table in the database. If the database fails to fetch that, then the database might have a bug.

Performance Metrics. We use the number of basic blocks covered and the number of bugs detected as two basic metrics. For a fair comparison, when we finish fuzzing, we collect the seeds generated by each fuzzer and dry-run the seeds to uniform the basic block coverage. To deduplicate unique bugs, we extract and compare their call stacks. We also manually trace back to their relevant source code to further confirm. For evaluations, we ran each fuzzing instance (one fuzzer + one target database) for 12 hours with 5 repeated times.

Experiment Environment. We perform the evaluation on a machine with 40 cores (Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz), 256 GiB RAM, and Ubuntu 20.04 as the operating system. Each fuzzing instance ran individually with 1 CPU and 8GiB of RAM. The initial seeds were collected from the built-in integration tests.

6.2 Overall Performance

Unicorn performs better than SQLsmith and SQLancer both in the number of basic blocks covered and bugs triggered. Specifically, in our 12-hour experiment, Unicorn covered 115.75% more basic blocks on four successfully adapted databases than other fuzzers'

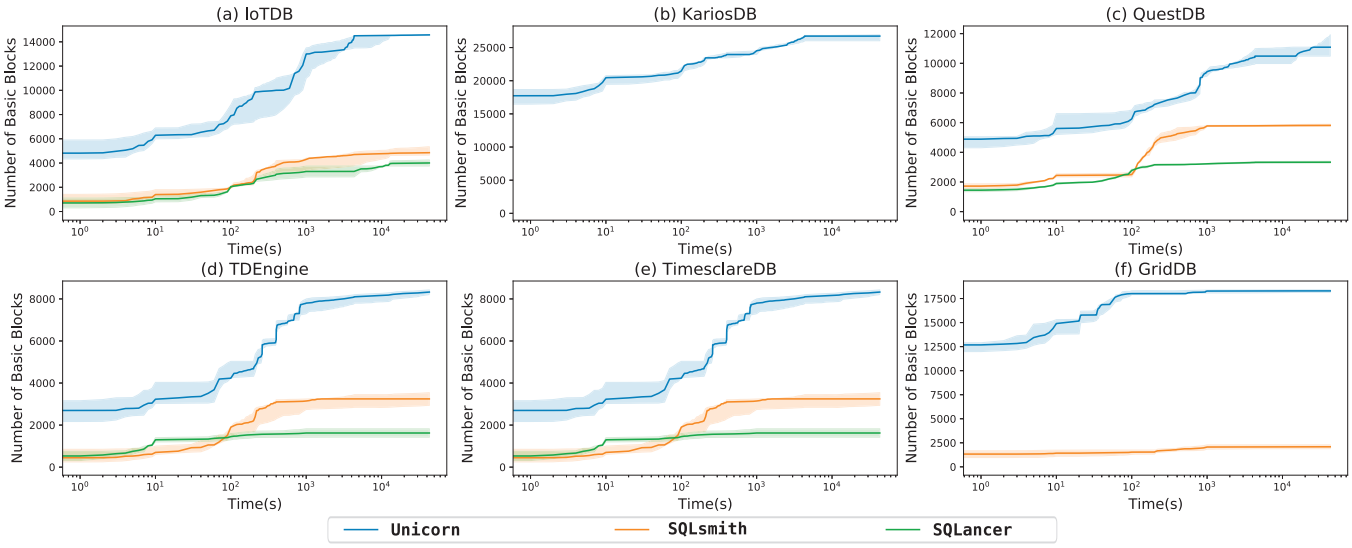

Figure 7: The growing trend of the number of basic blocks covered when fuzzing each time-series database by Unicorn, SQLsmith, and SQLancer over 5 runs in 12 hours. Displayed are the median and the 95% confidence intervals.

best results on average. Also, it found 42 previously unknown bugs, while other fuzzers detected 2 at most.

