Towards Generic Database Management System Fuzzing

Yupeng Yang and Yongheng Chen, Georgia Institute of Technology; Rui Zhong, Palo Alto Networks; Jizhou Chen and Wenke Lee, Georgia Institute of Technology

https://www.usenix.org/conference/usenixsecurity24/presentation/yang-yupeng

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.
August 14–16, 2024 • Philadelphia, PA, USA

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.
Towards Generic Database Management System Fuzzing

Yupeng Yang†, Yongheng Chen†, Rui Zhong*, Jizhou Chen†, Wenke Lee†
†Georgia Institute of Technology  *Palo Alto Networks

Abstract
Database Management Systems play an indispensable role in modern cyberspace. While multiple fuzzing frameworks have been proposed in recent years to test relational (SQL) DBMSs to improve their security, non-relational (NoSQL) DBMSs have yet to experience the same scrutiny and lack an effective testing solution in general. In this work, we identify three limitations of existing approaches when extended to fuzz the DBMSs effectively in general: being non-generic, using static constraints, and generating loose data dependencies. Then, we propose effective solutions to address these limitations. We implement our solutions into an end-to-end fuzzing framework, BUZZBEE, which can effectively fuzz both relational and non-relational DBMSs. BUZZBEE successfully discovered 40 vulnerabilities in eight DBMSs of four different data models, of which 25 have been fixed with 4 new CVEs assigned. In our evaluation, BUZZBEE outperforms state-of-the-art generic fuzzers by up to 177% in terms of code coverage and discovers 30x more bugs than the second-best fuzzer for non-relational DBMSs, while achieving comparable results with specialized SQL fuzzers for the relational counterpart.

1 Introduction
Database Management Systems (DBMSs) play an indispensable role in ensuring effective and efficient data storage, retrieval, and management in modern cyberspace. The landscape of DBMSs has evolved significantly, with the emergence of both relational (SQL) and non-relational (NoSQL) databases catering to the diverse requirements of various application domains [13, 16]. While relational DBMSs have been extensively studied and employed for decades, non-relational DBMSs such as key-value DBMSs, document DBMSs, and graph DBMSs have gained widespread adoption more recently due to their flexibility and performance advantages in handling large-scale, unstructured data. Considering the prevalence and criticality of these systems, it is paramount to strengthen the security and robustness of the diverse DBMSs.

Fuzzing, an automated software testing method that injects random data as inputs to software, has proven useful in uncovering faults in DBMSs. However, there exists a disparity in the extent of fuzzing efforts directed towards non-relational DBMSs compared to their relational counterparts. Fuzzing frameworks and research related to relational DBMSs [27, 28, 43, 44, 49, 56] have been developed and advanced extensively over the years, contributing to more secure and trustworthy systems in the relational DBMS venue. In contrast, non-relational DBMSs have not experienced the same level of scrutiny. State-of-the-art generic fuzzing frameworks such as [4, 7, 9, 14, 30, 54] all show non-promising fuzzing performances when applied to non-relational DBMSs because they cannot generate test cases that trigger DBMS behaviors effectively. There is a need for an effective solution capable of testing both relational and non-relational DBMSs.

However, multiple challenges exist when designing a fuzzer that extends to non-relational DBMSs. First, it is hard to generalize. The interfaces of non-relational DBMSs are diverse, accepting inputs ranging from key-value command sequences [40, 45], JSON documents [2, 33] to graph patterns represented in ASCII-art forms [5, 35, 41]. The diversity of the interfaces presents a unique challenge in designing a generic framework that handles the diverse types of DBMS interfaces effectively, because the semantics of the interfaces can vary drastically across different DBMS categories, putting us in a dilemma between promoting test case quality and maintaining the fuzzer’s generalizability. Second, semantics can change based on the context. For example, in the graph query language Cypher used by many graph DBMSs, one syntax structure can either define some data or use some data, depending on the syntactic context. Also, in the key-value DBMS redis, the type of some keys depends on the value specified in the context. For many non-relational DBMSs, failure to model such semantics leads to semantically incorrect test cases and can hardly reach deep logic. Existing methods used by relational DBMS fuzzers cannot scale to non-relational DBMSs in general because they do not consider the semantics based on contexts. Third, random mutations tend to generate
loose data dependencies, triggering less effective behaviors. Some fuzzers employ coverage as feedback, but the mutation process is still random and can waste time generating test cases triggering less effective behaviors, taking longer time to discover new coverage. We observe that data dependency plays an important role in effective DBMS fuzzing. Without data dependency, even semantically correct test cases can demonstrate less effective behaviors. For example, a test case that creates the same data 100 times does not trigger more behaviors of the DBMS than a test case that first creates the data once and then reads the data, even though they are both semantically correct.

In this work, we systematically analyze these challenges and propose three solutions, namely, semantics abstraction, context-sensitive constraint resolution, and dependency-guided mutation. Then, we implement the solutions into an end-to-end fuzzing framework, BUZZBEE, which can effectively fuzz both relational and non-relational DBMSs in general. To solve the first challenge, BUZZBEE models the DBMS semantics at a highly abstract level where the semantic differences are neutralized. To solve the second challenge, BUZZBEE incorporates an advanced annotation system, through which users can easily specify simple and expressive semantics based on the context for different DBMSs. Finally, to solve the third challenge, BUZZBEE performs novel principled mutations utilizing data dependencies as guidance, generating useful test cases more efficiently.

We implement BUZZBEE with 9,130 lines of code in C++ and Python, and apply it to 8 DBMSs, including relational DBMSs and three types of mainstream non-relational DBMSs: key-value DBMS, graph DBMS, and document DBMS. We evaluate BUZZBEE on the DBMSs and find 40 vulnerabilities, of which 25 have been fixed, and 4 new CVEs have been assigned. We also compare BUZZBEE with six state-of-the-art fuzzing frameworks. BUZZBEE achieves up to 177% performance in finding new program states and finds 30x more bugs when compared with the second-best generic fuzzer.

In summary, we make the following contributions:

- We systematically analyze the challenges for fuzzing the diverse DBMS interfaces, including relational and non-relational ones. We propose novel solutions that effectively tackle them.
- We implement a prototype of our solutions into an end-to-end fuzzing framework: BUZZBEE, enabling effective fuzzing for both relational and non-relational DBMSs.
- We perform extensive evaluations for BUZZBEE on eight mainstream DBMSs of four data models. BUZZBEE has identified 40 bugs in the DBMSs we tested on.

We release the source code of BUZZBEE at https://github.com/OMH4ck/BuzzBee.

2 Problem

In this section, we first briefly discuss the diversity of DBMS interfaces and how they handle user requests in general. Then, we show the challenges and limitations of existing fuzzers when applied to the various kinds of DBMS interfaces. Next, we analyze these unique challenges and present our insights in tackling them. Finally, we introduce our novel approaches and framework design for solving the problem.

2.1 Diverse DBMS Interfaces

DBMSs handle user requests by exposing different kinds of interfaces to the users. DBMSs with distinct data models demonstrate significant variations in their interfaces. Relational DBMSs [11, 12, 36, 39] often accept inputs in Structured Query Language (SQL), through which the user can manipulate the data stored within the database. Meanwhile, non-relational DBMSs [2, 5, 33, 40, 41, 45] have more diverse interfaces, which are associated with a wide variety of non-relational data models. They accept various input formats, including command sequences, JSON documents, and even ASCII-art. Based on how user inputs are processed, we divide these interfaces into two main categories: query-based interfaces and command-based interfaces. Query-based interfaces accept user inputs in the form of a query language. For example, SQL and Cypher [35] are the domain-specific languages used to interact with relational and some graph DBMSs respectively. Such inputs go through a parser and a query planner before hitting the query execution stage [8]. The parser checks the syntax validity of the query, filtering out syntax invalid inputs. The planner verifies the semantics of the query and performs analysis to generate an optimized query execution plan. Any input that fails to pass the parser or the query planner will most likely fail to trigger deep logic inside the target DBMS. Command-based interfaces accept a series of commands from the user and evaluate them in sequence. Although these commands are typically executed independently, meaning that an error in executing one command is unlikely to abort the remaining commands, command sequences with semantic errors or inadequate semantic dependency may still trigger fewer effective behaviors.

