Dinkel: Testing Graph Database Engines via State-Aware Query Generation

Dominic Wüst[*]
ETH Zurich
Switzerland

Zu-Ming Jiang[*]
ETH Zurich
Switzerland

Zhendong Su
ETH Zurich
Switzerland

ABSTRACT

Graph database management systems (GDBMSs) store and manipulate graph data and form a core part of many data-driven applications. To ensure their reliability, several approaches have been proposed to test GDBMSs by generating queries in Cypher, the most popular graph query language. However, Cypher allows queries with complicated state changes and data dependencies, which existing approaches do not support and thus fail to generate valid, complex queries, thereby missing many bugs in GDBMSs.

In this paper, we propose a novel state-aware testing approach to generate complex Cypher queries for GDBMSs. Our approach models two kinds of graph state, *query context* and *graph schema*. Query context describes the available Cypher variables and their corresponding scopes, whereas graph schema summarizes the manipulated graph labels and properties. While generating Cypher queries, we modify the graph states on the fly to ensure each clause within the query can reference the correct state information. In this way, our approach can generate Cypher queries with multiple state changes and complicated data dependencies while retaining high query validity. We implemented this approach as a fully automatic GDBMS testing framework, Dinkel, and evaluated it on three popular open-source GDBMSs, namely Neo4j, RedisGraph, and Apache AGE. In total, Dinkel found 60 bugs, among which 58 were confirmed and 51 fixed. Our evaluation results show that Dinkel can effectively generate complex queries with high validity (93.43%). Compared to existing approaches, Dinkel can cover over 60% more code and find more bugs within the 48-hour testing campaign. We expect Dinkel's powerful test-case generation to benefit GDBMS testing and help strengthen the reliability of GDBMSs.

KEYWORDS

Graph database, Cypher query language, testing.

1 INTRODUCTION

Graph database management systems (GDBMSs) are crucial for modern interconnected, data-driven computer software. By storing data in graph structures, GDBMSs efficiently manage data through graph properties (*e.g.*, pairs of data connected through certain relationships). Because of their high efficiency, GDBMSs have been widely adopted in large language models [33, 47], recommendation systems [48], social networking [32], and other data-driven applications [7, 35]. Their practicality has led 75% of the Fortune 100 and all of North America's top 20 banks to adopt the currently most popular GDBMS, Neo4j [11, 49].

GDBMSs are rapidly evolving and complex systems (*e.g.*, Neo4j has 1.2M LOC), where bugs are inevitable during their development and maintenance. These bugs are critical as they can corrupt the GDBMSs and lead them to malfunction. The application of error-prone GDBMSs in critical domains may lead to serious consequences if a malicious attack were to abuse a previously unknown GDBMS bug. For example, an attacker can leverage bugs in GDBMSs to disclose confidential data [8] or run a Denial of Service (DoS) attack by repeatedly crashing the GDBMS server [9].

To improve GDBMS reliability, several testing approaches [17, 18, 24, 56] have been proposed to find bugs by generating queries in Cypher, the most widely adopted graph query language [36]. As GDBMSs are large-scale, complex systems, effectively testing GDBMSs is difficult and hindered by two key challenges, *query generation* and *test-oracle construction*. Specifically, to cover a broader spectrum of functionalities and deeper query-processing logic in GDBMSs, testing approaches need to generate complex queries that involve various Cypher language features and complicated data dependencies. To accurately identify bugs (*e.g.*, logic bugs) triggered by the generated queries, testing approaches need suitable test oracles to validate the correctness of query execution. Query generation and test-oracle construction are orthogonal and equally important for effective GDBMS testing.

However, a significant gap exists in the query generation of existing GDBMS testing approaches. While all approaches focus on integrating test oracles to identify bugs triggered by the generated queries, none systematically models the Cypher query language to improve query generation. As a result, these approaches are limited in generating valid complex queries, *i.e.*, queries involving various clause combinations and complicated data dependencies. Generating such complex queries is challenging because the clauses invoked by Cypher queries can change the graph states of the manipulated graph database. Such state changes are visible in the subsequent clauses. Moreover, the data and variables used by queries have different scopes, depending on the clauses they are involved in. Lacking systematic modeling of these Cypher language features, existing approaches cannot effectively generate complex queries, and thus many critical functionalities and deep logic of GDBMSs are not exercised by the testing campaigns of these approaches. Therefore, many deep bugs (*e.g.*, the bug shown in Figure 3) in GDBMSs are missed by existing approaches.

To generate complex queries for testing GDBMSs, we need to systematically model the Cypher query language. Cypher is a declarative language that allows expressive data querying without describing specific control flows to achieve it, which enables GDBMSs to flexibly select efficient strategies to execute queries. However, different from typical declarative query languages (*e.g.*, SQL), Cypher queries can make changes to the graph database state while executing clauses of the queries, whose effects are visible in subsequent clauses [10]. For example, a graph node created in an outer `CREATE` clause can be referenced at other clauses (*e.g.*, `DELETE` and `MERGE`) later on in the same query. Such characteristics make it challenging to

[*]Both authors contributed equally to this research.

effectively test GDBMSs. On one hand, neglecting these characteristics tends to make the generated queries simpler, resulting in the GDBMSs not having to handle complicated data dependencies. Therefore, the testing campaign cannot reach the deep logic of GDBMSs. On the other hand, failing to correctly handle such characteristics can make the generated Cypher queries invalid, resulting in many queries being discarded by GDBMSs in early stages (*e.g*, query parsing). For example, the generated queries become invalid when they reference items that are nonexistent or out-of-scope.

In this paper, we propose, to the best of our knowledge, the first systematic approach to effectively generate complex Cypher queries. Orthogonal to the test-oracle construction (*e.g.*, oracle for detecting logic bugs in GDBMSs), our approach fills the gap in the query generation of GDBMS testing research. The approach models graph states and possible state changes caused by the Cypher queries. It abstracts the states into two categories, *query context* and *graph schemas*. Query context contains information related to temporary variables declared in the queries (*e.g.*, the type and scope of each variable), while graph schemas maintain the current graph information, including the graph labels and properties. As query context and graph schemas may change at different Cypher clauses, we must update these abstractions while generating Cypher queries. To this end, we propose *state-aware query generation*. Instead of determining query skeletons for later expressions complementary, our approach incrementally constructs clauses for the generated queries. When constructing a new clause, our approach references only the elements that are accessible with respect to the current query context and graph schema. After the clause is completed, the approach accordingly updates the state information. In this on-the-fly way, our approach can accurately maintain the dynamically evolving graph state and thus effectively generate queries involving complicated data dependencies and state changes.

Based on our approach, we implemented Dinkel, a fully automatic GDBMS testing framework. As Dinkel is designed to improve query generation, it does not integrate advanced test oracles for finding logic bugs, which is orthogonal to our research. Instead, Dinkel employs only ASan alarms, error messages, and code assertions to identify bugs but already outperforms the state-of-the-art. We have evaluated our tool on three popular GDBMSs (*i.e.*, Neo4j [34], RedisGraph [38], and Apache AGE [3]). In the evaluation, Dinkel efficiently generated complex Cypher queries and kept a high validity rate (93.43%). Using these queries, Dinkel found 60 unique, previously unknown bugs. So far, 58 of these bugs have been confirmed, and 51 fixed. Many bugs are long-latent and missed by all existing approaches in their extensive evaluation. Compared to existing approaches, Dinkel can cover over 60% more code and find more bugs within the 48-hour testing campaign, owing to the complex Cypher queries generated by Dinkel. These results demonstrate the effectiveness of Dinkel in generating complex and valid Cypher queries and finding bugs in real-world GDBMSs.

In summary, we make the following technical contributions:

- ***Novel approach***: We abstract the state information of Cypher queries into *query context* and *graph schema*. Based on the abstraction, we propose *state-aware query generation*, which generates Cypher queries by incrementally constructing Cypher clauses and continuously updating the abstracted states.

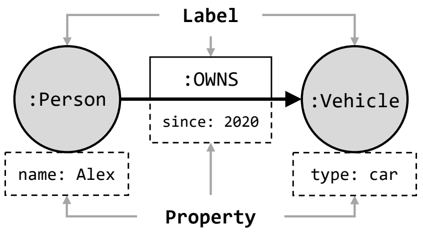


Figure 1: A property graph modeling a person named Alex owning a car.

```
CREATE (p:Person {Name: 'Alex'})
       -[:OWNS {since: 2020}]->
       (v:Vehicle {type: 'car'});
```

Figure 2: A Cypher query creating the graph in Figure 1.

- ***Practical realization***: Based on our approach, we realize a fully automatic and practical testing framework, Dinkel, to find bugs in GDBMSs by generating complex Cypher queries.
- ***Promising results***: We evaluated Dinkel on three popular open-source GDBMSs, namely Neo4j, RedisGraph, and Apache AGE. Dinkel found 60 unique and previously unknown bugs, among which 58 have been confirmed and 51 have been fixed, demonstrating its effectiveness.

