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 DBMS, document DBMS, and graph DBMS 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]. `SQLsmith` [44] generates ran-

```
// Partial grammar rules:
createtbl_stmt:
    'CREATE TABLE'
    tbl_name '('
        ...
    ')';

// The test case:
> CREATE TABLE t1(
     c1 date
);


// Bind "TABLE define"
// to `tbl_name`.

// When we traverse to the
// node `t1`, we know a
// TABLE t1 is defined.
```

(a) PostgreSQL examples

```
// Partial grammar rules:
match_clause:
  MATCH pattern_part where_part;

pattern_part:
  node_pattern;

where_part:
  WHERE node_pattern;

node_pattern:
  '(' identifier ')' | ...;

// The test case:
> MATCH (n:L) WHERE
(n)-[]->() RETURN n.x;

// Binding "Variable define"
// to `identifier` won't work.
```

(b) Cypher examples

```
> HSET k1 k1_field1 "Hello"   // stores an ASCII string
> HSET k2 k2_field1 "123"     // stores a numeric string

> HINCRBY k1 k1_field1 1      // increase the stored value
(error) value not an integer

// > DEL k2
> HINCRBY k2 k2_field1 1
(integer) 124
```

(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.

dom SQL queries according to predefined schema information about certain SQL DBMSs. SQUIRREL [56] and later works [27, 28] advance by introducing an intermediate representation (IR) that effectively incorporates SQL syntax and semantics, allowing it to adapt to multiple SQL databases. However, these frameworks still require non-trivial adoption efforts to support a new relational DBMS, as mentioned in [49]. Moreover, they either have not supported non-relational DBMS, or inherently cannot support them due to the model design. Considering the wide use of non-relational DBMSs, we lack a generic solution that enables efficient testing for the diverse DBMS interfaces.

Static Constraints. Mutation-based fuzzing has proven successful in fuzzing modern software. However, for DBMS fuzzing, mutations can easily break the semantic correctness of the test case. Recent studies [27, 28, 56] highlight the significance of maintaining the semantic correctness of the test cases in mutation-based DBMS fuzzing. In this work, we use *constraints* to represent the semantic rules the test case should obey to avoid a DBMS execution error. This includes *scope constraints* and *type constraints*. Scope constraints restrict where a variable is available for use. This is often enforced through data operations such as define, use, and invalidate. Type constraints further restrict the legitimate type of the variable in the data operations. To model the constraints, the approach used by existing fuzzing frameworks [9, 27, 28, 56] is

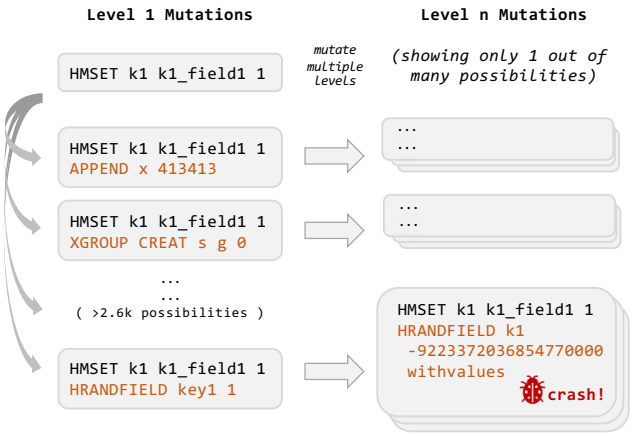


Fig. 2: Random Mutation Running Examples. This figure shows a demo fuzzer we made for testing `redis`. When performing random mutations, the level 1 mutations contain many possibilities that cannot form a data dependency with the initial command, which consequently affects the fuzzing performance at later mutation levels.

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.

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 AST 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 bails out. Consequently, the `HINCRBY` command at line 4 fails to trigger the value increasing logic because it tries to operate on `k1_field1`,

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 `"Hello" 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 `HINCRBY` 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 `HRANDFIELD` 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_field` is `"HSET field of k1"`, which contains the concrete symbol name `k1`, meaning that `k1_field` 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