Coverage. Table 2 demonstrates the number of basic blocks covered by each fuzzer in 12-hour experiments. From the table, we can see that Unicorn outperformed other fuzzers in terms of basic block coverage. Compared to SQLsmith and SQLancer, on the four successfully adapted databases for both of them (i.e. IoTDB, QuestDB, TDEngine, and TimescaleDB), Unicorn covered 115.75% more basic blocks on average than their best results. In particular, Unicorn covered 690% more basic blocks than SQLancer in GridDB. Unicorn outperforms other fuzzers because it promises syntax correctness and time series by hybrid input synthesis. In contrast, SQLsmith and SQLancer do not match the time-series characteristics because SQLsmith and SQLancer only focus on the relational query in the relational model. In addition, Unicorn performs well on all tested time-series databases while SQLancer and SQLsmith both can hardly adapt to KairosDB. Because KairosDB does not accept the SQL-like inputs but the JSON format input by HTTP-API, the generation model of SQLsmith and SQLancer can hardly generate inputs like that. Instead, Unicorn constructs the hybrid input specification for input generation by adding the time-series elements to the JSON grammar specification. As a result, Unicorn adapts KairosDB well to explore its state space.

Figure 7 shows the growth trends of the number of basic blocks for each fuzzer over 12 hours. By comparing the curves of Unicorn, SQLsmith and SQLancer in each sub-figure, we find that Unicorn is faster to cover basic blocks on target time-series databases than both SQLsmith and SQLancer. Specifically, in Figure 7 (a), Unicorn covers more basic blocks in IoTDB than SQLsmith and SQLancer from start to the end. The results on other time-series databases also have similar trends. The main reason is that the inputs generated by SQLsmith and SQLancer lack time-series elements, so most of them are invalid. Unicorn introduces time-series elements through hybrid input synthesis. The initial seeds are syntactically correct, which come from the test cases of time-series databases themselves. Based on them, hybrid input synthesis generates more valid inputs, which are helpful to explore the states of the target time-series databases. Therefore, Unicorn achieves higher coverage at the beginning and keeps ahead to the end.

Table 2: Average Number of Basic Blocks on Time-Series Databases For Each Fuzzer.

Target	SQLsmith	SQLancer	Unicorn
IoTDB	4, 857	3, 754	14, 532
KairosDB	N/A	N/A	26, 489
QuestDB	5, 742	3, 298	10, 486
TDEngine	3, 245	1, 423	8, 473
TimescaleDB	26, 423	13, 485	32, 124
GridDB	N/A	2, 286	18, 145
Total	40, 276	24, 246	110, 249

Table 3: Total Number of Detected Bugs on Time-Series Databases By Each Fuzzer.

Target	SQLsmith	SQLancer	Unicorn
IoTDB	1	0	18
KairosDB	N/A	N/A	2
QuestDB	0	0	3
TDEngine	0	0	6
TimescaleDB	1	1	3
GridDB	N/A	0	10
Total	2	1	42

Bugs. Table 3 shows the number of bugs discovered in six time-series databases by Unicorn, SQLsmith, and SQLancer. All these bugs have been confirmed by the database developers. Totally, Unicorn detected 42 previously unknown bugs. Among them, 23 were from databases implemented in Java, which run in VM. Furthermore, 2 bugs were also found by SQLsmith, while SQLancer did not detect any bug. Unicorn found more bugs mainly for two reasons. First, Unicorn finds more basic blocks. With more basic blocks covered, Unicorn could explore more state space in the target time-series databases, which assigns more possibility for Unicorn to detect more bugs. Second, Unicorn is able to detect bugs that are handled implicitly, while others cannot. For instance, of the discovered 18 bugs in IoTDB, only 5 bugs directly cause the server crash. Although some unchecked exceptions are caused by real bugs, such as returning the wrong results to the client, storing wrong time-series data, and memory explosion, IoTDB handles them implicitly without triggering any crashes. Consequently, other fuzzers will ignore them because the exceptions are handled by the database itself. Instead, with proactive exception detection, Unicorn detects them by obtaining the exception message from the runtime environment.