2 BACKGROUND AND MOTIVATION

Graph Database Management Systems. Graph database management systems (GDBMSs) utilize graph models to represent the stored data. The most widely used graph model in GDBMSs is the *labeled property graph model* [18], which stores interconnected data using nodes connected via relationships (*i.e.*, directed edges between nodes) [36]. Nodes and relationships, so-called *graph entities*, can be associated with labels and properties. Labels are used to group and classify elements, whereas properties are made up of key-value pairs, providing attribute information of elements.

Figure 1 shows a simple example represented in a labeled property graph. This example shows a person named Alex owning a vehicle of the type car since 2020. The node on the left-hand side has a label `Person` and a property `Name` with a value `Alex`. This node is connected to another node via a direct edge, *i.e.* relationship, whose label is `OWNS`. In this relationship, a property with a key `since` and a value `2020` is held. The node on the right-hand side has attached the label `Vehicle` and contains a property named `type` with value `car`.

Cypher Query Language. Cypher is the most widely adopted query language for property graph databases [36] and was proposed by Neo4j, the most popular GDBMS [11]. It is designed as a declarative query language, which allows users to specify the required data without realizing the detailed procedures. The general way to specify data in Cypher is to concretize the *graph patterns*, which follow the format `(n)-[r]->(m)` and can be used to reference graph entities that satisfy specified conditions. For example, Figure 2 shows a Cypher query that creates the graph in Figure 1. The query concretizes a graph pattern following the `CREATE` clause to specify the graph entities to be created.

Table 1: Grammar of Cypher query language

query	::=	*clause* [*query*]
clause	::=	*reading_clause* | *writing_clause* | *reading/writing_clause* | *projecting_clause* | ...
reading_clause	::=	['OPTIONAL'] 'MATCH' *pattern* ['WHERE' *expression*]
pattern	::=	*node* [*relationship pattern*]
node	::=	'(' *label* properties?* ')'
relationship	::=	'<-[' *label properties?* ']-' | '-[' *label properties?* ']->'
label	::=	':' *identifier*
properties	::=	'{' *identifier* : *expression* [, *identifier* : *expression*] '}'
expression	::=	*identifier* | *constant* | *operation* | *function* | ...
writing_clause	::=	*create_clause* | *delete_clause* | *set_clause* | *remove_clause* | *foreach_clause* | ...
reading/writing_clause	::=	*merge_clause* | *call_clause* | ...
projecting_clause	::=	*return_clause* | *with_clause* | *unwind_clause* | ...

```
MERGE (x)<-[:A]-(x)<-[:A]-(x)<-[y:A]->(x)
DELETE y
CREATE (x)<-[:B]-()
DELETE y;
```

Figure 3: A Cypher query triggering an assertion ("i < n") failure in RedisGraph.

Different from another declarative query language, SQL, Cypher does not differentiate between data declaration (DDL), manipulation (DML), and query (DQL) languages. Instead, a Cypher query can create, read, and modify data in a single statement, thereby allowing statements of procedural nature and non-trivial query state manipulation. Generally, Cypher queries access or operate on graphs via clauses, and each query can contain multiple clauses.

Table 1 shows the context-free grammar of Cypher query language. A Cypher query consists of a sequence of clauses. Each clause can be either a reading clause, writing clause, reading/writing clause, projecting clause, or others (*e.g.*, system configuration clause) [10]. Reading clauses (*i.e.*, `MATCH`) query the GDBMSs to extract information without modifying the graph entities. Writing clauses (*e.g.*, `CREATE`, `DELETE`) modify the data stored in GDBMSs by changing the nodes or relationships in the graph. Reading/writing clauses (*e.g.*, `MERGE`) can both read and write data in the graph. Projecting clauses (*e.g.*, `WITH`, `RETURN`) define expressions to be referenced in the subsequent clauses or the result set. When executing a query with multiple clauses, GDBMSs process these clauses sequentially, and the graph state may change after each clause is processed.

Figure 3 shows a Cypher query containing multiple clauses. This query triggers an assertion failure in RedisGraph. Figure 4 shows the corresponding graph state changes of Figure 3 after each clause is executed by RedisGraph. The first clause is a `MERGE` clause that will create graph entities following the specified graph pattern if no graph entity matches the pattern. After this clause is executed, a node `x` and three relationships with labels `A` are created. Each relationship is from node `x` to node `x`. The subsequent `DELETE` checks whether the graph entities corresponding to `y` exist and deletes the existing ones. This clause deletes a relationship. The second `CREATE` clause creates a new node connected to node `x` with a new relationship whose label is `B`. The last clause, `DELETE`, tries to delete relationship `y`, but as `y` has already been deleted and is non-existing, this `DELETE` clause should do nothing. However, RedisGraph

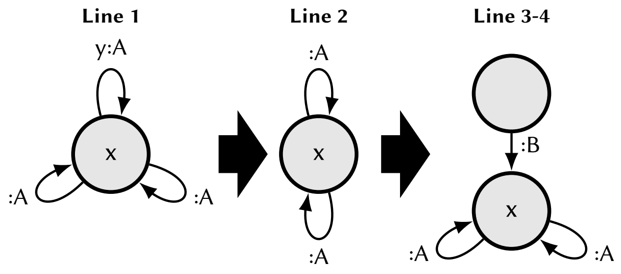


Figure 4: State changes of the query in Figure 3.

mistakenly recognized `y` as non-deleted and performed a deletion on a non-existing relationship, which triggered an assertion failure. Such a mistake is caused by the entity ID reuse mechanisms of RedisGraph, which reassigns the ID of the deleted relationship `y` to the new relationship created by the `CREATE` clause. When referencing `y` in the last `DELETE` clause, RedisGraph mistakenly considers `y` to still exist because its entity ID is being used.

Cypher queries with multiple clauses changing graph states are commonly used in the real world due to the complex nature of data relationships within graph databases. However, automatically generating such queries (*e.g.*, the query in Figure 3) poses challenges, because (1) the generation needs to be aware of the graph state changes caused by each clause (*e.g.*, `MERGE` and `DELETE`); and (2) the intermediate data (*e.g.*, variables `x` and `y`) for concretizing these state changes need to be correctly referenced at proper clauses.

Limitation of Existing Approaches. Several approaches have been proposed to test GDBMSs using Cypher [17, 18, 24, 30, 56]. All of them focus on test oracles to identify bugs triggered by generated queries. They employ only simple template-based query generation, without systematically considering the state changes caused by Cypher clauses. Therefore, their generated queries tend to be structurally simple (*i.e.*, containing fewer clauses) and semantically straightforward (*i.e.*, involving fewer data dependencies). One of the approaches, GDsmith [17], integrates techniques to improve the generation of `MATCH` clauses. It first generates proper graph patterns and `WHERE` conditions in the `MATCH`. Then, GDsmith builds a query skeleton according to its grammar template and fills its generated data components (*i.e.*, graph patterns and `WHERE` conditions). This approach is limited because (1) its query generation is customized

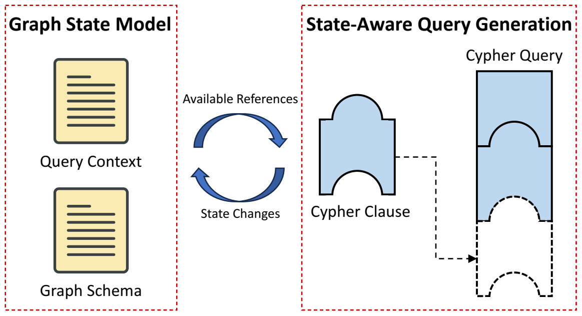


Figure 5: Approach overview.

for only `MATCH` clause and cannot be scaled to other clauses; and (2) it still involves no insight for handling the graph state changes caused by Cypher clauses. Therefore, similar to existing approaches, GDsmith is still limited in generating complex queries containing multiple clauses and complicated data dependencies.

Without a systematic model for handling the graph state changes, existing approaches are limited in generating complex Cypher queries. However, many bugs are hidden in the deep logic of DBMS implementations, which may only be covered by complex queries [20, 21]. To improve the reliability and security of GDBMSs, proposing an effective approach for generating complex Cypher queries has become imperative and critical for GDBMS testing research.

3 STATE-AWARE GDBMS TESTING

In this paper, we propose a novel, state-aware approach for testing GDBMSs using complex Cypher queries. Figure 5 shows the approach overview. Our approach introduces two abstractions, *query context* and *graph schema*, to precisely model the graph states maintained by GDBMSs during query processing. Specifically, query context describes the temporary variables declared at each clause along with the corresponding scope and type. Graph schema stores available graph labels and properties at each clause. To generate complex queries that involve multiple clauses, we need to accurately handle the possible state changes caused by these clauses. We propose *state-aware query generation*, which incrementally constructs Cypher queries clause by clause. Each time a new clause is being constructed, our approach checks the current query context and graph schema, and builds the clause with the available data references. After the clause is constructed, the approach updates the state information accordingly. In this way, our approach can continuously construct Cypher clauses for generating complex queries while accurately maintaining the graph state information for involving complicated data dependencies in the generated queries.

3.1 Graph State Modeling

Cypher query execution can be affected by two kinds of graph state, which we refer to as query context and graph schema, respectively.