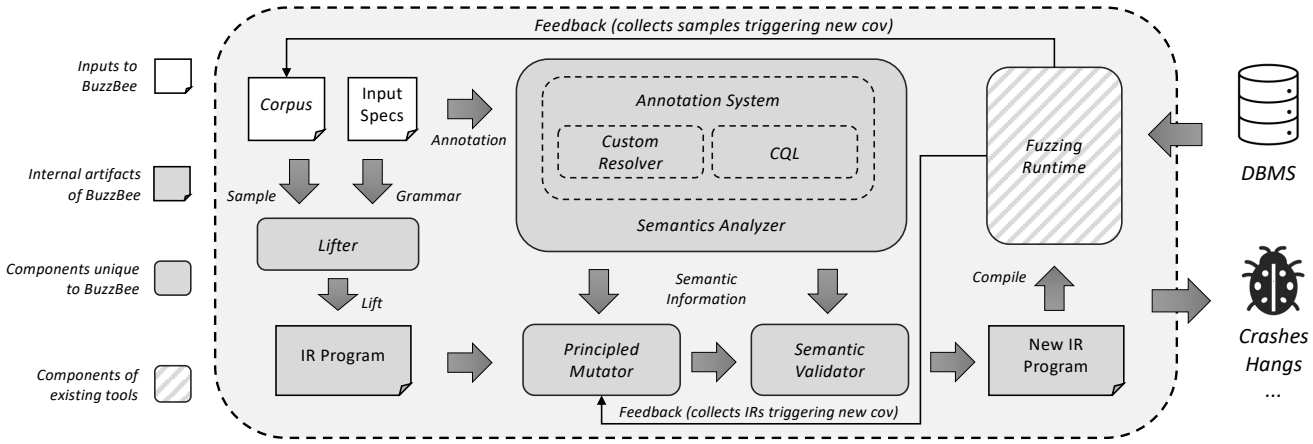


Fig.3: Overview of BUZZBEE. BUZZBEE takes as input the corpus and input specs, performs advanced semantic analysis and mutations with semantics validation on the input, and uses the mutated input to test the target DBMS.

semantic model.We design the *Annotation System* with two objectives in mind: simplicity and expressiveness. Simplicity ensures that the system is lightweight, intuitive, and easy to use so that it can apply to many DBMS interfaces with ease, while expressiveness allows it to convey complex semantics effectively. Generally, the system allows the user to annotate the basic semantic properties of a certain AST node. That is, whether this node *Defines*, *Uses*, or *Invalidates* a certain symbol, and how such semantics will relate to the context of the input. To achieve the first objective, we design *CQL*, a lightweight domain-specific language that can be used to query various context information when resolving semantic constraints. Next, to achieve the second objective, BUZZBEE exposes an interface to the user, allowing the user to design *Custom Resolvers* capable of describing arbitrarily complex rules to resolve the semantic constraints. Shown in Fig. 4 are concrete artifact examples involved in the annotating process. Next, we show through this figure how the system distills the semantic information provided by the user.

Grammar Tags. For the user to directly assign semantics to an AST node, BUZZBEE uses grammar tags to correlate the syntactic structures and the semantic information through the grammar rules. Specifically, we use a markup language that allows the user to tag the elements inside the grammar rules, which will be parsed into AST nodes. Afterward, the user can write specific semantics for tagged AST nodes in an annotation file. Fig. 4b shows the partial grammar rules for `redis` that are tagged with the markup language. For demonstration simplicity, we use circled numbers to represent tagged rule elements. For example, `key` at line 2 of Fig. 4b is tagged with ①. 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

```
HSET k1 k1_field1 "Hello"
HSET k2 k2_field1 "123"
HSET k3 k3_field1 "456"
// DEL k1
HINCRBY k1 k1_field1 1
```

(a) Example commands for `redis`.

```
hset:
    HSET key① (field② value)+;

hincrby:
    HINCRBY key③ field④ increment;

del:
    DEL key⑤;

cmd:
    hset | hincrby | del ;

cmds:
    cmd ('\n' cmd)*;

testcase:
    cmds⑥;
```

(b) Partial grammar rules for `redis`.

```
...
node_pattern:
  '(' identifier① ')';
...
```