2.2 Existing Challenges and Limitations

In this section, we discuss three limitations of current approaches in fuzzing the diverse DBMS interfaces.

Non-generic. As discussed in §2.1, the test case quality is important for DBMS testing. However, the variety of DBMS interfaces complicates the creation of a fuzzing framework that can generate high-quality test cases and remain easily adaptable to different DBMS interfaces. Current research has made significant advancements in relational DBMS testing [27, 28, 43, 44, 49, 56]. S Q Lsmith [44] generates ran-
where a variable is available for use. This is often enforced through data operations such as define, use, and invalidate. Constraints obey to avoid a DBMS execution error. This includes static constraints to represent the semantic rules the test case should be aware of. However, consider the Cypher commands shown in Fig.1b, we know a node would result in both identifier won’t work.

(a) PostgreSQL examples (b) Cypher examples

(c) redis commands

**Fig.1: Static Constraint Examples.** Example test cases illustrating the problem of static constraints. While using static constraint sufficiently models the semantics in Fig.1a, it does not correctly model the scope constraints in Fig.1b and the type constraints in Fig.1c.

We first illustrate the problem of static scope constraints. We show the grammar for the “CREATE TABLE” statement in PostgreSQL in lines 2-6 of Fig. 1a. A static approach binds the constraint “TABLE define” to the node “tbl_name”, and enforces this constraint during the traversal of the AST. Here, this approach is sufficient for modeling the semantics. However, consider the MATCH clause in the graph DBMS interfacing language Cypher. The grammar is shown in lines 2-12 of Fig.1b. Lines 15-16 show a Cypher query that first defines a variable n (i.e., “(n:L)”), and then uses the variable in the path pattern (i.e., “(n)->[->]”). Here, we cannot bind the constraint “variable define” to the identifier AST node, because both occurrences of n are of the identifier type. Binding the constraint “variable define” to the identifier AST node would result in both n treated as “variable define”. In fact, when an identifier is in the subtree of a pattern_part node, it means “variable define”. And when it is in the subtree of a where_part node, it means “variable use”. Binding a static scope constraint to any node cannot correctly model this semantics.

We next show the problem of static type constraints. Consider the redis commands shown in Fig.1c. In redis, users can use the HINCRBY command to increase the value of a field in a set created by HSET. However, if the field does not contain a numeric value, the HINCRBY command fails. Consequently, the HINCRBY command at line 4 fails to trigger the value increasing logic because it tries to operate on k1_field1, first to parse the test case using the target’s grammar specification, and then bind a static constraint to a particular syntactic structure, i.e., an AST node. However, in the case of fuzzing diverse DBMS interfaces, we observe that many constraints should not stay static, but should adapt to various contexts, such as the node’s position in the AST, the existence of other nodes in the AST, the text of literal nodes, etc.
a field pre-defined but storing a value of an ASCII string instead of a numeric string. Adopting the existing approach, we can only let the AST node of k1 yield type "HSET key", and let the node of k1_field1 yield type "HSET field", losing the information that the value implicitly says k1_field1 is an ASCII string instead of a numeric string. Here, using a static type constraint without looking at the values (i.e., literals "He1Io" and "123") loses vital information to model the semantics correctly.

**Loose Data Dependencies.** Existing mutation-based DBMS fuzzers [27, 28, 56] perform mutations randomly and rely on code coverage to explore more program behaviors. However, we observe that random mutations can waste huge efforts in fuzzing non-relational DBMSs, because they tend to generate loose data dependencies that trigger thin and less effective behaviors. The issue becomes significant when the database interface contains many operations that are not dependency-affiliative, which are common cases in non-relational DBMSs such as redis. We define two operations as dependency-affiliative when they can form a data dependency. We observe that test cases triggering deep logic require dependency-affiliative operations often. For instance, a test case containing two "Create key of type A" operations does not trigger more behaviors than a test case containing one "Create key of type A" operation and one "Delete key of type A" operation that deletes the created key. In this case, the latter two commands can form a data dependency and trigger deeper logic.

Fig. 1c shows a real-world example for redis. The second HSET command is not dependency-affiliative with the first HSET command because they cannot form any data dependency. In contrast, the HMCRBY command at line 5 is dependency-affiliative with the first two HSET commands since it can use the data defined by them, i.e., k1, k1_field1, k2, and k2_field1. To demonstrate the ineffectiveness of random mutation, we implement a fuzzer for redis that randomly mutates the test cases by either inserting new commands or mutating existing commands’ arguments randomly. Fig. 2 illustrates the mutation process. We randomly sample 2643 commands from redis’s official test suite as the mutation source by sampling at most 30 commands from each command type. The commands are deduplicated when they are identical (i.e., sharing the same type and arguments). The fuzzer starts with an initial test case "HMSET k1 k1_field1 1". In the first round of mutation (level 1), the fuzzer inserts a new command below HMSET. At this level, there are 2643 candidates available for insertion. However, only around 5.86% (155 / 2643) of them are dependency-affiliative with HMSET, which can use the variable k1 defined by HMSET. All other mutations (around 94.14% of the total) are less effective because they cannot utilize existing data (i.e., they either operate on a different data type, or do not contain any data operations at all), and thus inserting them into the test case contributes little to exploring deeper logics. One of the CVEs found by BUZZBEE (shown in Fig. 2) lies within the code that processes the HMFIELD command, which needs a data dependency to trigger. It is more difficult for a random fuzzer to find such bugs.

### 2.3 Our Insights and Solutions

In this section, we describe our insights and approaches to solving the aforementioned challenges. We first propose **Semantics Abstraction** to support modeling the diverse semantics of DBMS interfaces. Next, we utilize **Context-sensitive Constraint Resolution** to support general context-sensitive constraints. These two solutions help to achieve generalizability and improve the test case semantic correctness. Finally, we design **Dependency-guided Mutation** to tackle the challenges faced by random mutation, which helps to reduce wasted efforts and achieve better fuzzing efficiency.

**Semantics Abstraction.** To support fuzzing the diverse DBMS interfaces, we propose to abstract the semantics to a point where most semantic differences are neutralized. To achieve this goal, we first describe common DBMS operations at a highly abstract level using only three basic data operations: Define, Use, and Invalidate, which in turn defines a symbol, uses or updates a symbol, and deletes a symbol. We then use constraints to constrain the highly abstract semantics, such as specifying when to Define or Use, the type of the symbol that is Defined, or the type that a Use can take, etc. Inspired by existing works [27, 28, 56], we design an IR to incorporate both the syntactic structures and the abstract semantics of the inputs. Afterward, we design an **Annotation System** for users to specify the semantic constraints. Following this way, we neutralize the semantic differences across diverse DBMS interfaces and stay generic.

**Context-sensitive Constraint Resolution.** To avoid the problem of static constraints, we enable dynamic context-based constraints with **Context-sensitive Constraint Resolution**. Specifically, we craft two additional features for the Annotation System, through which users can specify the constraints based on the context. To achieve simplicity, we design a lightweight domain-specific language (DSL) for the users to query common context information with minimum effort. To achieve expressiveness, we expose an interface for users to write complex semantic rules based on the context. Together, they give users the opportunity to customize context-sensitive rules easily and effectively.

**Dependency-guided Mutation.** To avoid generating test cases of loose data dependencies, we propose **Dependency-guided Mutation**. Specifically, the mutator prioritizes mutations that can form new data dependencies by favoring operations that are dependency-affiliative with the data existing in the context. For instance, when mutating a test case that creates some data in the DBMS, the mutator favors inserting an operation that reads the created data, instead of an operation that creates the data again or operations doing nothing related to the created data. Through dependency-guided mutation,
we can reduce the redundant efforts and minimize the time wasted by the fuzzer, achieving better fuzzing efficiency.

3 Overview of BUZZBEE

We incorporate the proposed solutions into an end-to-end fuzzing framework named BUZZBEE. Shown in Fig.3 is an overview of BUZZBEE. Overall, BUZZBEE takes as input the corpus (i.e., initial test cases) and input specs (i.e., the annotation file and grammar file) of the target DBMS interface. Afterward, it performs mutation on the test cases and uses the mutated test cases to test the target DBMS. Guided by coverage feedback, BUZZBEE continuously tries to discover test cases covering new program states and outputs the ones triggering bugs along the process. Specifically, the user provides the grammar file to describe the syntax and the annotation file to constrain the abstract semantics of the target DBMS interface. Next, BUZZBEE conducts semantic analysis and performs principled mutations using data dependencies as the guidance. Lastly, BUZZBEE validates the semantics by fixing the errors introduced by mutation and generates new test cases for testing the DBMS.