6.3 Effectiveness of Hybrid Input Synthesis

To evaluate the effectiveness of hybrid input synthesis, we compare Unicorn against Unicorn-no-hybrid and Unicorn-random. Unicorn-no-hybrid disables the hybrid input synthesis. Specifically, it disables the time-oriented model and only uses traditional SQL specifications. Correspondingly, its mutation changes data or structures without distinguishing whether they are related to time-series features. Unicorn-random further disables all specifications. It is like AFL that uses random mutation to generate new seeds. All three versions of Unicorn use the same initial seeds.

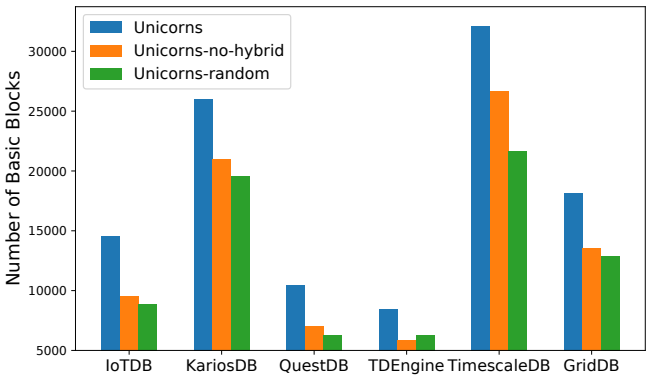

Figure 8: Number of Basic Blocks for Unicorn, Unicorn-no-hybrid, and Unicorn-random in 12h.

Figure 8 shows the number of basic blocks covered after fuzzing 12h with Unicorn, Unicorn-no-hybrid, and Unicorn-random. It shows the importance of hybrid input synthesis for improving coverage. First, the hybrid input synthesis is positive to Unicorn. Compared to Unicorn-random, Unicorn improves 32.8% block coverage on average in 12h. In particular, it covered 48.1% more basic blocks than that of Unicorn-random in TimescaleDB. The main reason is the random mutation can not match the requirements of highly-structured specification; most inputs generated by Unicorn-random have errors in syntax. For example, given "CREATE timeseries root.st0.senor0 with datatype=INT64" as the original time-series query, the generated input by Unicorn-random might like "CREdTE timeseriPS root.st0.seonhe with datatype=INT64;", which can not be parsed syntactically.

More importantly, we find that the time-series model in hybrid inputs specification plays a vital role in the Unicorn's performance. Unicorn covered 28% more basic blocks than Unicorn-no-hybrid on average. Unicorn-no-hybrid can hardly adapt to the time-series characteristics of time-series databases, though it can generate highly-structured inputs. For instance, the inputs generated for IoTDB are usually like "CREATE TABLE root.st0.senor (int value);", which does not conform to time-series specifications. Differently, Unicorn imports the time-oriented model into traditional SQL specifications to construct hybrid input specifications. Besides normal SQLs, it could generate lots of queries which implies time-series features. As a result, Unicorn covers more basic blocks than Unicorn-no-hybrid in the experiments.

Uncommonly, Unicorn-no-hybrid covered 23.1% basic block than that of Unicorn-random in TimescaleDB, which is significantly better than that in other databases. However, Unicorn-no-hybrid still performs worse than Unicorn. The phenomenon can also be explained by the time-series specification. Different from other databases, TimescaleDB runs as the plugin of PostgreSQL. It not only receives time-series data queries at runtime, but also executes traditional SQL statements for PostgreSQL. With the specification of PostgreSQL, the inputs generated only with mutations on AST can still be executed normally. Nevertheless, the inputs generated guided by traditional SQL specifications can only trigger the execution logic of PostgreSQL, while the time-oriented inputs can trigger the execution logic of both TimescaleDB and PostgreSQL. For those reasons, Unicorn-no-hybrid performs much better than Unicorn-random in TimescaleDB, but it still covers fewer blocks than Unicorn.

6.4 Effectiveness of Exception Detection

As shown in Section 6.2, Unicorn performs well in detecting time-series database bugs. Because Unicorn utilizes proactive exception handling, it detects many bugs which could not be found by other fuzzers that employ conventional test oracles. To further illustrate the effectiveness of proactive exception handling, we first collect the seeds causing the bugs from Unicorn, then rerun them with Unicorn-no-detection, which closes the proactive exception detection and only detect anomalies that cause crashes.