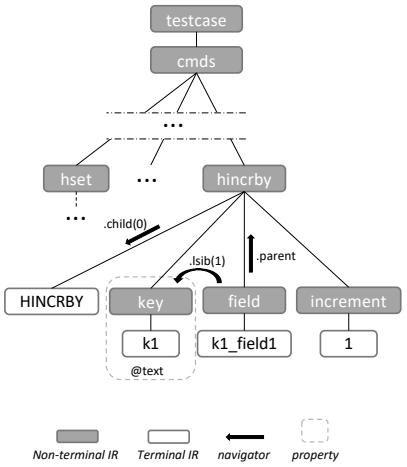
(c) Partial grammar rules for `Cypher`.

```
①: "default": {
      operation: Define,
      args: { type: "HSET key" }
}

②: "default": {
      operation: Define,
      args: {
        type:
            hset_field_type_resolver
      }
}

③: "default": {
      operation: Use,
      args: { type: "HSET key" }
}

④: "default": {
      operation: Use,
      args: {
        type: "HSET numeric field
               of {.lsib(1)@text}"
      }
}

⑤: "default": {
      operation: Invalidate,
      args: { type: ANY }
}
```

(d) Partial annotations for the tags in (b).

```
①: {
    "selector": ident_selector_resolver,
    "operation_0": { operation: Define, ...},
    "operation_1": { operation: Use, ... }
}
```

(e) Partial annotations for the tags in (c).



(f) CQL on the IR program.

Fig. 4: ***Annotation System Internals.*** The circled numbers represent tags to specific syntax structures. *Annotation System* uses these tags to track which syntax nodes should be assigned which semantics.

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 ④ as `"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

② 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

Table 1: Lines of code for a breakdown of BUZZBEE, summing up to 9,130 lines. As we integrate with AFL++, for Fuzzing Runtime, we only count the code that we add into AFL++.

Module	Language	LOC
Semantics Analyzer	C++	4,966
Semantics Validator	C++	214
Lifter Generator	Python	823
Principled Mutator	C++	1,961
Fuzzing Runtime	C++	185
Other Utilities	C++	981
Total	C++/Python	**9,130**

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 `AgensGraph` 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 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.

DBMS	Corpus	Grammar	Annotation Size
`redis`	Test Suite	Doc	496
`KeyDB`	Test Suite	Doc	100
`ArangoDB`	Test Suite	Exist	82
`PostgreSQL`	SQUIRREL-repo	Exist	71
`MySQL`	SQUIRREL-repo	Exist	71
`RedisGraph`	Test Suite	Exist	69
`MongoDB`	Test Suite	Doc	51
`AgensGraph`	ANTLR4-repo	Exist	42
Average			123

well-established SQL fuzzers SQUIRREL and SQLANCER to further understand BUZZBEE'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 DBMSs to the 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 BUZZBEE 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 BUZZBEE. The average number of lines in the annotation artifacts is 123. Thanks to BUZZBEE's highly abstract semantic modeling, most parts of the annotation are simple semantics such as "Node *A* in the grammar *Defines* symbol *s*", and "Node *B* can *Use* symbol *s*", which is easy and intuitive to write by a human analyst. Having familiarity with each DBMS as an average DBMS user, it takes us less than one hour on average to draft a 100-line annotation for a target. We show in later evaluations that spending such little adoption effort is enough for BUZZBEE to achieve a great fuzzing performance.

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

```
GRAPH.QUERY g "WITH 1 AS x MATCH (m),(n) WITH * ORDER BY m \
  SKIP 0 LIMIT 90 WHERE m = 0 RETURN 0"
```

(a) Assertion Failure in `RedisGraph`

```
HMSET k1 k1_field1 1
HRANDFIELD k1 \
    -9223372036854770000 \
    withvalues
```

(b) Integer Overflow in `redis`

```
HSET set1 k 549236
EXPIRE set1 0
HINCRBYFLOAT set1 k -inf
HRANDFIELD set1 -3 WITHVALUES
```

(c) Assertion Failure in `redis`

Fig.5: Case studies. Three bugs found by BUZZBEE demonstrate how BUZZBEE's ability to maintain constraints helps find real-world bugs. The examples are manually reduced for demonstration.

8.3 Real-world Bug Hunting

BUZZBEE finds bugs in all eight DBMSs of the four DBMS categories we have tested. BUZZBEE 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, BUZZBEE 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. BUZZBEE 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, BUZZBEE finds a known bug (ID 37).