In the next sections, we discuss in detail how the components of BUZZBEE are designed and how they collaborate together to support Generalizability §4, Context-sensitive Constraints §5, and Principled Mutation §6, which address the three aforementioned challenges respectively.

4 Generalization

Mutation can easily break the semantic correctness of a test case, as mentioned in many related works [27, 56]. BUZZBEE uses the abstract semantic model to check for the semantic correctness of the mutated test case. Similar to existing works, when BUZZBEE detects semantic errors, it tries to fix them before sending them to the fuzzing runtime. In this section, we discuss in detail how BUZZBEE stands out by modeling the semantics of the various DBMS interfaces at a highly abstract level to neutralize their differences.

BUZZBEE abstracts the DBMS interface semantics with three basic data operations: Define, Use, and Invalidate. Each operation is associated with a data type and name.

Define. Define represents data creation. For instance, in Fig. 1c, the first redis command HSET Defines data k1 with type "HSET key", and k1_field1 of type "HSET field of k1". Similarly, in Fig. 1a, the PostgreSQL query CREATE TABLE Defines two data: t1 of type table, and c1 of type "col_of_t1" meaning it’s a column of t1. Notice that these data can have subordination relations, e.g., k1_field1 is subordinate to k1, meaning it cannot exist without k1. The same applies to t1 and c1. Such relations can be modeled through our DSL Context Query Language (CQL), and Custom Resolvers, which we will detail later. The general idea is that we put concrete symbol names in the type of the data, e.g., the type of k1_field1 is "HSET field of k1", which contains the concrete symbol name k1, meaning that k1_field1 is affiliated to k1.

Use. Use represents data access and update operations on already Defined data. For example, the HINCRBY commands at lines 4 and 8 in Fig.1c Use data k1, k1_field1, k2, and k2_field1 which are Defined by the two HSET commands. Use of un-Defined data is considered a semantic error.

Invalidate. Invalidate represents data deletion operations on already Defined data. Once data is Invalidated, it cannot be Used again. Notice that data deletion should honor the subordination relations. For instance, in Fig. 1c, if we uncomment the DEL command at line 7, it will delete k2. Since k2_field1 has a subordination relation with k2, k2_field1 should also be deleted. Similarly, in Fig.1a, if we delete table t1, its column c1 should also be deleted. After an Invalidate operation, any Use of already Invalidated data will become a semantic error. Consequently, if we uncomment the DEL command at line 7 in Fig.1c, line 8 will yield two semantic errors, since k2 and k2_field1 are both invalidated.

We design an IR to carry both the syntactic and semantic information specified by the user. Inspired by existing works [27, 28, 56], BUZZBEE’s IR is a tree structure with a one-to-one mapping to the abstract syntax tree of the original test case. Besides, the IR captures the abstract semantics specified by the user. This allows us to conveniently lift the original test case into the IR, perform semantic analysis, mutations, and validations on the IR in a unified fashion, and compile the IR back to a new test case for fuzzing. We detail the structure of the IR in Fig.8.

5 Semantics Constraining

After BUZZBEE models the DBMS interfaces at a high level, it uses constraints to further express richer semantics. In this section, we introduce how BUZZBEE enforces flexible yet lightweight context-sensitive semantic constraints for different DBMS interfaces. Overall, BUZZBEE first uses the Annotation System powered by CQL and Custom Resolvers to accept customized semantics information from the user for different DBMS interfaces. Then, the Semantics Analyzer of BUZZBEE analyzes the (mutated) test cases according to the user-specified semantics information and get the semantics of the test cases. Having the semantics of the test cases, the Semantics Validator enforces the specified semantic constraints of the test cases, before sending them to the fuzzing runtime.

5.1 Annotation System

To handle different semantics across different DBMS interfaces, we design the Annotation System to let the user annotate the semantics of a target DBMS within our abstract
Then, in lines 1-4 of the annotations shown in Fig.4d, we specify specific semantics for (1), which maps to the element key under the grammar rule hset. After parsing, this corresponds to an AST node key under a parent AST node hset. With grammar tags, BUZZBEE can effectively correlate the semantic information with the AST nodes of the input, allowing the user to conveniently annotate the semantic properties directly through the grammar.

**Annotation.** The annotation consists of multiple entries. Each entry describes the semantics of a specific tagged syntax node and contains the following fields:

- **operation**: decides which abstract operation to perform.
- **args**: provide the arguments specific to the operation.

For scope constraints, operation specifies the abstract semantic operation to perform, *i.e.*, *Define*, *Use*, or *Invalidate*. An annotation entry can have multiple operations that should be selected under different contexts. For type constraints, we can use args to specify the data type of an operation, *e.g.*, what type a node *Defines* or should *Use*.

The unique aspect of our design is that all the constraints can be dynamically resolved based on the context, including deciding which operation to perform based on the context, and how to resolve the arguments (*e.g.*, types) of an operation based on the context. This is crucial to our solution to the second challenge described in §2.3. It enables BUZZBEE to enforce context-sensitive constraint resolution on the input. Next, we introduce how BUZZBEE achieves this through Context Query Language and Custom Resolver, which are designed to achieve simplicity and expressiveness respectively.

**Context Query Language.** To resolve a constraint based on the context, we need to be able to fetch information from the context. To fetch certain information from the context, we need to know *where* to fetch and *what* to fetch. The former specifies which part of the context we care about, and the latter states what property of that part we are interested in. For example, in line 5 of Fig.4a, to resolve the data type of k1_field1, which is "HSET numeric field of k1", we need to know the text (*what*) of its left-side node (*where*), which is k1. This is to specify that we can only use an "HSET numeric field" that belongs to k1. We develop a lightweight query language for BUZZBEE, Context Query Language (*CQL*), that can be used directly in the operation arguments to query such context information. We attach the grammar for *CQL* in Fig.9. *CQL* offers *navigator* and *property* for specifying *where* and...
what. Generally, we can use navigator to navigate to the target node, and then use property to specify what information to query. For the previous example of resolving the field’s type for the HSET command, we annotate the type argument of HSET "HSET numeric field of {.lsib(1)@text}". BUZZBEE will dynamically resolve this type by first evaluating the CQL query within the curly braces, and then formatting the evaluated value into the type string. During CQL evaluation, the navigator performs relative addressing from the current node. Fig. 4f shows a running example of how CQL is evaluated on the IR program for Fig. 4a. The evaluation for (.lsib(1)@text) starts from the IR node field tagged by 4. Then it uses the navigator ".lsib(1)" to reach the left first sibling node: key, and finally uses the property @text to retrieve the source code text of key, which is "k1". Thus, this CQL evaluates to "k1", and the argument will resolve to "HSET numeric field of k1". We show some other navigators in Fig. 4f, including .parent which navigates to the parent IR node, .child which navigates to a child node, etc. When stacked together, navigators can be used to reach any node within the IR program to query any part of the context. We also predefine several frequently used properties such as @id to retrieve the IR node’s id when creating a scope, @sym_type to retrieve the symbol type for enforcing type coherence, etc.

With CQL, users can directly put dynamic, context-dependent values in the argument of the operations. Annotating with CQL is intuitive and easy thanks to its simple design and direct integration into the arguments. However, CQL is designed to handle common context-sensitive type constraints, and it is limited in expressiveness because both navigator and property are hard-coded elements in the grammar and can convey only limited semantics. For instance, CQL currently does not support conditional behaviors, and is hard to use for selecting the appropriate operation when a node has multiple operations (context-sensitive scope constraints). We solve this by Custom Resolvers, which can resolve more complex semantics for different DBMS interfaces.

Custom Resolver. To support more complex semantics, BUZZBEE exposes an application interface to the user, allowing the user to customize programmatically how to resolve a constraint (e.g., the operation to perform, the type to use) through Custom Resolvers. Custom Resolvers can be written in high-level programming languages like C++ and act as plugins to the annotation system. Within a Custom Resolver, users can access all the context information visible to BUZZBEE and customize the resolution rules as needed. Specifically, BUZZBEE passes all the context information (e.g., symbol tables, the IRNodes in the program) to a Custom Resolver, and then waits for it to return a resolved value.