Query Context. One kind of graph state information is the temporary variables declared in the query. These variables have specific types and scopes, affecting only the query where they are declared. They can be either concrete values with specific types (*e.g.*, `INTEGER` 0) or aliased to specific nodes or relationships (*e.g.*, node `x` and relationships `y` in Figure 3). Such variables can be referenced only after they are defined and within their scope. We refer to these variables, their types, and their scope information as *query context*. Query context may change at different Cypher clauses. Specifically, query context includes new information when a new variable is declared by a clause, and excludes outdated information when the clause is out of the scope of existing variables.

```
1  CALL {
2    WITH 0 AS x
3    CREATE ()-[:A {n0:x}]->()
4  }
5  MATCH ()-[:(B & C) {n0: 0}]->(:!(D & E))
6  RETURN 0
```

Figure 6: A Cypher query triggering a Neo4j internal error: ExecutionFailed (Cannot invoke "RelationshipDataAccessor.type()" because "this.v3_relationships" is null).

Table 2: State changes of the query in Figure 6

Line	Query Context	Graph Schema	
		Label	Property
1	{}	{}	{}
2	{x, INT, CALL}	{}	{}
3	{x, INT, CALL}	{A}	{n0}
4	{}	{A}	{n0}
5	{}	{A}	{n0}
6	{}	{A}	{n0}

Graph Schema. Another graph state information is the schema of the stored graph data. It includes graph labels and properties (*e.g.*, label `A` and `B` for relationships in Figure 3). We refer to such information as *graph schema*. Graph schemas can be changed by specific Cypher clauses. For example, `CREATE` clauses can create nodes or relationships with new labels and properties, while `REMOVE` clauses can remove existing labels or properties.

Different from query context, whose effects are limited by their scope, the operations (e.g., `CREATE` clause that declares new labels) on graph schema affect the manipulated database permanently. In addition, graph schema can be referenced even though the labels and properties are non-existent in the manipulated graph. This design improves query flexibility as users can write valid queries without concerning the current graph schema.

Example. To better understand query context and graph schema, we discuss the Cypher query shown in Figure 6, which causes an internal error in the enterprise version of Neo4j, which is close-source. This query involves complex graph state changes by using multiple Cypher clauses. Table 2 shows the concrete state at each line of the query. At the first line of the query, query context and graph schema contain no state information as the `CALL` clause has not had any effects yet. At the second line, a `WITH` clause declares a temporary variable with value 0, and thus the query context now contains the corresponding variable identifier (*i.e.*, `x`), its type, and its scope. Because the `WITH` clause is inside the `CALL` clause, the scope of the variable `x` is limited to the `CALL` clause. At the third line, the `CREATE` clause updates the graph schema by adding a new relationship with label `A` and properties `n0` referencing variable `x`.

Algorithm 1: State-Aware Query Generation

```
   Output: query
1  Function GenQuery():
2      query ← EmptyQuery();
3      qc ← {}; // query context
4      gs ← {}; // graph schema
5      do
           // clause can update qc and gs
6          clause, qc, gs ← GenClause(qc, gs, ANY);
7          AppendQuery(query, clause);
8      while rand() < P and Length(query) < $L_{max}$;
9      return query;
10 Function GenClause(qc, gs, type):
11     if type = ANY then
12         type = RandClauseType();
       // initialize the clause according to its type
13     clause, qc, gs ← RandInitClause(qc, gs, type);
14     foreach subclause, subtype in clause do
15         subclause, qc, gs ← GenClause(qc, gs, subtype);
       // clean out-of-scope query context
16     CleanQC(qc);
17     return clause, qc, gs;
```

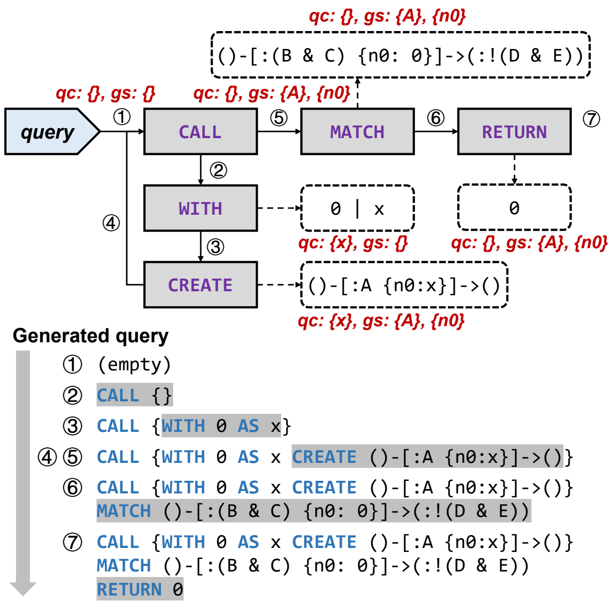
The new graph schema (*i.e.*, label `A` and property `n0`) is not limited by the scope of the `CALL` clause and still exists after the `CALL` clause and even when the query finishes. At the fifth line, the `MATCH` clause references the property `n0` and four non-existing labels `B`, `C`, `D`, and `E`. The query ends with a `RETURN` clause.

Generating such a query is challenging because it involves multiple clauses (5 clauses) and graph data references (*e.g*, the `CREATE` references variable `x`, and the `MATCH` references property `n0`). Utilizing the two abstractions, query context and graph schema, our approach can explicitly and precisely describe the graph state maintained by GDBMSs during query processing, which enables our approach to accurately reference available graph data in the generated queries.

3.2 State-Aware Query Generation

Insight. Our state-aware query generation is based on an observation that both query context and graph schema are updated only when Cypher clauses are invoked (*e.g.*, `CREATE` and `MATCH` in Figure 6), or exit specific clause contexts (*e.g.*, `CALL` in Figure 6). Utilizing this observation, our approach can accurately track the state changes inside the Cypher queries by analyzing the possible impact caused by specific clauses in the queries.

Algorithm. Algorithm 1 shows the procedure of our state-aware query generation. Our approach does not need any input. Initially, it constructs an empty query that contains no clause, and initializes query context *qc* and graph schema *gs* to empty sets (line 2-4). Then, the approach incrementally appends the query with a newly generated clause (line 5-8). Each time a new clause is generated, the query context and graph schema may be updated (line 6). Our approach decides whether to append the query with more clauses based on a certain probability P. If the length of the query reaches



```
① (empty)
② CALL {}
③ CALL {WITH 0 AS x}
④⑤ CALL {WITH 0 AS x CREATE ()-[:A {n0:x}]->()}
⑥ CALL {WITH 0 AS x CREATE ()-[:A {n0:x}]->()}
   MATCH ()-[:(B & C) {n0: 0}]->(:!(D & E))
⑦ CALL {WITH 0 AS x CREATE ()-[:A {n0:x}]->()}
   MATCH ()-[:(B & C) {n0: 0}]->(:!(D & E))
   RETURN 0
```

Figure 7: Generation process for the query in Figure 6.

the limit L_{max}, the approach stops appending the query (line 8). In the end, the generated query is returned (line 9).

To generate a clause, our approach needs to reference the query context and graph schema. If the clause type is not specified, the approach randomly chooses a clause type (line 11-12). According to the elements in query context, graph schema, and the specified clause type, our approach randomly initializes a clause (line 13). Initialization for different clauses can vary (*e.g.*, the procedures for `CALL` and `CREATE` are different), but it generally includes the procedure of determining the number and types of the components needed by the clause, generating corresponding expressions for the components, and analyzing the impacts of the generated clause. If the generated clause contains subclauses, the approach will recursively call `GenClause()` to generate subclauses with specified types (line 14-15). After the clause is generated, the approach cleans up the query context if necessary (line 16), such as removing out-of-scope variables at the end of a `CALL` clause. In the end, the generated clause, the updated query context, and the graph schema are returned (line 17).

Example. Figure 7 shows how our approach generates the query in Figure 6. Initially, the query is empty, and query context and graph schema contain nothing (step ①). The approach randomly determines that the first clause to be generated is a `CALL` clause (step ②). When initializing the `CALL` clause, our approach determines its components, which are two subclauses. Then, these two subclauses are recursively generated. The first subclause is randomly constructed as a `WITH` clause with two components, `0` and `x`. After the `WITH` subclause is completed, the approach updates the query context by adding a temporary variable `x` with its type and scope, according to the functionality of `WITH` [10] (step ③). The approach continues to generate the second subclause, which is randomly determined as a `CREATE` clause. The component of the `CREATE` clause is generated with graph pattern `()-[:A {n0:x}]->()`, which references the existing variable `x`. After the `CREATE` is constructed, the approach updates the graph schema with the newly generated label `A` and property `n0` (step

Table 3: Effects of Cypher clauses on graph states

Clause	Query Context	Graph Schema	
		Label	Property
`MATCH`	+	/	/
`CREATE`	+	+	+
`MERGE`	+	+	+
`DELETE`	/	/	/
`REMOVE`	/	-	-
`SET`	/	+/-	+/-
`UNWIND`	+	/	/
`WITH`	+	/	/
`RETURN`	+	/	/
`FOREACH`	+/-	+/-	+/-
`CALL`	+/-	+/-	+/-
`UNION`	+/-	+/-	+/-
`EXISTS`	+/-	+/-	+/-
`COUNT`	+/-	+/-	+/-

/: the clause has no effect on the graph state;
+: the clause can add elements to the graph state;
-: the clause can remove elements from the graph state;
+/-: the clause can both add and remove elements.