Next, we conduct case studies of three fixed bugs found by BUZZBEE 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, BUZZBEE 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, BUZZBEE proactively searches for operations that can form new data dependencies and quickly finds

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 `HINCRBYFLOAT` 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 `HRANDFIELD` 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 *Invalidate* 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

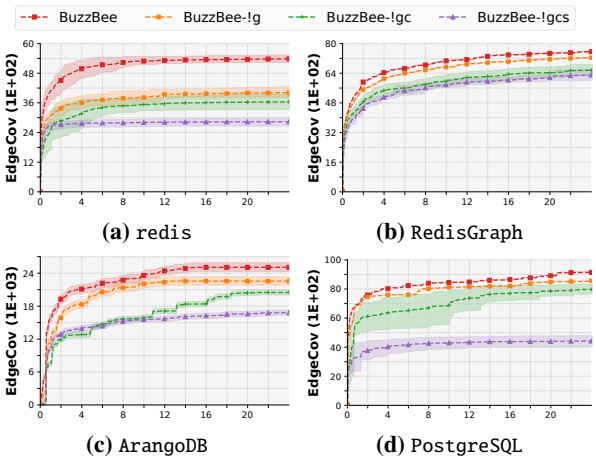
To understand the contribution of each solution we propose, we compare BUZZBEE with three variants of BUZZBEE by turning off the solutions gradually. Specifically, we make: BUZZBEE$^{!g}$ by turning off the dependency-guided mutation of BUZZBEE and making the mutations purely random; BUZZBEE$^{!gc}$ 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 semantics 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.

Table 3: Semantic correctness rates of test cases generated by the four versions of BUZZBEE in 24 hours.

Fuzzers v.s. / DBMSs	`redis`	`RedisGraph`	`ArangoDB`	`PostgreSQL`
BUZZBEE	98%	62.7%	31.6%	9.8%
BUZZBEE$^{!g}$	90.8%	38.5%	25.2%	8.4%
BUZZBEE$^{!gc}$	85.5%	0.3%	18.2%	4.5%
BUZZBEE$^{!gcs}$	79.7%	0.1%	13.6%	1.7%



(a) `redis` **(b)** `RedisGraph` **(c)** `ArangoDB` **(d)** `PostgreSQL`

Fig. 6: Edge coverage found by each version of BUZZBEE in 24h. BUZZBEE$^{!g}$ is BUZZBEE without dependency-guided mutation. BUZZBEE$^{!gc}$ is BUZZBEE$^{!g}$ without context-sensitve constraint resolution. BUZZBEE$^{!gcs}$ is BUZZBEE$^{!gc}$ with the whole annotation system turned off. We repeat the experiments five times and report the average results.

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$^{!g}$, but it only adds 4.1%, 7.3%, and 6.7% coverage increase to BUZZBEE$^{!g}$ 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$^{!gc}$ brings only a 3.8% coverage increase over BUZZBEE$^{!gcs}$ for `RedisGraph`, as shown in Fig.6b. Interestingly, with the help of context-sensitive constraint resolution, BUZZBEE$^{!g}$ then outperforms BUZZBEE$^{!gcs}$ 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$^{!gc}$ 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$^{!g}$ finds only 8 bugs, which are all covered by BUZZBEE. And without context sensitivity, BUZZBEE$^{!gc}$ finds only 2 of the bugs, because it fails to model the semantics correctly without context. BUZZBEE$^{!gcs}$ 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$^{!g}$. For bugs 2 and 20 that are marked as *Sem*, BUZZBEE$^{!g}$ successfully finds them, but BUZZBEE$^{!gc}$ cannot, showing that *Context-sensitive Constraint Resolution* helps in enforcing the correct semantics of the test cases. Moreover, BUZZBEE$^{!gc}$ finds two more bugs than BUZZBEE$^{!gcs}$, 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$^{!gcs}$ 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.

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. ⊖ means the fuzzer does not support the DBMS.