For instance, to resolve the type constraint of the field in the HSET command at line 1 of Fig. 4a, we annotate...
with the name of a custom resolver we implement, hset_field_type_resolver, at line 8 of Fig. 4d. With access to the context information, this custom resolver first goes to the right first sibling of the field node, which is value that contains the string of the field. Then, it checks if the string is a numeric string, which is false. Next, to follow the subordination rule, it goes to the left first sibling of the field node, which is the key associated with this field, and obtains the text of the key: k1. And finally, it returns "HSET field of k1". This resolution is complex since we need to check whether the value is a numeric string or not. Similarly, this Custom Resolver resolves the type of k2_field1 at line 2 as "HSET numeric field of k2", since "123" is a numeric string.

Moreover, Custom Resolvers can easily support context-sensitive scope constraints by returning the appropriate operation to select based on customized rules. For instance, for the example previously mentioned in Fig. 1b, we tag the identifier node with ①, as shown in Fig. 4c, and annotate it with multiple operations, as shown in Fig. 4e. Then, we implement a Custom Resolver ident_selector Resolver as the selector that returns "operation_0" or "operation_1" or NULL based on the context. Specifically, in ident_selector resolver, we return "operation_0" when detecting that the current IRNode is in the subtree of an IRNode of type pattern_part, i.e., this node should Define a variable. We return "operation_1" when detecting the node is in the subtree of a where_part, resolving the scope constraint to a Use. We return NULL for other cases to signal no matching operation. Then, BUZZBEE selects the operation based on the value returned.

BUZZBEE also maintains local states for each test case and allows all the Custom Resolvers to access the states. This provides the opportunity for modeling stateful semantics.

Regardless of its complexity, the customized rules will eventually resolve a value plugged into the Annotation System, allowing BUZZBEE to manage the semantics within its abstract model.

5.2 Semantic Analysis and Validation

Semantics Analyzer is the component that checks for semantic correctness within our abstract semantic model, according to the user-specified constraints. To check the semantics, the analyzer performs an Execution Simulation on the IR program, which executes the abstract semantic operations in the IR program, following the correct order.

The analyzer achieves this through two analysis stages: the Dependency Analysis stage, which tries to figure out the correct execution order, and the Execution Simulation stage, which executes the semantic operations.

Dependency Analysis. BUZZBEE first performs a dependency analysis on the IR program before executing any operations. It goes over every operation, gathers the contexts each operation depends on, and constructs a dependency graph accordingly. Next, it performs topological sorting on the graph to rank all the IR nodes in the IR program. When two IR nodes do not have any dependency relation, BUZZBEE ranks them according to their sequence in the preorder traversal of the IR program. In this way, the dependency analysis can output a safe and correct execution order the IR nodes in the IR program, which will be followed in the execution simulation.

Execution Simulation. Once the correct execution order is determined, BUZZBEE performs execution simulation by executing the semantic operations. That is, for the operation Define, BUZZBEE tries to define a symbol of the specified type and name in the current scope. For Use, it tries to find a symbol in the scope tree that matches the specified type and name and then use it. And for Invalidate, it does the same thing as Use, plus invalidating the symbol so that it cannot be used again. During this process, the semantics analyzer evaluates the CQL queries used in the operation arguments, and invokes the Custom Resolvers if they are specified to resolve certain values. Meanwhile, symbol re-Define, Use before Define, or Use after Invalidate will all be considered semantic errors. Moreover, if the context an operation depends on contains semantic errors, this operation will also be set as a semantic error. BUZZBEE maintains symbol tables and scope trees to track successfully executed operations (i.e., operations with no semantic errors). Finally, the semantics analyzer returns all the symbol and scope information, along with the semantic errors as the analysis result.

Lastly, the Semantics Validator tries to fix the semantic errors in the test case before BUZZBEE sends it to the DBMS. For Use before Define, Use after Invalidate errors, it finds another available data to use. For re-Define errors, it tries to define the data with an undefined name. When the validator fails to fix the errors, BUZZBEE drops the test case.

6 Principled Mutation

As discussed in §2.3, with dependency-guided mutation, we can reduce the redundant efforts and minimize the time wasted by the fuzzer, thus achieving better fuzzing efficiency. The way BUZZBEE abstracts the semantics naturally enables dependency-guided mutation as a principled mutation strategy. Overall, the principled mutator takes a lifted IR program as input. Then, it queries the dependency information about the IR program from the semantics analyzer, and then performs guided mutations accordingly, as illustrated in Fig. 3.

Specifically, the mutator asks the semantics analyzer to perform semantic analysis on the IR program. It then retrieves the symbol and scope information, i.e., what symbols are defined at which locations and scopes. Then, the mutator performs correctness-preserving syntax mutation, which are mutations that will not break the syntax correctness, including
replacing a node $A$ in the IR program with another node $B$ of the same IR type from the IR pool (node replacing), inserting a node $B$ from the IR pool into the IR program (node insertion), and removing a node in the IR program (node deletion). Here, the IR pool is where the mutator collects the unique IR nodes it has seen triggering new coverage in the DBMS. Additionally, the mutator adds guidance to these mutations. For node replacing, it will first get the symbols $s_i \in S$ that are available at $A$, where $S$ are symbols Defined but not yet Invalidated before $A$. Then, when randomly selecting $B$ from the IR pool, BUZZBEE favors $B$ containing IRs tagged with actions Use or Invalidate whose type can match the type of any $s_i \in S$. This is because such $B$ can form a dependency relation with existing symbols in the test case, and thus can often trigger deeper program states. Node insertion follows the same logic to find the $B$ for insertion. We do not design guidance for node deletion at the moment.

Moreover, to cover more behaviors, BUZZBEE prioritizes the candidates of $B$, based on whether the candidate already exists in the test case or not. For instance, in Fig.4a, assume HINCRBY and DEL are the only two redis commands that contain actions Use and Invalidate for the data defined by HSET. Then, at line 4, BUZZBEE prioritizes inserting DEL over inserting another HINCRBY, because there is already an HINCRBY at line 5. This is achieved by assigning weights to the semantic actions. If an action appears more times in the current test case, it gets assigned a lower weight. BUZZBEE randomly selects an action based on its weight, and searches for $B$ from the IR pool containing this action. Eventually, the number of different actions in the mutated test case (e.g., DEL and HINCRBY) will be uniformly distributed, covering more behaviors.

In conclusion, BUZZBEE uses the knowledge readily available in the annotation system to guide the mutation prior to the input’s execution in the fuzzing runtime, favoring dependency-affiliative mutations and driving the mutation towards discovering deeper program states more efficiently.

7 Implementation

We implement a prototype of BUZZBEE with 9,130 lines of code mainly in C++ and Python. Table 1 shows the breakdown of the major components. We implement the Lifter Generator to generate the lifter that lifts the input into BUZZBEE’s IR. The generator takes as input an ANTLR4 [1] grammar and a JSON annotation file. We pick ANTLR4 in our implementation because there already exists many ANTLR4 grammars for various DBMS interfaces on the Internet. The annotation is implemented as a separate JSON file, which maps to the tagged rules in the grammar file. We build the Fuzzing Runtime of BUZZBEE on top of AFL++ [14]. Specifically, BUZZBEE, which focuses on generating high-quality test cases, integrates itself into AFL++ as a Custom Mutator, allowing it to inherit AFL++’s well-tested instrumentation module, execution module, feedback collection module, etc.

8 Evaluation

We evaluate BUZZBEE to answer the following questions:

- Can BUZZBEE apply to the diverse DBMS interfaces (generalizability and portability)?
- Can BUZZBEE find real-world bugs and vulnerabilities (effectiveness)?
- How does each proposed solution contribute to the performance of BUZZBEE?
- How does BUZZBEE compare to state-of-the-art tools?

8.1 Environment Setup

Hardware. We perform all our evaluations on a machine that runs Ubuntu 22.04.2 LTS with two AMD EPYC 7452 32-core Processors and 1,024GB RAM.