④). After the two subclauses are constructed, the `CALL` generation is completed. Before generating other clauses, our approach cleans the variable `x` in the query context, because it is scoped to only the `CALL` clause (step ⑤). Then, our approach continues to generate a `MATCH` clause, which references the existing property `n0`, according to the latest graph schema, and four non-existing and randomly constructed labels `B`, `C`, `D`, and `E` (step ⑥). In the end, our approach generates a `RETURN` clause to finish the query (step ⑦).

Clause Impact Analysis. Different Cypher clauses have different effects on query context and graph schema, and our approach models their effects and supports the clause generation accordingly (*e.g.*, `RandInitClause` function in Algorithm 1). Table 3 lists the effects of different clauses, according to the openCypher specification [10]. Most clauses can modify either query context, graph schema, or both. The `DELETE` clause does not affect either query context or graph schemas, because it manipulates only data but does not affect available references or graph attributes. For example, in Figure 3, variable `y` can still be referenced despite being operated on by a subsequent `DELETE` clause. Five clauses (*i.e.*, `FOREACH`, `CALL`, `UNION`, `EXISTS`, and `COUNT`) can involve subqueries using subclauses, and they can have effects on both query context and graph schemas via invoking specific subclauses. In addition, these clauses demarcate the scope of the query contexts defined by the subclauses. For example, the `CALL` clause in Figure 6 modifies both query context and the graph schema via the `WITH` and `CREATE` clauses, and the variable `x` defined by the `WITH` clause can be referenced only within the `CALL` clause.

4 IMPLEMENTATION

We implement our approach into a practical and fully automatic GDBMS testing framework, Dinkel, integrating Cypher query generation, bug-triggering query reduction, and efficient bug deduplication. Figure 8 shows its architecture. The overall code base of

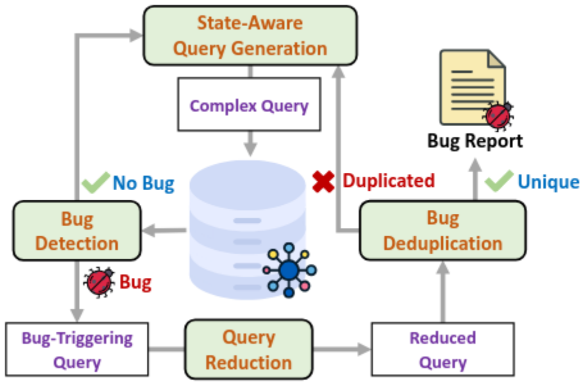


Figure 8: Achitecture of Dinkel.

Dinkel consists of 10k lines of Go code. The source code of Dinkel will be available on GitHub. The following discusses important implementation details.

Supported Clauses in Query Generation. As of writing this paper, Dinkel supports all non-administrative clauses (*e.g.*, all the clauses shown in Table 3) in the tested GDBMSs. For example, in Neo4j, 21 out of the total 25 clauses are non-administrative and thereby supported. Administrative clauses, such as `SHOW FUNCTIONS` and `USE`, are not supported because they are not related to the logic of query processing in GDBMSs. For example, The clause `SHOW FUNCTIONS` is designed to list all available functions and cannot be combined with other clauses. The clause `USE` is used to switch the manipulated databases, which is not necessary as all databases are initialized to empty and they function in the same way.

Compared to existing approaches [17, 18, 24, 30, 56], Dinkel supports more Cypher clauses, as shown in Table 4. None of the existing approaches supports `FOREACH` clauses, because this clause complicates the Cypher control flows and can significantly change graph states during query execution. `UNION`, `EXISTS`, and `COUNT` are not supported as well, as these clauses invoke subqueries that can inherit and change the graph states of the main query. Lacking a systematic model for handling graph state changes, existing approaches are limited in supporting various Cypher clauses.

Bug Detection. In this paper, we focus on the query generation of GDBMS testing and apply only general methods for identifying triggered bugs. Specifically, for each executed query, Dinkel checks the messages returned from the GDBMS. If the message indicates an internal error in the GDBMS (*e.g.*, the error in Figure 6), a bug is identified. We also enable all the assertions embedded in the tested GDBMSs to catch assertion failures. In addition, we enable ASan [1] for the GDBMSs developed in C/C++ to identify memory bugs triggered by our generated queries.

Query Reduction. No existing tool is available for reducing Cypher queries, which makes it difficult for developers to minimize bug-triggering queries and investigate bugs. To address this problem, we implemented Dinkel with an automatic query reduction method. The core idea of this method is to reduce queries clause by clause. For each Cypher clause, Dinkel first tries to delete it. If the query without the clause still triggers the bug, the clause will be permanently removed. Otherwise, the deleted clause will be recovered,

Table 4: Cypher clauses supported by existing approaches

Clause	GDsmith	GDBMeter	GraphGenie	GAMERA	Dinkel
`MATCH`	●	●	●	●	●
`CREATE`	●	●	●	●	●
`MERGE`	○	○	○	○	●
`DELETE`	○	●	○	○	●
`REMOVE`	○	●	○	○	●
`SET`	○	●	○	○	●
`UNWIND`	●	○	●	●	●
`WITH`	●	○	●	●	●
`RETURN`	●	●	●	●	●
`CALL`	○	○	●	○	●
`FOREACH`	○	○	○	○	●
`UNION`	○	○	○	○	●
`EXISTS`	○	○	○	○	●
`COUNT`	○	○	○	○	●

and Dinkel goes on to try to replace the clause with an alternative clause if possible. For example, Dinkel can try to replace the `CALL` clause in Figure 6 with its subclause `CREATE`. If some clauses are successfully deleted or replaced in one try, Dinkel will restart the reduction process for the reduced queries. The process stops when no clause in the query can be further reduced. With this method, bug-triggering queries can be efficiently reduced.

Bug Deduplication. When a query generated by Dinkel triggers a bug, we need to demonstrate whether this bug is unique or a duplicate of another bug already found by Dinkel. A feasible way is bisection [15, 52], which finds the earliest bug-inducing commits for each triggered bug and deduplicates bugs by checking whether their bug-inducing commits are the same. As GDBMSs are large-scale system software, compiling and building a version of a GDBMS demands considerable time and resources (*e.g.*, 8 minutes for Neo4j in our evaluation), which can make bisection infeasible when a large number of commits need to be bisected (*e.g.*, 79K commits in Neo4j's GitHub repository [34]). To improve the efficiency of bisections, Dinkel leverages containerization techniques (*e.g.*, Docker [12]) to cache and manage the built GDBMS versions. Given a batch of bugs needed to be deduplicated, Dinkel sets up multiple containers to bisect the bug-inducing commits. During bisection for a bug, Dinkel needs to test many commits by checking whether the bug can be reproduced in this commit version. For each commit that has not been tested yet, Dinkel compiles and builds a GDBMS version, and caches the built version in a container image. When testing a cached commit during bisection for a new bug, Dinkel directly utilizes the image to set up the container, without repeatedly compiling and building the same versions of GDBMSs.

5 EVALUATION

To demonstrate the effectiveness of Dinkel in GDBMS bug detection, we evaluate it on real-world GDBMSs and seek to answer the following questions:

Q1 Can Dinkel find real bugs in widely-used and extensively-tested GDBMSs? (Section 5.2)

Table 5: Status of the bugs found by Dinkel

GDBMS	Reported	Confirmed	Fixed
Neo4j	32	32	32
RedisGraph	17	17	10
Apache AGE	11	9	9
Total	60	58	51

Q2 How complex and valid are the queries generated by Dinkel? (Section 5.3)

Q3 How do query context and graph schema contribute to the effectiveness of Dinkel? (Section 5.4)

Q4 Can Dinkel outperform state-of-the-art GDBMS testing approaches? (Section 5.5)

5.1 Experimental Setup

We evaluated Dinkel on three real-world GDBMSs, Neo4j [34], RedisGraph [38], and Apache AGE [3]. We chose them because they are popular and open-source GDBMSs supporting Cypher. Neo4j is the most popular GDBMS, according to DB-Engines Ranking [11], and has been extensively tested by existing approaches [17, 18, 24, 56]. RedisGraph was widely adopted by Redis (the 6th most popular DBMS [11]) to provide functionalities of graph data querying and storage. Its fork, FalkorDB [13], is also actively developed. Apache AGE is a widely-used extension for PostgreSQL (the 4th most popular DBMS [11]) that enables applications to manipulate graph data on the top of a relational DBMS.

We evaluate Dinkel on these GDBMSs with the latest versions. During our testing, if the code of the tested GDBMSs is updated (*e.g.*, a new version is released), we set up new Dinkel instances to test the updated versions. Specifically, we test Neo4j starting from version 5.6.0, RedisGraph starting from version 2.12.0, and ApacheAGE starting from version 1.3.0. We clone the code of these GDBMSs from their official GitHub repositories. To demonstrate the effectiveness of Dinkel on testing black-box GDBMSs, we also evaluate Dinkel on the Enterprise version of Neo4j from 5.6.0. Each time we implement new features on Dinkel, we stop and restart the testing. Overall, the testing process lasts three months. The evaluation was performed on Ubuntu 20.04 with a 64-core AMD EPYC 7742 processor running at 2.25GHz and 256GB of RAM.