DBMS	ID	Property	BUZZBEE	BUZZBEE$^{!g}$	BUZZBEE$^{!gc}$	BUZZBEE$^{!gcs}$	AFL++	POLYGLOT	REDQUEEN	Grammarin.	SQUIRREL	SQLANCER
`redis`	2	Sem	✔	✔	✗	✗	✗	✗	✗	✗	⊖	⊖
`redis`	3	Data	✔	✔	✔	✗	✗	✗	✗	✗	⊖	⊖
`redis`	5	Data	✔	✗	✗	✗	✗	✗	✗	✗	⊖	⊖
`redis`	6	Data	✔	✔	✗	✗	✗	✗	✗	✗	⊖	⊖
`redis`	7	Data	✔	✗	✗	✗	✗	✗	✗	✗	⊖	⊖
`redis`	8	Syn	✗	✗	✗	✔	✗	✔	✗	✗	⊖	⊖
`redis`	12	Sem	✔	✔	✔	✗	✗	✗	✗	✗	⊖	⊖
`RedisGraph`	19	Data	✔	✔	✗	✗	✗	✗	✗	✗	⊖	⊖
`RedisGraph`	20	Sem	✔	✔	✗	✗	✗	✗	✗	✗	⊖	⊖
`RedisGraph`	21	Data	✔	✗	✗	✗	✗	✗	✗	✗	⊖	⊖
`ArangoDB`	35	Data	✔	✗	✗	✗	✗	✗	⊖	✗	⊖	⊖
`ArangoDB`	36	Data	✔	✔	✗	✗	✗	✗	⊖	✗	⊖	⊖
`PostgreSQL`	37	Data	✔	✔	✗	✗	✗	✗	⊖	✗	✗	✗
Count			12	8	2	1	0	1	0	0	0	0

> 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.

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-

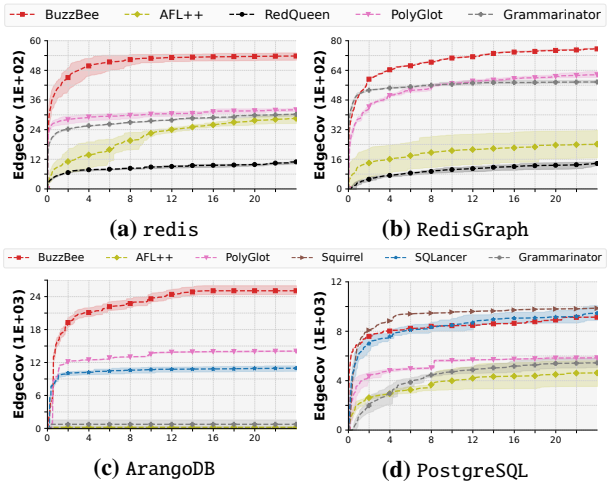


Fig. 7: Edge coverage comparison with existing fuzzers in 24h.

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 *Syn*, 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

[1] ANTLR. Grammars written for ANTLR v4. https://github.com/antlr/grammars-v4, 2020. Accessed: 2023-09-14.

[2] ArangoDB Inc. ArangoDB. https://www.arangodb.com/, 2015. Accessed: 2023-09-14.

[3] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. REDQUEEN: Fuzzing with Input-to-State Correspondence. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[5] Bitnine Global Inc. AgensGraph. https://bitnine.net/, 2017. Accessed: 2023-09-14.

[6] Tim Blazytko, Cornelius Aschermann, Moritz Schlögel, Ali Abbasi, Sergej Schumilo, Simon Wörner, and Thorsten Holz. GRIMOIRE: Synthesizing Structure while Fuzzing. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1985–2002. USENIX Association, 2019.

[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based Greybox Fuzzing as Markov Chain. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1032–1043. ACM, 2016.

[8] Surajit Chaudhuri. An Overview of Query Optimization in Relational Systems. In Alberto O. Mendelzon and Jan Paredaens, editors, *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington, USA*, pages 34–43. ACM Press, 1998.

[9] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. One Engine to Fuzz 'em All: Generic Language Processor Testing with Semantic Validation. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 642–658. IEEE, 2021.

[10] Nicolas Coppik, Oliver Schwahn, and Neeraj Suri. MemFuzz: Using Memory Accesses to Guide Fuzzing. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*, pages 48–58. IEEE, 2019.

[11] Microsoft Corporation. Microsoft SQL Server. https://docs.microsoft.com/en-us/sql/, 1989. Accessed: 2023-09-14.

[12] Oracle Corporation. Oracle Database. https://www.oracle.com/database/technologies/, 1979. Accessed: 2023-09-14.

[13] Ali Davoudian, Liu Chen, and Mengchi Liu. A Survey on NoSQL Stores. *ACM Comput. Surv.*, 51(2):40:1–40:43, 2018.

[14] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining Incremental Steps of Fuzzing Research. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020.

[15] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. Griffin : Grammar-Free DBMS Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 49:1–49:12. ACM, 2022.

[16] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. NoSQL database systems: a survey and decision guidance. *Comput. Sci. Res. Dev.*, 32(3-4):353–365, 2017.

[17] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based Whitebox Fuzzing. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 206–215. ACM, 2008.