Table 4 shows the number of bugs found with two methods. The second column presents the number of bugs found by Unicorn-no-detection. For comparison, we also list the number of bugs detected by Unicorn in the third column. From the table, we can see that proactive exception detection is critical in discovering bugs. Unicorn-no-detection will wrongly consider some buggy seeds as normal. Specifically, without proactive exception detection, Unicorn-no-detection only finds 21 bugs using the same seeds, which is only half as much as that of Unicorn. In particular, Unicorn-no-detection only detected 5 crashes in IoTDB, which

Table 4: Number of Bugs Found by Unicorn Without Proactive Exception Detection.

Target	Unicorn-no-detection	Unicorn
IoTDB	5	18
KairosDB	1	2
QuestDB	0	3
TDEngine	4	6
TimescaleDB	3	3
GridDB	8	10
Total	21	42

missed 13 bugs than Unicorn. The main reason is that the missed bugs by Unicorn are handled implicitly, which do not cause the crash of time-series databases. Unicorn uses agents to collect all exceptions directly from the runtime environment, including the ones handled by implicit mechanisms. As a result, Unicorn captures most bugs triggered by abnormal inputs. Table 5 also lists the statistics of 42 discovered bugs. Among them, 33 have been fixed by developers. Among the 42 new bugs discovered by Unicorn, 21 of them found by Unicorn did not directly crash the time-series databases, but these implicitly handled bugs also caused a series of problems such as loss of data, incorrect results, and out of memory.

Table 5: Statistics of Unknown Bugs Detected by Unicorn

Project (Fixed/Detected)	Component	Bug Type and Number
IoTDB (18/18)	db.qp	Check metadata error (3), NumberFormatException (1), Aggregation Error (1)
	db.metadata	BufferUnderflowException (2), Internal server error(1), NullPointerException (1),
	db.engine	ClassCastException (1), ClosedByInterruptException (1), StorageEngineFailureException (1)
	tsfile	IndexOutOfBoundsException (1), NullPointerException (1), ClassCastException (1)
	thrift	TTransportException (1)
	db.concurrent	OutOfMemoryError (1)
	cluster	IndexOutofBoundsException (1)
KairosDB (1/2)	db	WriteTimeoutException (1)
	concurrent	NullPointerException(1)
QuestDB (1/3)	griffin	Aggregation Error (1)
	cairo	InvalidColumnException (1)
QuestDB	cutlass	Infinite loop (1)
TDengine (5/6)	query	Heap-Buffer-Overflow (1), Segmentation Fault (1)
	vnode	Assertion Failure (1), Buffer Overflow (1)
	common	Assertion Failure (1)
TimescaleDB (3/3)	server	Segmentation Fault (1), Assertion Failure (2)
GridDB (5/10)	server	Assertion Failure (1)
	utility	Segmentation Fault (1), Assertion Failure (3)
	server	Heap-Buffer-Overflow (1),Pointer-Overflow (2), Implicit-Conversion (2)
Total	19 components	33 fixed, 42 confirmed

7 DISCUSSION

We discuss several limitations of our current implementation of Unicorn and our plan to address them in future work.

Overhead of Monitoring All Implicit Exceptions. With the proactive exception detection, Unicorn can detect amounts of implicitly handled exceptions from the runtime environment. The proactive exception detection does import overhead in time-series query fuzzing, but it is also helpful to detect abundant exceptions. On the one hand, proactive exception detection does not increase too much additional execution time. For example, Unicorn executed 696, 245, and 754, 263 seeds for IoTDB during 24-hours fuzzing when the proactive exception detection is on and off, respectively. It only increases 8% execution time to detect all the exceptions from runtime environments rather than just find the crash bugs. On the other hand, monitoring all the implicit exceptions helps discover bugs. In our practice, Unicorn discovered 112 exceptions and found 14 unique bugs in IoTDB after fuzzing 24 hours with the proactive exception detection. But when we closed the exception detection, only 5 crashes were found. Overall, while monitoring all exceptions increases the overhead of Unicorn, It also assigns the ability to detect bugs that may cause serious damage to time-series databases in production applications. We believe reducing the overhead is significant to speed fuzzing. We plan to optimize it by reducing capture messages of *agent*.