5.2 Bug Detection

Table 5 shows the status of the bugs found by Dinkel. In total, Dinkel found 60 unique bugs, including 32 bugs in Neo4j, 17 in RedisGraph, and 11 in Apache AGE. Among these bugs, 58 are confirmed, and 51 are fixed. None of the 60 bugs are duplicates.

Bug Classification. We classify the bugs found by Dinkel into two categories according to their manifestation:

- *Internal errors.* The tested GDBMSs unexpectedly throw exceptions or errors when processing syntactically and semantically valid queries. The error messages can indicate the inconsistency of the internal execution status (*e.g.*, the error triggered by the query shown in Figure 6).

Table 6: Classifying the found bugs

GDBMS	Internal Error	Crash
Neo4j	32	0
RedisGraph	3	14
Apache AGE	8	3
Total	43	17

Table 7: GDBMS components affected by bugs

GDBMS	Parser	Planner	Executor
Neo4j	17	11	6
RedisGraph	7	4	4
Apache AGE	8	0	1
Total	32	15	11

- *Crashes.* The queries cause GDBMSs to crash due to assertion failures or memory corruption, *e.g.*, a null-pointer dereference.

Note that Neo4j is realized in Java and implements exception handling for unexpected errors (*e.g.*, null dereferences in Figure 6 and out-of-bound accesses in Figure 16), and thus Neo4j does not crash but posts exception information when triggering memory corruption. Therefore, we classify all the bugs Dinkel found in Neo4j as internal errors. Table 6 shows the classification results. Among the 60 Dinkel found, 43 cause internal errors, 32 of which are Neo4j bugs. Dinkel can find 17 bugs that crash RedisGraph and Apache AGE, which are implemented in C/C++. Among these 17 bugs, 8 are caused by memory corruptions, and 9 are caused by assertion failures. These results demonstrate that Dinkel is effective in finding bugs in GDBMSs.

Affected GDBMS Components. We investigated 42 of the 51 fixed bugs, where we can analyze the fix patches to identify the GDBMS components affected by the bugs accurately. The 9 fixed bugs in the Enterprise version of Neo4j are not included because developers keep the fix patches confidential. Table 7 shows the results. Among the 42 fixed bugs, 32 affect the parsers of GDBMSs, 15 affect the planners, and 10 affect the executor. Interestingly, 9 bugs can affect two components, and 4 can affect all the components. For example, the bug shown in Figure 16 can affect both the parser and planner of Neo4j. In total, 19 bugs can affect either the planner, the executor, or both. These results demonstrate that (1) Dinkel can find bugs in various GDBMS components; and (2) Dinkel can find bugs in the deep logic of GDBMS implementation, considering 45% (19/42) of bugs are related to the planners or executors.

Diversity of Cypher Features. We investigate the queries triggering the 60 bugs and analyze the distribution of the Cypher features used in these queries. All these queries are reduced. The results are shown in Figure 9. The `RETURN` keyword is present in most bug-triggering queries, appearing in 64% of them. This is unsurprising, as `RETURN` is required to appear at the end of a query containing only reading clauses (*e.g.*, `MATCH`), otherwise, the query would be syntactically incorrect. `MERGE` (33%) and `CREATE` (26%) are the two clauses inducing the creation of graph properties (*e.g.*, providing available

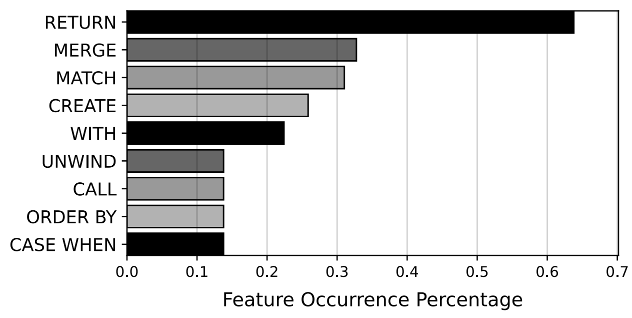


Figure 9: Feature distribution of bug-triggering queries.

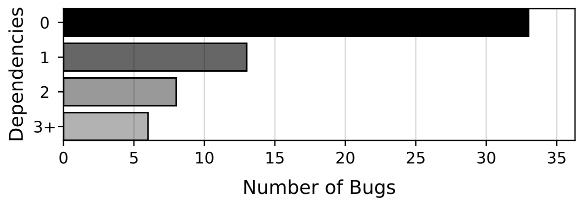
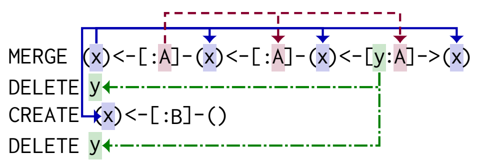


Figure 10: Data dependencies of bug-triggering queries.



```
MERGE (x)<-[:A]-(x)<-[:A]-(x)<-[y:A]->(x)
DELETE y
CREATE (x)<-[:B]-()
DELETE y
```

Figure 11: Data dependencies within the query in Figure 3.

elements for the operations of subsequent clauses). We can deduce from this that graph data is often required to be present for bugs to trigger. The remaining keywords can induce complicated data flow (*e.g.*, `WITH` (22%) and `UNWIND` (14%)) or control flow (*e.g.*, `CALL` (14%) and `CASE WHEN` (14%)) in the query, indicating that bugs often appear when the database system has to handle such flows. `ORDER BY` also appears frequently (14%), which forces GDBMSs to keep track of the manipulated graph data and perform non-trivial comparisons to ensure correct order.

Data Dependencies for Triggering Bugs. To convey the query complexity Dinkel required to trigger bugs, we analyze the complexity of the 60 bug-triggering queries. The results are shown in Figure 10. Nearly half of the bugs (45%) require at least one data dependency. Specifically, 13 bug-triggering queries contain one data dependency, 8 contain two, and 6 contain three or more. For example, in Figure 3, the bug-triggering query contains 8 data dependencies, as illustrated in Figure 11. Among the 8 dependencies, 6 are involved in query context (*i.e.*, the variables `x` and `y` assigned to a node and a relationship), and 2 are involved in graph schema (*i.e.*, the label `A`). These results demonstrate that some GDBMS bugs can be triggered only when the queries contain multiple data dependencies, and Dinkel can effectively find these bugs.

Size of Bug-Triggering Queries. Figure 12 shows the size of the 60 bug-triggering queries. 54 bugs can be triggered by queries whose size is less than 100 bytes. The bug-triggering queries shown in Figure 3 and Figure 6 are examples of such queries. As query

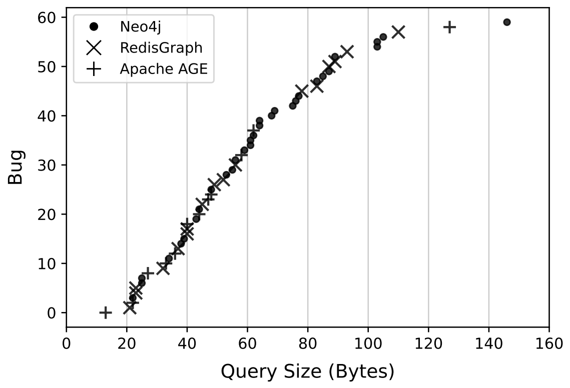


Figure 12: Query sizes of bug-triggering queries.

Table 8: Queries generated by Dinkel within 24 hours

GDBMS	Gen.	Valid	Clause	Dep.	Size
Neo4j	68K	89.88%	23.68	30.16	1202.39
RedisGraph	956K	98.14%	23.42	30.15	992.41
Apache AGE	367K	81.79%	16.94	15.60	830.35
Total	1391k	93.43%	21.72	26.31	959.85

Gen.: the number of generated queries;
Valid: the number of valid queries among the generated ones;
Clause, **Dep.**, and **Size**: the average number of clauses, data dependencies, and bytes over all generated queries, respectively.

size increases from 0 to 100 bytes, the number of triggered bugs increases almost linearly. The bug-triggering query with the largest size is 146 bytes and is shown in Figure 16. This query triggers an internal error caused by an out-of-bounds access in Neo4j.

Bug Importance. Neo4j, the GDBMS we focus on, provides Community and Enterprise versions. The bugs in the Enterprise version are critical because this version is commonly deployed on commercial applications. Among the 32 Neo4j bugs Dinkel found, 19 bugs can be triggered in both Enterprise and Community versions, and 8 bugs can be triggered in only the Enterprise version. RedisGraph and Apache AGE did not provide severity information about the reported bugs, but we noticed that most of the bugs were fixed, which indicates that the bugs are non-trivial.

Some developers express their appreciation for our effort in finding bugs in their GDBMSs. Particularly, Neo4j provides very positive feedback written in German. We translate it into English:

> My colleague has told me that you've been busy creating GitHub issues and have contributed to improving Neo4j! Thanks a lot for that! In the name of Neo4j, I would like to send you some swag [...]