[18] Google. Honggfuzz. https://google.github.io/honggfuzz/, 2016. Accessed: 2023-09-14.

[19] Rahul Gopinath, Björn Mathis, Matthias Höschele, Alexander Kampmann, and Andreas Zeller. Sample-Free Learning of Input Grammars for Comprehensive Software Fuzzing. *CoRR*, abs/1810.08289, 2018.

[20] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[21] HyungSeok Han, DongHyeon Oh, and Sang Kil Cha. CodeAlchemist: Semantics-Aware Code Generation to Find Vulnerabilities in JavaScript Engines. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[22] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir, editors, *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST@SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 05, 2018*, pages 45–48. ACM, 2018.

[23] Matthias Höschele and Andreas Zeller. Mining Input Grammars from Dynamic Taints. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 720–725. ACM, 2016.

[24] Bo Jiang, Ye Liu, and W. K. Chan. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In Marianne Huchard, Christian Kästner, and Gordon Fraser, editors, *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, pages 259–269. ACM, 2018.

[25] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4949–4965. USENIX Association, 2023.

[26] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. Testing Graph Database Engines via Query Partitioning. In René Just and Gordon Fraser, editors, *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, pages 140–149. ACM, 2023.

[27] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. Sequence-Oriented DBMS Fuzzing. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 668–681. IEEE, 2023.

[28] Yu Liang, Song Liu, and Hong Hu. Detecting Logical Bugs of DBMS with Coverage-based Guidance. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 4309–4326. USENIX Association, 2022.

[29] libFuzzer. libFuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2015. Accessed: 2023-09-14.

[30] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1949–1966. USENIX Association,

2019.

[31] Chenyang Lyu, Shouling Ji, Xuhong Zhang, Hong Liang, Binbin Zhao, Kangjie Lu, and Raheem Beyah. EMS: History-Driven Mutation for Coverage-based Fuzzing. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[32] Alessandro Mantovani, Andrea Fioraldi, and Davide Balzarotti. Fuzzing with Data Dependency Information. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*, pages 286–302. IEEE, 2022.

[33] MongoDB Inc. MongoDB. https://www.mongodb.com/, 2007. Accessed: 2023-09-14.

[34] MozillaSecurity. funfuzz. https://github.com/MozillaSecurity/funfuzz, 2020. Accessed: 2023-09-14.

[35] Neo4j and openCypher. Cypher Query Language. https://opencypher.org/, 2015. Accessed: 2023-09-14.

[36] Oracle Corporation. MySQL. https://dev.mysql.com/, 1995. Accessed: 2023-09-14.

[37] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. Fuzzing JavaScript Engines with Aspect-preserving Mutation. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1629–1642. IEEE, 2020.

[38] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Model-based Whitebox Fuzzing for Program Binaries. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 543–553. ACM, 2016.

[39] PostgreSQL Global Development Group. PostgreSQL. https://www.postgresql.org/, 1986. Accessed: 2023-09-14.

[40] Redis Ltd. Redis. https://redis.io/, 2009. Accessed: 2023-09-14.

[41] Redis Ltd. RedisGraph. https://oss.redislabs.com/redisgraph/, 2009. Accessed: 2023-09-14.

[42] Manuel Rigger and Zhendong Su. Detecting Optimization Bugs in Database Engines via Non-optimizing Reference Engine Construction. In Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann, editors, *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, pages 1140–1152. ACM, 2020.

[43] Manuel Rigger and Zhendong Su. 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*, pages 667–682. USENIX Association, 2020.

[44] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. SQLsmith. https://github.com/anse1/sqlsmith, 2016. Accessed: 2023-09-14.

[45] Snap Inc. KeyDB. https://docs.keydb.dev/, 2019. Accessed: 2023-09-14.

[46] Spandan Veggalam, Sanjay Rawat, István Haller, and Herbert Bos. IFuzzer: An Evolutionary Interpreter Fuzzer Using Genetic Programming. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine Meadows, editors, *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, volume 9878 of *Lecture Notes in Computer Science*, pages 581–601. Springer, 2016.

[47] Joachim Viide, Aki Helin, Marko Laakso, Pekka Pietikäinen, Mika Seppänen, Kimmo Halunen, Rauli Puuperä, and Juha Röning. Experiences with Model Inference Assisted Fuzzing. In Dan Boneh, Tal Garfinkel, and Dug Song, editors, *2nd USENIX Workshop on Offensive Technologies, WOOT'08, San Jose, CA, USA, July 28, 2008, Proceedings*. USENIX Association, 2008.