Benchmark. We evaluate BUZZBEE on three types of mainstream non-relational DBMSs (i.e., key-value, graph, document DBMSs) and relational DBMSs (i.e., SQL DBMSs). We choose the targets based on their popularity and choose C/C++ targets because the current fuzzing runtime (AFL++) best supports them. Table 5 shows the DBMSs we evaluate BUZZBEE on. Specifically, we choose redis and KeyDB from the key-value category, RedisGraph and AgenGraph from the graph category, MongoDB and ArangoDB from the document category, and PostgreSQL and MySQL from the relational category.

For real-world bug-hunting evaluation, we evaluate BUZZBEE on the latest release version or the dev branch of the chosen targets. We then perform a comprehensive evaluation of the effectiveness of the proposed solutions in terms of semantic correctness, code coverage, and bug detection capability on four targets, namely redis, ArangoDB, RedisGraph, and PostgreSQL. We choose them over the other four DBMSs because they demonstrate higher fuzzing stability and cover all four categories we evaluate. We also compare BUZZBEE with state-of-the-art tools. We compare BUZZBEE with general-purpose fuzzers that support all four DBMS categories: AFL++ [14], RedQueen [4], Polyglot [9], and Grammarinator [22], to understand BUZZBEE’s generalizability and effectiveness. We then compare BUZZBEE with

<table>
<thead>
<tr>
<th>Module</th>
<th>Language</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics Analyzer</td>
<td>C++</td>
<td>4,966</td>
</tr>
<tr>
<td>Semantics Validator</td>
<td>C++</td>
<td>214</td>
</tr>
<tr>
<td>Lifter Generator</td>
<td>Python</td>
<td>823</td>
</tr>
<tr>
<td>Principled Mutator</td>
<td>C++</td>
<td>1,961</td>
</tr>
<tr>
<td>Fuzzing Runtime</td>
<td>C++</td>
<td>185</td>
</tr>
<tr>
<td>Other Utilities</td>
<td>C++</td>
<td>981</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>C++/Python</td>
<td><strong>9,130</strong></td>
</tr>
</tbody>
</table>
well-established SQL fuzzers SQUIRREL and SQLANCER to further understand BUZZEE’s ability to handle relational targets. Some fuzzers do not support certain DBMSs and we detail this information in Table 6. For all the fuzzers we evaluate, we feed them with the same input (if they need one) and constrain the computing power to one CPU core. For bug detection evaluations, we roll back the DBMS versions where all bugs remain unfixed. We run each experiment for 24 hours, repeat five times, and report the average results.

### 8.2 Generalizability and Portability

We apply BUZZEE to eight real-world DBMSs to understand its generalizability and portability in terms of adoption effort. We collect the grammar and input corpus for each DBMS interface from publically available sources such as official documentation pages, GitHub repositories, or other fuzzers’ open-sourced artifacts. Afterward, we manually add the annotation for each DBMS interface. Table 2 shows a breakdown of the artifact details we collect for BUZZEE. The average number of lines in the annotation artifacts is 123.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Corpus</th>
<th>Grammar</th>
<th>Annotation Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>redis</td>
<td>Test Suite</td>
<td>Doc</td>
<td>496</td>
</tr>
<tr>
<td>KeyDB</td>
<td>Test Suite</td>
<td>Doc</td>
<td>100</td>
</tr>
<tr>
<td>ArangoDB</td>
<td>Test Suite</td>
<td>Exist</td>
<td>82</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>SQUIRREL-repo</td>
<td>Exist</td>
<td>71</td>
</tr>
<tr>
<td>MySQL</td>
<td>SQUIRREL-repo</td>
<td>Exist</td>
<td>71</td>
</tr>
<tr>
<td>RedisGraph</td>
<td>Test Suite</td>
<td>Exist</td>
<td>69</td>
</tr>
<tr>
<td>MongoDB</td>
<td>Test Suite</td>
<td>Doc</td>
<td>51</td>
</tr>
<tr>
<td>AgensGraph</td>
<td>ANTLR4-repo</td>
<td>Exist</td>
<td>42</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>123</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Adoption effort for each target. Doc means we collect the grammar by preprocessing the DBMS’s documentation. Exist means there already exists an ANTLR4 grammar for that DBMS interface online. Annotation size is the total number of lines in the JSON file, CQL and Custom Resolvers.

In conclusion, BUZZEE demonstrates great generalizability for major DBMS interfaces, and the adoption effort is reasonable for major DBMS interface categories, demonstrating decent portability.

---

8.3 Real-world Bug Hunting

BUZZEE finds bugs in all eight DBMSs of the four DBMS categories we have tested. BUZZEE finds bugs in the latest versions of the DBMSs except for PostgreSQL. We present the full bug list in Table 7 in the appendix. As of writing, BUZZEE has discovered 40 bugs in the latest DBMSs, out of which 38 are confirmed by the vendors, 25 are fixed, and 4 are assigned new CVE IDs. BUZZEE has yet to discover bugs in the latest PostgreSQL. In fact, no fuzzers we evaluate can find bugs in the latest PostgreSQL using well-tested corpus in our experiment, including SQL-specialized fuzzers SQUIRREL and SQLANCER. Rolling PostgreSQL back to a legacy version, BUZZEE finds a known bug (ID 37).

Next, we conduct case studies of three fixed bugs found by BUZZEE to understand its capabilities further. We have manually minimized the test cases for demonstration.

**Case Study A.** Fig.5a shows a graph query that crashes the RedisGraph server v2.10.8 by triggering a runtime assertion failure. The crash originates in the function that handles star projection (i.e., the "WITH *" part). However, to trigger this bug, a correct data dependency needs to be satisfied. Here, if we change the variable m in "ORDER by m" to an undefined variable (e.g., v0), the server returns "(error) v0 not defined" and does not crash. Through the annotation system, BUZZEE recognizes the defined variables m and n and maintains the correct data dependency, eventually finding this bug.

**Case Study B.** Fig.5b shows a test case that crashes the redis server by triggering an integer overflow. This is the bug demonstrated earlier in Fig.2. The HMSET command creates a hash set k1 and stores a field/value pair (k1_field1, 1) in it. The HRANDFIELD command returns one or more random fields from a hash set. In this case, the processing logic of HRANDFIELD has an integer overflow bug when handling the specially crafted command. Regardless of its seemingly simple structure, we trace back the relevant code and confirm the bug has been in the code base hidden for at least three years. Finding such bugs in redis is non-trivial due to the large number of operations that are not dependency-affiliative.

To solve this problem, when there exists an HMSET command inside the test case, BUZZEE proactively searches for operations that can form new data dependencies and quickly finds

---

1. `GRAPH.QUERY g "WITH 1 AS x MATCH (m),(n) WITH * ORDER BY m \ 2. SKIP 0 LIMIT 99 WHERE m = 0 RETURN 0"`

(a) Assertion Failure in RedisGraph

1. `HMSET k1 k1_field1 1`
2. `HRANDFIELD k1 \ 3. "-9223372036854775808" \ 4. HRANDFIELD set1 k -inf`

(b) Integer Overflow in redis

(c) Assertion Failure in redis

---

**Fig.5: Case studies.** Three bugs found by BUZZEE demonstrate how BUZZEE’s ability to maintain constraints helps find real-world bugs. The examples are manually reduced for demonstration.
the bug with the help of dependency-guided mutation.

**Case Study C.** Fig. 5c shows another interesting test case that crashes the redis server by triggering an assertion failure. This test case first creates a hash set named set1 and then calls `EXPIRE` to invalidate it. Next, at line 3, the `INCRBYFLOAT` command fails to handle an expired symbol and wrongly implements the error handler, leaving the DBMS in a vulnerable state, which can then be crashed by another `RANDFIELD` command shown at line 4. Interestingly, this bug requires a use-after-invalidate data dependency to trigger. We manually modified the annotation for the `EXPIRE` command from `Invalid` to `Use` to test the DBMS’s handling of invalidated data and discovered this one bug. Based on this finding, in the future, we plan to introduce a new mode into BUZZBEE that automatically modifies certain annotations to find bugs that partially violate data dependency rules, such as use-after-invalidate and use-before-define bugs.

In conclusion, BUZZBEE successfully finds bugs in eight popular real-world targets from four major DBMS categories. Its effectiveness has been acknowledged by both its strong bug-finding capability and bug confirmations from the DBMS vendors.