5.3 Query Generation

To understand the complexity and validity of the queries generated by Dinkel, we ran Dinkel on each tested GDBMS for 24 hours and collected all generated queries. Table 8 shows the statistical results.

```
SELECT * FROM cypher('graph',$$
    CREATE (x) SET x.n0 = (true OR true)
    RETURN 0 AS y, 1 AS z
$$) AS (v agtype);
```

Figure 13: An invalid SQL query that wraps a valid Cypher query, which crashes Apache AGE and PostgreSQL.

Query Complexity. For the 1391K generated queries, Dinkel constructs 30.23M clauses. The average number of clauses for each query is 21.72. Additionally, on average, each query contains 26.31 data dependencies. Because of the large number of query clauses and data dependencies, the queries generated by Dinkel are typically big. The average query size is 959.85 bytes. These results demonstrate that Dinkel can effectively generate complex queries, which are large and contain multiple clauses with complicated data dependencies. The generated queries are more complex than the bug-triggering queries discussed in Section 5.2, because all the bug-triggering queries are reduced, while the generated queries inevitably contain many redundant parts [21, 29, 39]. We find that the queries generated for Apache AGE are less complex (*i.e.*, fewer clauses, fewer data dependencies, and smaller size), because Apache AGE supports fewer clauses, and many clauses that can induce data dependencies are missing. As a result, the relative chance of Dinkel choosing clauses (*e.g.*, `RETURN`) that terminate query generation becomes much higher, and thus the queries generated for Apache AGE exhibit lower complexity.

Query Validity. Among the 1391K Cypher queries generated by Dinkel, 1300K are valid. The percentage of valid queries is 93.43%. The result demonstrates that Dinkel can keep a high validity rate even when generating complex queries. We investigated the invalid queries generated by Dinkel and found that they were mainly caused by illegal arithmetic operations. Some arithmetic operations require their operands to satisfy some constraints (*e.g.*, the divisor in a division operation must be non-zero). However, Dinkel cannot track the value of each operator in queries because the value may depend on complex expressions, whose results are difficult to calculate (*e.g.*, hash functions). Queries failing to satisfy the constraints of operations cause semantic errors like divisions by zero.

We noticed that the validity of queries generated for Apache AGE is lower than that of the other two GDBMSs. This is caused by the fact that Apache AGE is designed to be a PostgreSQL extension and thus needs to interact with the SQL contexts of PostgreSQL. Specifically, the columns returned by Cypher queries of Apache AGE have to match the type definitions in the wrapping SQL queries used by PostgreSQL, otherwise, the SQL queries become invalid. Figure 13 shows an example, where the Cypher query is valid but the overall SQL query is invalid because the Cypher query returns two columns while the SQL query expects only one column. Such invalid cases do not affect bug detection, because the error of SQL queries occurs only after the Cypher queries are fully executed. For example, in Figure 13, even though the SQL query is invalid, it can crash Apache AGE and PostgreSQL, because the internal Cypher query triggers the bug before it returns values to the SQL query.

Throughput. On average, one Dinkel instance generates 5.36 test cases per second. Specifically, at each second, one Dinkel instance

Table 9: Bug-triggering queries related to graph states

Version	Found	QC	GS	Both
Neo4j	32	11	1	2
RedisGraph	17	6	0	2
Apache AGE	11	5	0	0
Total	60	22	1	4

QC: the queries depend on only query context;
GS: the queries depend on only graph schema;
Both: the queries depend on two graph states.

generates 0.78 test cases for Neo4j, 11 test cases for RedisGraph, and 4.25 test cases for Apache AGE. To understand the bottleneck of test throughput, we further investigated the time used by Neo4j in query generation and query execution, and found that 92.79% of CPU time is occupied by Neo4j executing the generated queries, indicating that the testing throughput is determined by the performance of the tested GDBMS. We believe the current throughput is practical considering (1) GDBMS testing typically lasts for several months [18, 24], and thus a sufficient number of test cases can be executed; and (2) setting up multiple Dinkel instances can significantly improve the test efficiency.

5.4 Ablation Analysis

To understand the contribution of query context and graph schema for bug detection in GDBMSs, we analyze the 60 bug-triggering queries by investigating whether the generation for these queries depends on specific graph states. Specifically, we extract the data dependencies contained by each query and check whether at least one of the data dependencies is related to query context or graph schema. Table 9 shows the analysis results.

Among the 60 bug-triggering queries, 22 depend on only query context, 1 depends on only graph schema, and 4 depend on both (*e.g.*, the bug-triggering query shown in Figure 11). According to these results, without query context, 26 bugs cannot be found. This demonstrates that query context is important because it enables Dinkel to generate queries referencing specific nodes, relationships, or expressions constructed in internal clauses. Without graph schema, a further 5 bugs cannot be found. These bug-triggering queries require Dinkel to properly reference the latest graph labels or properties. The following shows two queries involving query context and graph schema, respectively.

Example 1: Query referencing query context. The query shown in Figure 14 triggers an internal error. This query uses a `UNWIND` clause to iterate over the elements in the array `[0]+[]` (*i.e.*, `[0]`). Then, the query invokes `toBoolean` to convert the iterated element to the boolean type and return each converted element. However, when concatenating the arrays `[0]` and `[]`, Neo4j incorrectly maintains the type information of the concatenated array of `[0]` and `[]`. It directly assigns the type of elements in the right-hand-side array (*i.e.*, `[]`), which is implicit and not determined, to elements of the concatenated array. In the end, the type of the elements in the concatenated array is unexpectedly assigned as `Float` (while `Integer` is

```
UNWIND [0]+[] AS i
RETURN toBoolean(i)
```

Figure 14: A query that references query context and triggers a Neo4j internal error—Type mismatch: expected Boolean, Integer or String but was Float (line 1, column 37 (offset: 36))

```
MERGE ()-[:A]->({x:0})
RETURN EXISTS {(:!(B&C))-[{x:0}]->()};
```

Figure 15: A query that references graph schema and triggers a Neo4j internal error—Neo4jError: ExecutionFailed (Cannot invoke "RelationshipDataAccessor.properties(...)" because "relationshipCursor" is null).

the expected type), resulting in the internal error of type mismatching. To fix this bug, when concatenating two arrays, developers make Neo4j consider the element type in both arrays.

Example 2: Query referencing graph schema. In Figure 15, the query specifies a graph pattern `()-[:A]->({x:0})`, where the relationships with type `A` are connecting an arbitrary node to another node with property `x` equal to `0`. The `MERGE` clause tries to find the nodes and relationships satisfying the pattern. If there is no such node and relationship, the `MERGE` clause creates new ones satisfying the patterns. After the `MERGE` clause, the `EXISTS` clause checks whether the graph pattern `(:!(B&C))-[{x:0}]->()` can be matched in the current graph database and returns the corresponding boolean value. The `EXISTS` clause references two non-existing labels, `B` and `C`, and the property `x` created in the `MERGE` clause. As the bug is triggered in the Enterprise version of Neo4j, the root cause and the corresponding fixing patch are kept confidential by Neo4j developers.

5.5 Comparison

Bug Latency Study. To demonstrate that Dinkel can find bugs missed by existing approaches [17, 18, 24, 30, 56], we investigate the latencies of the bugs found by Dinkel. For each bug, we check whether its bug-inducing commit was created before the years when existing approaches were published. If Dinkel finds some long-latent bugs, we can conclude that Dinkel can find bugs missed by existing approaches. This comparison is reasonable and objective because: (1) none of the bugs found by Dinkel are marked as duplicated by GDBMS developers, which means that no approach found these bugs until Dinkel found them; and (2) all existing approaches have extensively tested Neo4j and RedisGraph [17, 18, 24, 30, 56], which means that in these two GDBMSs, no approach found the long-latent bugs found by Dinkel during their evaluation.

GDsmith [17], GDBMeter [24], GraphGenie [18], and GAMERA [56] were published in 2023, and GRev [30] was published in 2024. Therefore, for each bug, we checked whether its bug-inducing commit was created before 2024 (*i.e.*, in 2023 or earlier) and 2023. Table 10 shows the results. Among the 49 bugs Dinkel found in Neo4j and RedisGraph, 33 bugs already existed before 2023, indicating that all the existing approaches proposed after 2023 failed to find these 33 bugs in their extensive evaluation. In addition, 14 bugs were introduced to Neo4j and RedisGraph between 2023 and 2024, while GRev, which was proposed after 2024, cannot find these

Table 10: Latency of the logic bugs found by Dinkel

DBMS	Found	Bug-involved year		Longest latency
		< 2024	< 2023	
Neo4j	32	30	20	2016
RedisGraph	17	17	13	2020
Apache AGE	11	11	7	2021
Total	60	58	40	2016

```
WITH [] AS n0 ORDER BY null
CALL {
  WITH [] AS n1 ORDER BY null
  UNWIND [0] AS x
  UNWIND [x] AS n2
  RETURN n2 AS n3
}
FOREACH (n4 IN null | MERGE ({key:n3}))
```

Figure 16: A query triggering a Neo4j internal error—Neo4j-Error: ExecutionFailed (arraycopy: last destination index 6 out of bounds for object array[5]).

bugs. These results indicate that existing approaches indeed miss many long-latent bugs, while Dinkel can effectively find them.