[48] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware Greybox Fuzzing. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle, editors, *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pages 724–735. IEEE / ACM, 2019.

[49] Mingzhe Wang, Zhiyong Wu, Xinyi Xu, Jie Liang, Chijin Zhou, Huafeng Zhang, and Yu Jiang. Industry Practice of Coverage-Guided Enterprise-Level DBMS Fuzzing. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*, pages 328–337. IEEE, 2021.

[50] Peng Xu, Yanhao Wang, Hong Hu, and Purui Su. COOPER: Testing the Binding Code of Scripting Languages with Cooperative Mutation. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022.

[51] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011.

[52] Guixin Ye, Zhanyong Tang, Shin Hwei Tan, Songfang Huang, Dingyi Fang, Xiaoyang Sun, Lizhong Bian, Haibo Wang, and Zheng Wang. Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 435–450. ACM, 2021.

[53] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 745–761. USENIX Association, 2018.

[54] Michal Zalewski. American Fuzzy Lop (2.52b). http://lcamtuf.coredump.cx/afl, 2019. Accessed: 2023-09-14.

[55] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. Finding bugs in Gremlin-based graph database systems via Randomized differential testing. In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 302–313. ACM, 2022.

[56] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 955–970. ACM, 2020.

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=terminal, text="k1", annotation={},
      children={}, parent=&node35);
node37(id=37, type=field, 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 *text*s into a test case.

```
cql:
    navigator* property ;

navigator:
    '.parent'
    | '.child' arg | '.lsib' arg | '.rsib' arg
    | ... ;

arg:
    '(' num ')' ;

property:
    '@text' | '@id' | '@sym_type' | ... ;
```

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.

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

Table 6: DBMSs supported by fuzzers.

Fuzzer v.s. / DBMS	redis	RedisGraph	ArangoDB	PostgreSQL
BUZZBEE	✔	✔	✔	✔
AFL++	✔	✔	✔	✔
POLYGLOT	✔	✔	✔	✔
REDQUEEN	✔	✔	✗	✗
Grammarinator	✔	✔	✔	✔
SQUIRREL	✗	✗	✗	✔
SQLANCER	✗	✗	✔	✔

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.

DBMS	ID	Version	Status	CVE
redis	1	cb1fff3	Fixed	
	2	395d801	Fixed*	CVE-2022-35977
	3	1ec82e6	Fixed	CVE-2023-22458
	4	7.0.7	Fixed	CVE-2023-22458
	5	7.0.8	Fixed	CVE-2023-25515
	6	7.0.8	Fixed	CVE-2023-25515
	7	7.0.8	Fixed	CVE-2023-25515
	8	7.0.8	Fixed	CVE-2023-28425
	9	af0a4fe2	Confirmed	
	10	7.0.10	Fixed*	CVE-2023-28856
	11	7.0.8	Fixed*	
	12	af0a4fe2	Fixed	
KeyDB	13	6.3.2	Fixed	
	14	6.3.2	Fixed	
	15	6.3.2	Fixed	
	16	6.3.2	Fixed	
	17	6.3.2	Fixed	
	18	6.3.2	Fixed	
RedisGraph	19	v2.10.8	Fixed	
	20	c425ad8	Fixed	CVE-2023-25310
	21	v2.10.9	Confirmed	
	22	v2.10.9	Fixed	
	23	v2.10.10	Confirmed	
	24	v2.10.10	Confirmed	
	25	v2.10.9	Confirmed	
	26	v2.10.9	Reported	
	27	v2.10.9	Reported	
	28	v2.10.9	Confirmed	
	29	v2.10.9	Confirmed	
	30	v2.10.9	Fixed	
	31	v2.10.9	Fixed*	
AgensGraph	32	v2.13.1	Confirmed	
	33	v2.13.1	Confirmed	
MongoDB	34	87d48e1	Fixed	
ArangoDB	35	v3.11.0	Fixed	
	36	v3.11.0	Fixed	
PostgreSQL	37	v11.12	Fixed†	
MySQL	38	v8.1.0	Confirmed*	
	39	v8.1.0	Confirmed*	
	40	v8.1.0	Confirmed*	
	41	v8.1.0	Confirmed	