### 8.4 Contributions of the Solutions

To understand the contribution of each solution we propose, we compare BUZZBEE with three variants of BUZZBEE by turning off the solutions gradually. Specifically, we make: BUZZBEE\(^{g}\) by turning off the dependency-guided mutation of BUZZBEE and making the mutations purely random; BUZZBEE\(^{gc}\) by turning off the context-sensitive constraint resolution routines of BUZZBEE\(^{g}\), which is achieved by making all CQLs and custom resolvers return static values and thus ignoring the context; BUZZBEE\(^{gcs}\) by completely stripping the semantic validator of BUZZBEE\(^{gc}\), shutting down the Annotation System as a whole.

We evaluate the four fuzzers on real-world targets, and compare the fuzzing performance differences in terms of the test case semantic correctness, coverage, and bug-finding capabilities. Evaluating semantic correctness allows us to understand how BUZZBEE improves the fuzzing performance under the hood. We dump the test cases during a 24-hour fuzzing session, send them to the target DBMS, and then wait for the execution result provided by the DBMS. When the DBMS finishes executing the test case without reporting an error, we regard the test case as semantically correct. Otherwise, the test case is considered semantically incorrect.

**Semantic Correctness.** As shown in Table 3, BUZZBEE achieves a 0.22 to 626 times higher semantic correctness rate than the other fuzzers. BUZZBEE\(^{gcs}\) has the lowest semantic correctness rate, because it does not enforce any semantics. BUZZBEE\(^{gc}\) improves over BUZZBEE\(^{gcs}\) by enforcing semantics (without context-sensitivity), and thus the test cases it generates have a higher semantic correctness rate. After introducing context-sensitivity, BUZZBEE\(^{g}\) generates more semantic correct test cases. Interestingly, BUZZBEE improves the semantic correctness over BUZZBEE\(^{g}\), which means the guided mutation also helps increase the test case semantic correctness. We manually examine the test cases and discover the reason: with dependency-guided mutation, the mutator favors mutations that form data dependencies, which are semantics described in the annotation. Then, the semantics validator can fix the errors introduced in the mutation and promote semantic correctness. Without guidance, however, the mutator randomly mutates the input and will more likely generate semantics not described in the annotation. For this case, BUZZBEE cannot fix the potential errors, resulting in lower semantic correctness rates. Meanwhile, the semantic correctness rate on redis is higher than the other DBMSs. This is because redis employs more flexible type checks. As shown in Table 2, even BUZZBEE without validating any semantics (BUZZBEE\(^{gcs}\)) achieves a 79.7% correctness rate. Other DBMSs enforce stricter checks and achieve a lower correctness rate.

**Coverage.** As shown in Fig. 6, BUZZBEE finds on average 29.2%, 8.2%, 22.6%, and 36.8% more edges than the other fuzzers in redis, RedisGraph, ArangoDB, and PostgreSQL, re-
spectively. In general, for the targets we evaluate, the edge discovering capability of BUZZBEE degrades when we strip off the component implementing each solution, effectively showing the contribution of each solution to BUZZBEE in discovering more program states.

Interestingly, the performance gain of some solutions vary on different targets. BUZZBEE outperforms other versions best on the target redis shown in Fig. 6a, which adds a 35.5% increase over the non-guided version BUZZBEE[^1], but it only adds 4.1%, 7.3%, and 6.7% coverage increase to BUZZBEE[^2] on RedisGraph, ArangoDB, and PostgreSQL respectively. This is because redis offers more operations that are not dependency-affiliative, which are operations that cannot form any data dependency and trigger deeper program behaviors. Mutating test cases containing such operations has a high chance of wasting time generating operation sequences with no data dependency and is less likely to be efficient. In comparison, other DBMS interfaces contain more operations that are dependency-affiliative. For example, the tables and columns defined in SQL DBMSs can be used in many operations (e.g., SELECT, UPDATE, DELETE, COUNT, SUM, JOIN), and forming data dependencies becomes easier. This shows that the principled mutation performs better at DBMSs containing many operations that are not dependency-affiliative.

Moreover, BUZZBEE[^3] brings only a 3.8% coverage increase over BUZZBEE[^4] for RedisGraph, as shown in Fig. 6b. Interestingly, with the help of context-sensitive constraint resolution, BUZZBEE[^2] then outperforms BUZZBEE[^4] with a 14.4% coverage increase. This is because context-sensitive constraints are necessary to model the semantics of RedisGraph effectively. The underlying reasons are two-fold. First, the grammar specification readily available on the Internet is written in ways that one node could be shared by multiple parent nodes, which requires the context to determine the correct scope constraint. Second, the operation arguments (type constraints) also depend heavily on the context to resolve the correct values. Therefore, without context-sensitive constraint resolution, BUZZBEE[^3] fails to model the semantics effectively and achieves non-promising results.

**Bug Finding.** As shown in Table 4, BUZZBEE finds 12 bugs within 24 hours in the targets, the most among the four variants. Without guided mutation, BUZZBEE[^2] finds only 8 bugs, which are all covered by BUZZBEE. And without context sensitivity, BUZZBEE[^3] finds only 2 of the bugs, because it fails to model the semantics correctly without context. BUZZBEE[^4] finds one bug that others cannot find, because this bug does not comply with the semantic properties specified in the annotation. For further insights, we categorize the bugs by their triggering conditions. Specifically, we mark the property of the bug as Data when it is semantically correct and contains data dependency relations. When the test case is semantically correct but contains no data dependencies (e.g., a single Define or consecutive Defines of different symbols), we mark the bug as Sem. For the bug that is only triggerable by syntax correct but semantic incorrect test cases, we mark it as Syn. As we can see in Table 4, BUZZBEE finds more Data bugs than BUZZBEE[^2]. For bugs 2 and 20 that are marked as Sem, BUZZBEE[^2] successfully finds them, but BUZZBEE[^3] cannot, showing that Context-sensitive Constraint Resolution helps in enforcing the correct semantics of the test cases. Moreover, BUZZBEE[^4] finds two more bugs than BUZZBEE[^3], which are bugs that require semantic correctness to trigger. This demonstrates the effectiveness of the Annotation System as a whole in enforcing semantic correctness. Finally, BUZZBEE[^4] finds one bug (ID 8) that only semantic incorrect test cases can trigger. Other versions cannot find this bug because they will enforce semantic correctness and filter out such test cases.

In conclusion, BUZZBEE’s success comes from all the solutions we have proposed. The contribution of each solution varies when applied to different DBMS targets that have distinct characteristics.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>redis</td>
<td>2</td>
<td>Sem</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>redis</td>
<td>3</td>
<td>Data</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>redis</td>
<td>5</td>
<td>Data</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>redis</td>
<td>6</td>
<td>Data</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>redis</td>
<td>7</td>
<td>Data</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>redis</td>
<td>8</td>
<td>Syn</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>redis</td>
<td>12</td>
<td>Sem</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>RedisGraph</td>
<td>19</td>
<td>Data</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>RedisGraph</td>
<td>20</td>
<td>Sem</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>RedisGraph</td>
<td>21</td>
<td>Data</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>ArangoDB</td>
<td>35</td>
<td>Data</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>ArangoDB</td>
<td>36</td>
<td>Data</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>37</td>
<td>Data</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔</td>
<td>✔ ✔ ✔</td>
</tr>
</tbody>
</table>

**Table 4: Bugs found in 24 hours by each fuzzer on four DBMSs.**

We run each fuzzer five times for 24h and report the bugs it finds during this period. Property Data means the bug needs correct data dependencies to trigger. Sem means the bug does not need data dependencies to trigger, but the triggering test case is semantically correct. Syn means the bug is not semantically correct but is syntactically correct. Syn means the fuzzer does not support the DBMS.

---

8.5 Comparison with Existing Tools

In this section, we compare BUZZBEE with state-of-the-art works regarding code coverage and bug detection capabilities.