We analyze the bug-triggering queries of these bugs and conclude the reason why existing approaches missed these bugs as the following: (1) the bug-triggering queries (*e.g.*, the queries shown in Figure 3 and Figure 6) contain data dependencies, which existing approaches lack support for handling; and (2) the bug-triggering queries contain Cypher features (*e.g.*, `CALL` and `FOREACH` in the query shown in Figure 16) that are not supported by existing approaches. The following is an example of the 33 bugs.

Example: Query using data dependencies and advanced Cypher features. Figure 16 shows a bug in Neo4j, which is present in 2022 but none of the existing approaches found it. The bug-triggering query involves complicated data dependencies and advanced Cypher clauses (*e.g.*, `FOREACH` and `UNWIND`). This query constructs an empty array `[]` and invokes a subquery using the `CALL` clause. The subquery also constructs an empty array. It then invokes two `UNWIND` clauses, which iterate over each element in the operated array. For each iterated element, the `UNWIND` executes the subsequent clause under the context of this element. For example, the array `[x]` used by the second `UNWIND` references the variable `x`, which is `0` when the first `UNWIND` iterates over the item `0` in the array `[0]`. After the `CALL`, the query utilizes `FOREACH`, whose execution depends on the query context of the subquery in the `CALL`. To optimize the query execution, Neo4j tries to flatten the `FOREACH` loop. However, such optimization does not work well when the loop involves update operations (*i.e.*, `MERGE`) under complicated contexts (*i.e.*, the contexts produced by `CALL`). The improper optimization corrupts the internal data structures of Neo4j, *Eagers*, which are the production of another optimization, *Eagerness analysis*. In the end, an internal error is triggered when Neo4j tries to access the corrupted Eagers. To fix this bug, Neo4j developers modify both the Eagerness analysis and the flattening strategy for `FOREACH` clauses to ensure they work consistently.

Table 11: Results of comparison

	Neo4j		RedisGraph	
	Covered Line	Bug	Covered Line	Bug
GDsmith	533k	0	15k	0
GDBMeter	283k	1	19k	1
GraphGenie	506k	0	3k	0
GAMERA	283k	1	3k	0
GRev	601k	1	14k	1
Dinkel	737k	11	20k	18

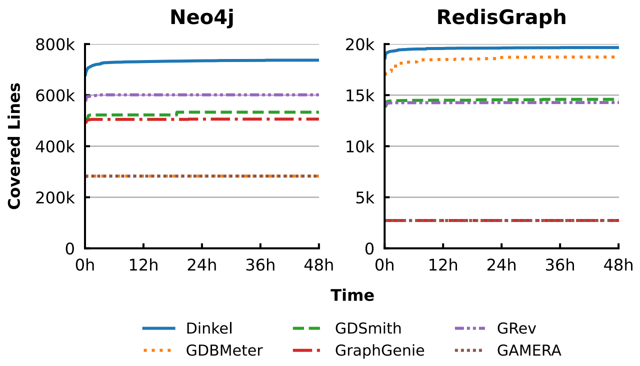


Figure 17: Covered lines of Neo4j and RedisGrpah.

Empirical Comparison. To empirically demonstrate the effectiveness of Dinkel on code coverage and bug detection over the state-of-the-art, we evaluated Dinkel and existing Cypher testing tools (*i.e.*, GDSmith, GDBMeter, GraphGenie, GAMERA, and GRev) on Neo4j and RedisGraph, which are the only two GDBMSs supported by all these tools. Each evaluation persisted for 48 hours. Table 11 shows the comparison results of Dinkel and existing tools.

Code Coverage. As shown in Table 11, on average, Dinkel can cover 67% and 85% more code than existing approaches in Neo4j and RedisGraph, respectively. Among the tools except for Dinkel, GRev covers the most code (*i.e.*, 601k lines) in Neo4j. Compared to GRev, Dinkel covers 23% more code. Benefiting from the powerful capability of generating complex Cypher queries, Dinkel can cover much deeper logic of GDBMSs that is related to processing advanced Cypher features and complicated data dependencies. In contrast, simple queries generated by existing tools have little chance to touch such logic. In RedisGraph, except for Dinkel, GDBMeter covers the most code (*i.e.*, 19k lines). Compared to GDBMeter, Dinkel covers 5% more code. The coverage improvement is less significant than in Neo4j, because RedisGraph supports fewer Cypher features and cannot handle some complicated Cypher semantics. For example, RedisGraph does not support processing the subqueries in `FOREACH` and `CALL` clauses. As a result, the space for Dinkel to improve the code coverage in RedisGraph is limited. Figure 17 shows the trends of the covered code in each GDBMS during testing. Dinkel and all the state-of-the-art cover new codes quickly and saturate in earlier stages. In almost all testing campaigns, Dinkel can cover more code than the existing tools.

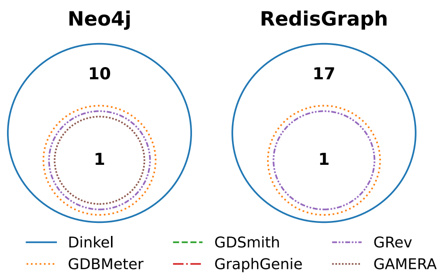


Figure 18: Relation of bugs found by existing tools.

Found Bugs. As shown in Table 11, compared to existing approaches, Dinkel finds more bugs in both Neo4j and RedisGraph. We describe the relations of the found bugs in Figure 18. Note that none of these approaches have found logic bugs within 48 hours, so the bugs shown are related to internal errors or crashes. Specifically, Dinkel found 11 bugs in Neo4j. GDBMeter, GRev, and GAMERA found 1 bug, and this bug can also be found by Dinkel. In RedisGraph, Dinkel found 17 bugs, while GRev and GDBMeter found only 1 bug, and this bug can also be found by Dinkel. For the 27 bugs missed by existing approaches, their bug-triggering queries contain either complicated data dependencies or advanced Cypher features that are not supported by existing approaches. The bug shown in Figure 16 is one of the 27 bugs. These results demonstrate that Dinkel can find more bugs than existing approaches by generating more complex Cypher queries.

6 RELATED WORK

GDBMS Testing. GDBMS testing is an emerging research field and several approaches [17, 18, 24, 30, 54, 56] are proposed. All these approaches integrate techniques for constructing test oracles to identify bugs, especially logic bugs (*i.e.*, the bugs cause GDBMSs to return incorrect results). Grand [54] targets GDBMSs using the Gremlin query language and detects logic bugs using differential testing [31]. Several approaches [17, 18, 24, 30, 56] support GDBMSs using Cypher. Similar to Grand, GDsmith [17] also uses differential testing to find logic bugs. GDBMeter [24] leverages the idea of query partitioning [41], which decomposes the predicate of a query and checks whether the queries with the decomposed predicates produce consistent results with the original query. GraphGenie [18] modifies the graph patterns used in a query and checks whether the query with modified graph patterns satisfies the expected relationship (*i.e.*, equivalence, subset, or superset) with the original queries. GAMERA [56] extracts metamorphic relations [6] from the manipulated graph data and validates GDBMSs by checking whether the outputs of generated queries satisfy these relations.

Different from all these existing approaches, which focus on new test oracles for finding logic bugs, Dinkel focuses on improving query generation, which is the groundwork of GDBMS testing. An interesting future work is to combine the techniques of Dinkel and existing approaches, enhancing both query generation and test-oracle construction for finding more bugs in GDBMSs.

RDBMS Testing. Compared to GDBMS testing, testing relational database management systems (RDBMSs) is more mature, where both query generation [14, 20, 26, 45, 55] and test-oracle construction [16, 22, 40–43, 46] are well-researched. We focus on discussing the work on query generation for RDBMSs. SQLsmith [45] embeds the SQL grammar [44] and can generate queries containing complex statements. SQUIRREL [55] integrates a new intermediate representation (IR) for modeling the dependencies among statements in a SQL query. Therefore, SQUIRREL can generate queries containing multiple statements. DynSQL [20] incrementally generates SQL statements for a query. For each statement, DynSQL queries the tested RDBMS to capture the latest DBMS state. In this way, DynSQL can generate complex and valid queries with multiple statements. LEGO [26] proposes type-affinity to describe the pattern of composing two types of SQL statements. Based on the type-affinity extracted from existing queries, LEGO can mutate and synthesize new queries that have a higher chance of increasing the code coverage of RDBMSs.

Different from these approaches, Dinkel is designed for GDBMSs and integrates techniques for improving Cypher query generation.

State-Aware Fuzzing. Fuzzing is a promising technique for detecting bugs in various software [2, 5, 19, 23, 27, 37, 51]. Some approaches [4, 20, 25, 28, 53] have been proposed to find bugs more efficiently in state-sensitive systems. RESTler [4] analyzes the API specification of the tested cloud service and generates request sequences that follow the inferred producer-consumer dependencies. RESTler also collects the response observed during prior executed requests to guide subsequent request sequence generation. LOKI [28] proposed to test blockchain consensus protocols. It builds a state model to dynamically track the state transition of each node in the blockchain systems and accordingly generates inputs with proper targets, types, and contents. StateFuzz [53] is designed to test Linux drivers. It utilizes static analysis to recognize critical variables that affect the control flows or memory accesses, and represents program states using these variables. StateFuzz prioritizes test cases triggering new states using three-dimensional feedback. To effectively test USB gadget stacks, FuzzUSB [25] extracts the internal state machines from USB gadget drivers via static analysis and symbolic execution, and uses such state information as fuzzing feedback to guide test-case generation.