Cost of Adapting to New Time-series Databases. The scalability of database fuzzers plays a crucial role in industrial applications. Generally, the most labor-consuming task is building the input model of target databases. For example, SQLancer uses 8,134 lines of Java code to support generating syntax-correct queries. Different from conventional fuzzers, Unicorn doesn't cost much to adapt to new time-series databases. With the collected `ANTLR` format grammar of traditional SQL specification, we only need to manually append less than 100 lines of code for the time-oriented model.

Compare Against Coverage-Guided Database Fuzzers. Most coverage-guided database fuzzers are unable to test time-series databases directly because their instrumentation only supports C/C++. In contrast, Unicorn supports the heterogeneous implementation of C/C++ and Java. To further investigate Unicorn's performance, we also compare against Squirrel [38], the state-of-the-art coverage-guided database fuzzer, on two C++ time-series databases, namely TimescaleDB and TDEngine. The results show that Unicorn performed better than Squirrel. Squirrel covered 19, 843 basic blocks on TimescaleDB and 5, 439 basic blocks on TDEngine after fuzzing 12 hours, while Unicorn covered 32, 124 and 8, 473 basic blocks on two databases, respectively.

8 RELATED WORK

In this section, we focus on most related works on testing databases, along with the works for IoT domain testing.

Test Query Generation. Based on the generated test data, many database testing tools focus on generating various types of queries. To guarantee the syntax correctness of queries, a common approach is constructing the generation model. SQLsmith [31] generates queries with a pre-built generation model, and it randomly generates various queries. Squirrel's generation model is an Abstract Syntax Tree (AST). It parses each query to construct the query's AST and mutates the node in AST to generate new queries that correct in syntax. However, constructing parsers for generation

is laborious. For example, SQLsmith uses 42 elements to adapt the `select` statement of PostgreSQL. In addition, these generation models are built based on the relational storage model of relational databases. It can not perform well for the time-series storage model.

Unlike these works, Unicorn automatically constructs hybrid input specification for each time-series database by combining time-oriented models and traditional SQL specifications. With the time-oriented model, the generated queries have abundant time-series features. Besides, Unicorn is adaptable. Generally, it only needs to append less than 100 lines of code to the traditional specification for the time-oriented model to adapt to a new time-series database.

Test Oracles for Databases. Various test oracles have been proposed to detect various bugs in databases and have achieved good results [13, 25–27, 31, 33, 38]. The basic test oracle is *monitoring whether the tested databases crashes*. SQLsmith [31] determines bugs in databases through the running state of the server. In other words, if the server crashes, there might be a bug in the database. Squirrel [38] and Ratel [36] focuses on detecting memory corruption bugs in databases with the help of AddressSanitizer (ASAN) [10]. It records bugs when ASAN crashes the program. However, limited by the dependency on ASAN, it can only detect bugs in the C/C++ databases. Therefore, it can hardly detect problems for the database implemented in memory-safe languages such as Java. Other tools focus on oracles *related to logic correctness*. SQLancer [25] designs three test oracles according to the logic characteristics of relational databases. Specifically, it constructs inputs to fetch a randomly selected row from a relational table. If the database server fails to return the rows, it should contain a bug. Some other works define test oracles based on *differential testing*. RAGS [33] detects correctness bugs in databases through differential testing. It generates and executes queries in multiple databases. Any inconsistency among results indicates at least one database contains bugs. Apollo [12, 13] is proposed to detect performance regression bugs by comparing the execution time with the same inputs on different versions of the same databases.

Unicorn is different from these works. It designs proactive exception detection to catch the exception message from the runtime environment of databases. Consequently, it can detect bugs even if they are hidden by implicitly handling mechanisms. It is important because these bugs may lead to various serious problems. Based on the detection mechanism, Unicorn can also be adapted to databases implemented in different languages.