We compare BUZZBEE with general-purpose fuzzers AFL++ [14], REDQUEEN [4], syntax-aware fuzzers POLYGLOT [9], Grammarinator [22], and SQL-specialized SQUIRREL [56] and SQLANCER [43]. AFL++ and REDQUEEN are widely used coverage-guided fuzzers that perform syntax-
unaware mutations. REDQUEEN does not support fuzzing C/S programs, and we skip its evaluation on ArangoDB and PostgreSQL. POLYGLOT is a coverage-guided fuzzer tailored for fuzzing compilers such as Clang. It models the semantics of language processors effectively, but cannot model the semantics of DBMS interfaces due to its insensitivity to the context. It is a great tool to compare against because it acts as a coverage-guided syntax-aware mutator for targets outside its support range. We also compare against Grammarinator, a fuzzer currently used by RedisGraph, that generates random programs from a grammar without coverage guidance. For relational DBMSs, we compare BUZZBEE with SQUIRREL [56] and SQLANCEr (PQS [43]), two DBMS fuzzers specialized in SQL DBMS fuzzing. We also evaluate SQLANCEr on ArangoDB because SQLANCEr recently adds support for it. Table 6 shows the DBMSs each fuzzer supports. In this evaluation, we feed the same input to all fuzzers and the same grammar specification to POLYGLOT and Grammarinator.

Coverage. As shown in Fig. 7, BUZZBEE achieves the best edge coverage in 24 hours in all targets except PostgreSQL. BUZZBEE achieves 69.2%, 19.8%, and 76.9% more coverage in redis, RedisGraph, and ArangoDB than the second-best fuzzer POLYGLOT. This performance gain demonstrates BUZZBEE’s contribution to the non-relational DBMS fuzzing venue. On PostgreSQL, BUZZBEE achieves 92.7% code coverage of the state-of-the-art SQL fuzzer SQUIRREL, showing comparable abilities considering BUZZBEE generalizes to many DBMS categories. In Table 3, we notice BUZZBEE achieves a correctness rate of 9.8% on PostgreSQL. This is also comparable with SQUIRREL. Referring to SQUIRREL’s paper [56], it achieves a semantic correctness rate of 11.7% on PostgreSQL, which is only 1.9% higher than BUZZBEE.

Bug Finding. As shown on the right side of Table 4, out of the six existing fuzzers we evaluate, only POLYGLOT finds one bug in redis, which falls into the category of Syntax, meaning this bug only needs syntax awareness to discover. AFL++ and REDQUEEN cannot find any bugs during this period. None of BUZZBEE, SQUIRREL, and SQLANCEr can find bugs in the latest version of PostgreSQL within 24 hours. We roll back PostgreSQL to an older version, and BUZZBEE finds one legacy bug missed by SQUIRREL and SQLANCEr. Notice that BUZZBEE does not perform as well as SQUIRREL and SQLANCEr in terms of code coverage. To understand the reason, we manually analyze this bug and find that it involves a syntax structure SQUIRREL does not model, potentially leading to its missing the bug. SQLANCEr focuses on finding logic errors using specific syntax structures and does not discover the bug as well. The result demonstrates the capability of BUZZBEE in discovering real-world bugs that existing tools cannot discover, proving its fuzzing effectiveness and contribution to real-world bug detection.

In conclusion, BUZZBEE outperforms existing generic solutions in non-relational DBMS fuzzing, while achieving comparable results with tools specialized in relational DBMS fuzzing, in terms of code coverage and bug-finding capabilities.

9 Discussion

9.1 Complex Semantics and Completeness

Some DBMS semantics can be very complex. For instance, the foreign key constraint in SQL DBMSs requires us to track the column reference relations. BUZZBEE can model complex constraints like this through custom resolvers. Specifically for foreign key constraints, we can use custom resolvers to resolve the type constraints of columns and tables, and track the foreign key relation, i.e., which column is a foreign key to which foreign column. Then, if we want to honor the foreign key constraints in "INSERT INTO" statements, we can use custom resolvers to resolve the scope constraints of the inserted values. That is, when the value is inserted into a foreign key column, we resolve the operation as a Use of existing values in the foreign column. Otherwise, we resolve the operation as a Define of a new column value.

However, as shown in Table 3, BUZZBEE does not achieve 100% semantic correctness under its abstract semantic model for any of the targets. In fact, BUZZBEE does not claim to mimic all the semantics of a DBMS perfectly. Instead, BUZZBEE proposes to generically model and enforce the basic but vital DBMS semantics that impacts fuzzing performance. As shown in the evaluation, BUZZBEE achieves a decent fuzzing performance with this methodology.

9.2 Common Database Interface

Common database interfaces (DBIs) have been proposed decades ago and are used extensively nowadays. They provide
a unified layer for developers to integrate storage engines into their products without worrying about the underlying DBMS that actually powers it, which offers advantages in many ways. One may intuitively develop the idea of using common DBIs for fuzzing the diverse DBMS interfaces. During our research, we realized they are unsuitable for fuzzing for several reasons. First, DBIs tend to use a limited number of APIs of the underlying DBMS, and we cannot test the functionalities of the unused ones. Second, many modern DBIs enforce sanity checks before invoking the underlying DBMS for security reasons, adding another impeding layer. We propose solutions that let users annotate abstract semantics of the target DBMS to enable effective DBMS fuzzing.

10 Related Work

10.1 General Fuzzing Strategies

Fuzzing strategies are mainly divided into two categories: generation-based fuzzing and mutation-based fuzzing. Generation-based fuzzing creates test cases from scratch using a specification of the input format [6, 17, 19, 23, 24, 34, 38, 38, 44, 46, 47, 51, 51, 52]. These fuzzers do not require a seed corpus and are well-suited for testing systems with restricted input formats. Mutation-based fuzzing modifies existing test cases to explore new ones [14, 18, 18, 29, 31, 50, 53, 54]. They rely on a seed corpus and mutate the seeds to create new test cases. Mutations are typically guided by coverage feedback. Fuzzers such as AFL [54], AFL++ [14], libFuzzer [29], and Honggfuzz [18] use coverage obtained during code execution to guide the fuzzing process. AFL employs a lightweight yet effective instrumentation technique to track code coverage. AFL++ maintains the core principles of AFL while improving performance by introducing several optimizations. Researchers also develop more efficient feedback mechanisms. DDFuzz [32] incorporates the coverage in the data dependency graph to guide fuzzing. MemFuzz [10] uses memory accesses to guide fuzzing. BUZZBEE is a mutation-based fuzzer that extends AFL++ and focuses on the unique challenges in generic DBMS fuzzing. We believe it can benefit from the advancements in feedback mechanisms as well.

10.2 Language Processors Fuzzing

Fuzzing language processors (e.g., interpreters and compilers) faces unique challenges. These targets require structural inputs with specific semantic properties to be fuzzed effectively. For instance, to fuzz JavaScript engines, CodeAlchemist [21] uses code bricks to maintain semantic correctness during mutation. DIE [37] and Fuzzilli [20] propose strategies to maintain the semantics stressing the just-in-time (JIT) engines. For compilers, CSmith [51] specializes in modeling C semantics to produce entirely correct test cases for C compilers. Tools such as Nautilus [3] and Superion [48] propose generic grammar-based solutions to fuzz more language processors. POLYGLOT [9] further advances using a semantic validation strategy that performs well when no context-sensitive constraint is required. BUZZBEE focuses on the challenges in fuzzing the diverse DBMS interfaces effectively in general.

10.3 DBMSs Fuzzing

Fuzzing DBMSs has been an active research area in recent years [15, 25–27, 43, 44, 49, 56]. Tools like SQLancer [42], SQLsmith [44], and Squirrel [56] have emerged to test relational DBMSs. SQLsmith generates random SQL queries. Squirrel employs semantic validation and coverage feedback to test SQL DBMSs. SQLancer uses differential testing techniques to report inconsistencies in query results. They have been proven successful in finding vulnerabilities in widely used relational DBMSs. Researchers also propose some solutions targeting non-relational DBMSs [26, 55]. Existing techniques present unique solutions targeting specific DBMS categories, but fail to exhibit decent fuzzing performance for the diverse DBMS interfaces in general. As a result, BUZZBEE is designed to address this with a focus on generalizability and effectiveness. Some optimizations proposed in SQL fuzzers seem effective and generalizable to other DBMSs. DynSQL [25] combines the dynamic feedback from SQL DBMSs with fuzzing. Ratel [49] tackles the challenges of fuzzing DBMSs in real-world settings. Griffin [15] mutates the input without relying on grammar by tracking dependencies among SQL statements. These optimizations are orthogonal research, and we think BUZZBEE can potentially benefit from them to improve its performance.