Similar to these approaches, Dinkel also leverages state information and is the first state-aware testing approach designed for detecting bugs in GDBMSs.

7 CONCLUSION

In this paper, we have presented a novel, practical approach for testing GDBMSs using complex Cypher queries. Our approach models the graph state into two categories: *query context* and *graph schema*. Based on the model, we have proposed *state-aware query generation*, which incrementally constructs the clauses for the generated queries and intermittently updates the state information after each clause is constructed. We have implemented our approach as an automatic GDBMS testing framework, Dinkel, and evaluated it on three popular GDBMSs. The evaluation results demonstrate that Dinkel can effectively generate complex Cypher queries with high validity. Dinkel found 60 bugs, many of which had been missed by existing approaches. Considering its effectiveness in Cypher query generation, we believe Dinkel can be the foundation for GDBMS testing, facilitating and inspiring follow-up research, like Csmith [50] for compiler testing.

REFERENCES

[1] Address Sanitizer. 2019. https://clang.llvm.org/docs/AddressSanitizer.html.

[2] American Fuzzy Lop. 2021. https://github.com/google/AFL.

[3] Apache AGE. Graph database optimized for fast analysis and real-time data processing. 2024. https://github.com/apache/age.

[4] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful REST API Fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE).* 748–758.

[5] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 2018 Symposium on Security and Privacy (S&P).* 711–725.

[6] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. HKUST-CS98-01. Department of Computer Science, HKUST.

[7] Creating a Modern Solution to Finding What's Lost with Neo4j. 2024. https://neo4j.com/case-studies/notlost.

[8] CVE-2018-1000820. 2018. https://nvd.nist.gov/vuln/detail/CVE-2018-1000820.

[9] CVE-2020-35668. 2020. https://nvd.nist.gov/vuln/detail/CVE-2020-35668.

[10] Cypher Query Language Reference. 2022. https://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf.

[11] DB-Engines Ranking. 2024. https://db-engines.com/en/ranking.

[12] Docker: Accelerated Container Application Development. 2024. https://www.docker.com/.

[13] FalkorDB. A super fast Graph Database uses GraphBLAS under the hood for its sparse adjacency matrix graph representation. 2024. https://github.com/FalkorDB/FalkorDB/.

[14] Jingzhou Fu, Jie Liang, Zhiyong Wu, Mingzhe Wang, and Yu Jiang. 2022. Griffin: Grammar-Free DBMS Fuzzing. In *Proceedings of the 37th International Conference on Automated Software Engineering (ASE).* 1–12.

[15] Git-bisect: Use binary search to find the commit that introduced a bug. 2024. https://git-scm.com/docs/git-bisect.

[16] Zongyin Hao, Quanfeng Huang, Chengpeng Wang, Jianfeng Wang, Yushan Zhang, Rongxin Wu, and Charles Zhang. 2023. Pinolo: Detecting Logical Bugs in Database Management Systems with Approximate Query Synthesis. In *Proceedings of the 2023 USENIX Annual Technical Conference (ATC).* 345–358.

[17] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of the 2023 International Symposium on Software Testing and Analysis (ISSTA).* 163–174.

[18] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland Hock Chuan Yap, Zhenkai Liang, and Manuel Rigger. 2023. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of the 46th International Conference on Software Engineering (ICSE).* 531–542.

[19] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2020. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *Proceedings of the 29th USENIX Security Symposium.* 2595–2612.

[20] Zu-Ming Jiang, Jia-Ju Bai, and Zhendong Su. 2023. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. In *Proceedings of the 32nd USENIX Security Symposium.* 4949–4965.

[21] Zu-Ming Jiang, Si Liu, Manuel Rigger, and Zhendong Su. 2023. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 397–417.

[22] Zu-Ming Jiang and Zhendong Su. 2024. Detecting Logic Bugs in Database Engines via Equivalent Expression Transformation. In *Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 821–835.

[23] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. 2022. Context-Sensitive and Directional Concurrency Fuzzing for Data-Race Detection. In *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS).*

[24] Matteo Kamm, Manuel Rigger, Chengyu Zhang, and Zhendong Su. 2023. Testing Graph Database Engines via Query Partitioning. In *Proceedings of the 2023 International Symposium on Software Testing and Analysis (ISSTA).* 140–149.

[25] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. 2022. FuzzUSB: Hybrid Stateful Fuzzing of USB Gadget Stacks. In *Proceedings of the 2022 International Symposium on Security and Privacy (S&P).* 2212–2229.

[26] Jie Liang, Yaoguang Chen, Zhiyong Wu, Jingzhou Fu, Mingzhe Wang, Yu Jiang, Xiangdong Huang, Ting Chen, Jiashui Wang, and Jiajia Li. 2023. Sequence-Oriented DBMS Fuzzing. In *Proceedings of the 2023 International Conference on Data Engineering (ICDE).* 668–681.

[27] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers. In *Proceedings of the 28th USENIX Security Symposium.* 1949–1966.

[28] Fuchen Ma, Yuanliang Chen, Meng Ren, Yuanhang Zhou, Yu Jiang, Ting Chen, Huizhong Li, and Jiaguang Sun. 2023. LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols. In *Proceedings of the 30th Network and Distributed System Security Symposium (NDSS).*

[29] David Maciver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP).* 13:1–13:27.

[30] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. In *Proceedings of the 46th International Conference on Software Engineering (ICSE).* 143:1–143:12.

[31] William M McKeeman. 1998. Differential testing for software. *Digital Technical Journal* 10, 1 (1998), 100–107.

[32] NBC News Analyzes Hundreds of Thousands of Russian Troll Tweets Using Neo4j. 2021. https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-case-study-NBC-News-EN-US.pdf.

[33] Neo4j and Generative AI 2023. https://neo4j.com/generativeai/.

[34] Neo4j. Graphs for Everyone. 2024. https://github.com/neo4j/neo4j.

[35] Novartis Captures the Latest Biological Knowledge for Drug Discovery. 2021. https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-case-study-Novartis-EN-US.pdf.

[36] openCypher. 2024. https://opencypher.org/.

[37] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th International Symposium on Software Testing and Analysis (ISSTA).* 329–340.

[38] RedisGraph. A graph database as a Redis module. 2023. https://github.com/RedisGraph/RedisGraph.

[39] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 2012 International Conference on Programming Language Design and Implementation (PLDI).* 335–346.

[40] Manuel Rigger and Zhendong Su. 2020. Detecting Optimization Bugs in Database Engines via Non-optimizing Reference Engine Construction. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESE/FSE).* 1140–1152.

[41] Manuel Rigger and Zhendong Su. 2020. Finding Bugs in Database Systems via Query Partitioning. In *Proceedings of the 2020 International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA).* 1–30.

[42] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI).* 667–682.

[43] Jiansen Song, Wensheng Dou, Ziyu Cui, Qianwang Dai, Wei Wang, Jun Wei, Hua Zhong, and Tao Huang. 2023. Testing Database Systems via Differential Query Execution. In *Proceedings of the 45th International Conference on Software Engineering (ICSE).* 2072–2084.

[44] SQL standard. 1992. https://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt.

[45] SQLsmith: A random SQL query generator. 2023. https://github.com/anse1/sqlsmith.

[46] Xiu Tang, Sai Wu, Dongxiang Zhang, Feifei Li, and Gang Chen. 2023. Detecting Logic Bugs of Join Optimizations in DBMS. In *Proceedings of the 2023 International Conference on Management of Data (SIGMOD).* 1–26.

[47] Unifying LLMs and Knowledge Graphs for GenAI: Use Cases and Best Practices 2023. https://neo4j.com/blog/unifying-llm-knowledge-graph/.

[48] White Paper: Powering Real-Time Recommendations with Graph Database Technology. 2024. https://neo4j.com/whitepapers/recommendations-graph-database-business/.

[49] Who Uses Neo4j. 2024. https://neo4j.com/who-uses-neo4j.

[50] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd International Conference on Programming Language Design and Implementation (PLDI).* 283–294.

[51] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium.* 745–761.

[52] Andreas Zeller. 2009. *Why programs fail: a guide to systematic debugging.* Elsevier.

[53] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. StateFuzz: System Call-Based State-Aware Linux Driver Fuzzing. In *Proceedings of the 31st USENIX Security Symposium.* 3273–3289.

[54] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. Finding Bugs in Gremlin-Based Graph Database Systems via Randomized Differential Testing. In *Proceedings of the 2022 International Symposium on Software Testing and Analysis (ISSTA).* 302–313.

[55] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. 2020. SQUIRREL: Testing Database Management Systems with Language Validity and Coverage Feedback. In *Proceedings of the 2020 International Conference on Computer and Communications Security (CCS).* 955–970.

[56] Zeyang Zhuang, Penghui Li, Pingchuan Ma, Wei Meng, and Shuai Wang. 2023. Testing Graph Database Systems via Graph-Aware Metamorphic Relations. *Proceedings of the 50th International Conference on Very Large Databases (VLDB)* (2023), 836–848.