IoT Domain Testing. With the development of IoT devices, time-series database testing has attracted more and more attention. And a large number of test benches for time-series databases have been designed. Schemakeit [30] has created a test bench focusing on the writing speed, CPU time, memory usage of time-series databases. Rudoly [29] evaluates different time-series databases for smart space. However, these test benches only test the performance of time-series databases. Different from them, Unicorn detects security problems (e.g. memory errors) in time-series databases. In addition to performance testing, many time-series databases developers [2, 6, 11, 23, 24] also use a large amount of unit tests to ensure their security. However, unit tests can only cover basic functionalities of time-series databases, and some hidden bugs can not be discovered due to low coverage. Differently, Unicorn tests target databases as a whole system. In other words, Unicorn could detect anomalies triggered by the interactions from different components in one database. More importantly, Unicorn generates a large amount of time-series queries automatically, which can cover more basic blocks and detect hidden bugs.

9 CONCLUSION

In this paper, we present Unicorn to automatically detect anomalies in time-series databases with time-series query fuzzing. We first observe that generating grammatically correct time-series queries and detecting anomalies handled implicitly are two major challenges to perform fuzzing on time-series databases. We also find that two characteristics, namely time-series query specification and implicit exception handling, of time-series databases are the root cause for two challenges, respectively. We design a hybrid input synthesis to generate grammatically correct time-series queries to improve the coverage. Furthermore, we employ proactive exception detection, which catches and analyzes the exceptions from runtime environments directly, to detect bugs handled implicitly. We implement the approach in Unicorn. On average, Unicorn covers 115.75% more basic blocks than the best result of the state-of-the-art methods. Furthermore, it detected 42 previously unknown bugs.

ACKNOWLEDGMENTS

This research is sponsored in part by the NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730), and the National Key Research and Development Project (No. 2019YFB1706203, No2021QY0604).

REFERENCES

[1] Mike Aizatsky, Kostya Serebryany, Oliver Chang, Abhishek Arya, and Meredith Whittaker. 2016. Continuous fuzzing for open source software. https://opensource.googleblog.com/2016/12/announcing-oss-fuzz-continuous-fuzzing.html. [Online; accessed 15-May-2021].

[2] ASuherman, Masayoshi KURITA, Katsuhiko Nonomura, and Samurai Chanko. 2016. GridDB: Highly Scalable, In-Memory NoSQL Time Series Database Optimized for IoT and Big Data. https://griddb.org/

[3] Alex Crichton Carl Lerche. 2016. Tokio:Build reliable network applications without compromising speed. https://tokio.rs/. [Online; accessed 15-May-2021].

[4] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 711–725. https://doi.org/10.1109/SP.2018.00046

[5] cppreference. 2017. GESH, MediaWiki. https://en.cppreference.com/w/cpp/error/exception. [Online; accessed 15-May-2021].

[6] TAOS Data. 2019. TDengine: Big Data Platform Designed and Optimized for the Internet of Things (IoT). https://www.taosdata.com/

[7] Michael Eddington. 2015. Peach Fuzzer: Discover unknown vulnerabilities. http://web.archive.org/web/20210121202148/https://www.peach.tech/. [Online; accessed 15-May-2021].

[8] Philippe Esling and Carlos Agon. 2012. Time-series data mining. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 1–34.

[9] Tak-chung Fu. 2011. A review on time series data mining. *Engineering Applications of Artificial Intelligence* 24, 1 (2011), 164–181.

[10] Google. [n.d.]. AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html. [Online; May 13, 2022].

[11] Vlad Ilyushchenko, Brian Thomas Smith, and TheTanc. 2014. QuestDB: Database Designed to Process Time Series Data of IoT, Faster. https://github.com/questdb/questdb

[12] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2020. APOLLO: a toolchain for automatically detecting, reporting, and diagnosing performance bugs in DBMSs. https://github.com/sslab-gatech/apollo

[13] Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woonhak Kang. 2020. APOLLO: Automatic Detection and Diagnosis of Performance Regressions in Database Systems (to appear). In *Proceedings of the 46th International Conference on Very Large Data Bases (VLDB)*. Tokyo, Japan.