11 Conclusion

In this work, we identify the unique challenges in fuzzing the diverse DBMS interfaces. We propose our solutions and incorporate them into an open-source end-to-end fuzzing framework: BUZZBEE. We conduct a comprehensive evaluation for BUZZBEE, in which BUZZBEE achieves up to 177% code coverage compared with state-of-the-art fuzzers and finds 40 real-world vulnerabilities in mainstream DBMSs, demonstrating its generalizability and effectiveness.

12 Acknowledgment

We thank the anonymous reviewers for their helpful and informative feedback. This material was supported in part by the National Science Foundation (NSF) under grant No. 2229876 and the Defense Advanced Research Projects Agency (DARPA) under contracts N6600121-C-4024. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DARPA.
References


A Appendix

// BuzzBee's IR is a tree structure formed by IRNodes.
// An IRNode has the following properties:
// IRNode(id, type, text, annotation, children, parent, ...);

// BuzzBee lifts the example shown in Fig.4a into the following
// IRNodes, forming the IR program shown in Fig.4d:
node0(id=0, type=testcase, text=NULL, annotation={},
children={&node1}, parent=NULL);
node1(id=1, type=cmds, text=NULL, annotation={},
children={&node2,&node12,&node22,&node32}, parent=&node0);
node2(id=2, type=cmd, text=NULL, annotation={},
children={&node3}, parent=&node1);
node3(id=3, type=hset, text=NULL, annotation={},
children={&node4, &node5, &node7, &node9}, parent=&node2);

... node33(id=33, type=hincrby, text=NULL, annotation={},
children={&node34, &node35, &node37, &node39},
parent=&node32);
node34(id=34, type=terminal, text="HINCRBY", annotation={},
children={}, parent=&node33);
node35(id=35, type=key, text=NULL,
annotation="\"default\":\{operation:Use,
args:\"type\":\"HSET key\"\}\}
children={&node36}, parent=&node33);
node36(id=36, type=field, text="k1", annotation={},
children={}, parent=&node35);
node37(id=37, type=increment, text=NULL,
annotation="\"default\":\{operation:Define,
args:\"type\":\"hset_field_type_resolver\"\}\}
children={&node38, parent=&node33);
node38(id=38, type=terminal, text="k1_field1", annotation={},
children={}, parent=&node37);
node39(id=39, type=increment, text=NULL, annotation={},
children={&node40, parent=&node33});
node40(id=40, type=terminal, text="1", annotation={},
children={}, parent=&node39);

Fig. 8: BUZZBEE's IR structure and the IR program of the example in Fig.4a. The IR program is a tree structure, which is also illustrated in Fig.4f. When lifting, BUZZBEE parses the test case, traverses over the AST of the test case, collects the corresponding annotations, and then creates the IRNodes forming the IR program. The complete source of the test case is stored in the text property of the IRNodes of the type terminal. When compiling, BUZZBEE traverses over the IR program, collects the terminal IRNodes, and concatenates their texts into a test case.

Fig.9: Context Query Language (CQL)
Table 5: Real world targets we test BUZZBEE on. Popularity is the number of Stars of the DBMS’s GitHub repository. X* in Data Model means the DBMS supports other models but we only test the interface for its primary model X.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>LOC</th>
<th>Popularity</th>
<th>Data Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>redis</td>
<td>281K</td>
<td>60.1K</td>
<td>Key-value*</td>
</tr>
<tr>
<td>MongoDB</td>
<td>8.7M</td>
<td>23.8K</td>
<td>Document*</td>
</tr>
<tr>
<td>ArangoDB</td>
<td>6.8M</td>
<td>13K</td>
<td>Document*</td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>1.6M</td>
<td>12.5K</td>
<td>Relational*</td>
</tr>
<tr>
<td>MySQL</td>
<td>5.5M</td>
<td>9.6K</td>
<td>Relational*</td>
</tr>
<tr>
<td>KeyDB</td>
<td>348K</td>
<td>7.6K</td>
<td>Key-value</td>
</tr>
<tr>
<td>RedisGraph</td>
<td>1.7M</td>
<td>1.9K</td>
<td>Graph</td>
</tr>
<tr>
<td>AgensGraph</td>
<td>1.4M</td>
<td>1.3K</td>
<td>Graph</td>
</tr>
</tbody>
</table>

Table 6: DBMSs supported by fuzzers.

<table>
<thead>
<tr>
<th>Fuzzer v.s. / DBMS</th>
<th>redis</th>
<th>RedisGraph</th>
<th>ArangoDB</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUZZBEE</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>AFL++</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>POLYGLOT</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>REDQUEEN</td>
<td>✔</td>
<td>✔</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Grammarinator</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>SQUIRREL</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>✔</td>
</tr>
<tr>
<td>SQLANCER</td>
<td>X</td>
<td>X</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Table 7: Real-world bugs found by BUZZBEE. BUZZBEE found 40 bugs in latest DBMSs during evaluation. * means BUZZBEE found the bug in the latest DBMS, but it was already known by the vendors when we reported it. † means the bug is known and not in the latest DBMS.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>ID</th>
<th>Version</th>
<th>Status</th>
<th>CVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>redis</td>
<td>1</td>
<td>cb1fff3</td>
<td>Fixed</td>
<td>CVE-2022-35977</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>395d801</td>
<td>Fixed*</td>
<td>CVE-2023-22458</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1ec82e6</td>
<td>Fixed</td>
<td>CVE-2023-22458</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7.0.7</td>
<td>Fixed</td>
<td>CVE-2023-22458</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>7.0.8</td>
<td>Fixed</td>
<td>CVE-2023-25515</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7.0.8</td>
<td>Fixed</td>
<td>CVE-2023-25515</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>7.0.8</td>
<td>Fixed</td>
<td>CVE-2023-25515</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>7.0.8</td>
<td>Fixed</td>
<td>CVE-2023-28425</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>a04a4fe2</td>
<td>Confirmed</td>
<td>CVE-2023-28856</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>7.0.10</td>
<td>Fixed*</td>
<td>CVE-2023-28856</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>7.0.8</td>
<td>Fixed*</td>
<td>CVE-2023-28856</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>a04a4fe2</td>
<td>Fixed</td>
<td>CVE-2023-28856</td>
</tr>
<tr>
<td>KeyDB</td>
<td>13</td>
<td>6.3.2</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>6.3.2</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>6.3.2</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>6.3.2</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>6.3.2</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>18</td>
<td>6.3.2</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>v2.10.8</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>c425ad8</td>
<td>Fixed</td>
<td>CVE-2023-25310</td>
</tr>
<tr>
<td></td>
<td>21</td>
<td>v2.10.9</td>
<td>Confirmed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22</td>
<td>v2.10.9</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23</td>
<td>v2.10.10</td>
<td>Confirmed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>24</td>
<td>v2.10.10</td>
<td>Confirmed</td>
<td></td>
</tr>
<tr>
<td>RedisGraph</td>
<td>25</td>
<td>v2.10.9</td>
<td>Confirmed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>v2.10.9</td>
<td>Reported</td>
<td></td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>v2.10.9</td>
<td>Reported</td>
<td></td>
</tr>
<tr>
<td></td>
<td>28</td>
<td>v2.10.9</td>
<td>Confirmed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>29</td>
<td>v2.10.9</td>
<td>Confirmed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>v2.10.9</td>
<td>Fixed*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>31</td>
<td>v2.10.9</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>AgensGraph</td>
<td>32</td>
<td>v2.13.1</td>
<td>Confirmed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>33</td>
<td>v2.13.1</td>
<td>Confirmed</td>
<td></td>
</tr>
<tr>
<td>MongoDB</td>
<td>34</td>
<td>87d48e1</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>ArangoDB</td>
<td>35</td>
<td>v3.11.0</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>36</td>
<td>v3.11.0</td>
<td>Fixed</td>
<td></td>
</tr>
<tr>
<td>PostgreSQL</td>
<td>37</td>
<td>v11.12</td>
<td>Fixed†</td>
<td></td>
</tr>
<tr>
<td>MySQL</td>
<td>38</td>
<td>v8.1.0</td>
<td>Confirmed*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>39</td>
<td>v8.1.0</td>
<td>Confirmed*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>v8.1.0</td>
<td>Confirmed*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>41</td>
<td>v8.1.0</td>
<td>Confirmed</td>
<td></td>
</tr>
</tbody>
</table>