[14] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, and Taesoo Kim. 2019. Finding semantic bugs in file systems with an extensible fuzzing framework. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 147–161.

[15] lcamtuf. 2017. American Fuzzy Lop (AFL). http://lcamtuf.coredump.cx/afl/

[16] Jie Liang, Mingzhe Wang, Chijin Zhou, Zhiyong Wu, Yu Jiang, Jianzhong Liu, Zhe Liu, and Jiaguang Sun. 2022. PATA: Fuzzing with Path Aware Taint Analysis. In *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA. 154ś170.*

[17] libfuzzer@googlegroups.com. 2020. libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html

[18] Rui Liu and Jun Yuan. 2019. Benchmark Time Series Database with IoTDB-Benchmark for IoT Scenarios. *CoRR* abs/1901.08304 (2019). arXiv:1901.08304 http://arxiv.org/abs/1901.08304

[19] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. 2019. Polar: Function Code Aware Fuzz Testing of ICS Protocol. *ACM Trans. Embed. Comput. Syst.* 18, 5s (2019), 93:1–93:22. https://doi.org/10.1145/3358227

[20] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. 2020. ICS protocol fuzzing: coverage guided packet crack and generation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[21] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12, 32–44. https://doi.org/10.1145/96267.96279

[22] Oracle. 2017. Java Exception. https://docs.oracle.com/javase/tutorial/essential/exceptions/runtime.html. [Online; accessed 15-May-2021].

[23] Fernando Paladini and Brian Hawkins. 2014. KairosDB: Fast Time Series Database on Cassandra. https://kairosdb.github.io/

[24] Manuel Rigger. 2019. Apache IoTDB: Database for Internet of Things. https://iotdb.apache.org/

[25] Manuel Rigger. 2020. SQLancer: detecting logic bugs in DBMS. https://github.com/sqlancer/sqlancer

[26] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-Optimizing Reference Engine Construction. In *Proceedings of the 2020 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Sacramento, California, United States) *(ESEC/FSE 2020)*. https://doi.org/10.1145/3368089.3409710

[27] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 211:1–211:30. https://doi.org/10.1145/3428279

[28] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 667–682. https://www.usenix.org/conference/osdi20/presentation/rigger

[29] Christoph Rudolf. 2017. SQL, noSQL or newSQL–comparison and applicability for Smart Spaces. *Network Architectures and Services* (2017).

[30] Mitja Schmakeit, Florian Stinner, DJH Ziegeldorf, Ing Klaus Wehrle, and Ing Dirk Müller. 2017. Performance Evaluation of Low-Overhead Messaging Protocols and Time Series Databases via a Common Middleware. (2017).

[31] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2018. SQLsmith: a random SQL query generator. https://github.com/anse1/sqlsmith

[32] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2020. SQLsmith: Score list. https://github.com/anse1/sqlsmith/wiki#score-list

[33] Donald R. Slutz. [n.d.]. Massive Stochastic Testing of SQL. In *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, New York, USA*. Morgan Kaufmann, 618–622.

[34] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nickolai Zeldovich (Eds.). ACM, 344–358. https://doi.org/10.1145/3477132.3483547

[35] Chen Wang, Xiangdong Huang, Jialin Qiao, Tian Jiang, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin Mcgrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. 2020. Apache IoTDB: Time-series database for Internet of Things. *Proc. VLDB Endow.* 13, 12 (2020), 2901–2904. https://doi.org/10.14778/3415478.3415504

[36] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. 2021. Industry practice of coverage-guided enterprise-level DBMS fuzzing. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 328–337.

[37] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 818–834.

[38] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. Squirrel: Testing Database Management Systems with Language Validity and Coverage Feedback. In *The ACM Conference on Computer and Communications Security (CCS), 2020.*

[39] Chijin Zhou, Mingzhe Wang, Jie Liang, Zhe Liu, and Yu Jiang. 2020. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 858–870.

[40] Feilong Zuo, Zhengxiong Luo, Junze Yu, Zhe Liu, and Yu Jiang. 2021. PAVFuzz: State-Sensitive Fuzz Testing of Protocols in Autonomous Vehicles. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 823–828.