ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

AST

Dinkel: Fuzzing Graph Databases with Complex and Valid Cypher Queries

Bachelor's Thesis

by

Dominic Wüst

Advanced Software Technologies Lab
ETH Zürich

Supervised by

Zu-Ming Jiang
Prof. Dr. Zhendong Su

August 4, 2023

Abstract

Graph database management systems (GDBMSs) have become essential in data-driven applications. To ensure the reliability and security of these systems, solutions for fuzzing their graph query languages (GQL) have been implemented. However, the most common GQL, Cypher, introduces complicated data flow and state, which creates difficulty in generating complicated but still valid queries. As a result, previous results have been unable to incorporate these language constructs when generating queries, causing bugs triggered by these features to remain undetected.

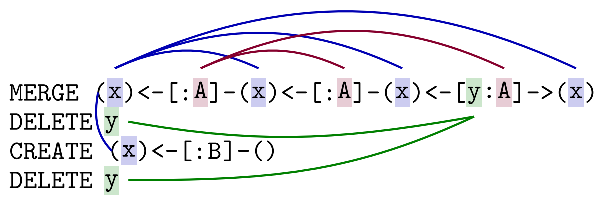
In this paper, we propose a novel technique for generating complicated and valid Cypher queries with complicated data flow and data dependencies. We achieve this by keeping track of what we call the query context and the abstract graph summary. These hold information about variables in scope and an approximation of the graph at a certain point in the query, respectively. We use on-the-fly state manipulation to modify this information during query generation, thereby ensuring that clauses within the query can use this state information to generate complicated queries. We implement this approach as a fully automatic fuzzer, Dinkel. Dinkel is evaluated on three of the most popular GDBMSs (Neo4j, RedisGraph and Apache AGE). In our evaluation, Dinkel outperforms other state-of-the-art Cypher fuzzers, creating queries which are difficult for previous approaches to generate and finds 53 previously unknown bugs.

1 Introduction

Graph Database management systems (GDBMSs) play an essential role in today's interconnected, data-driven lives. Their practicality in financial fraud detection [7,10], networking [8,9] and other data-driven applications [3,11] has led 75% of the Fortune 100 and all of North America's top 20 banks to make use of the currently most popular [6] GDBMS, Neo4j [15].

GDBMSs are rapidly evolving, complex systems and thus error-prone. Their usage in critical fields could lead to serious consequences if a bad actor were to abuse a previously unknown bug. An attacker may run a Denial of Service attack by repeatedly crashing the system [13,17] or leak confidential data [16].

Previous approaches [28,30] have attempted to fuzz GDBMSs through automatic Cypher query generation. However, they do not account for some characteristic Cypher features which results in a lower complexity of generated queries. Different from declarative languages like SQL [25], Cypher queries have the capability of making changes to the database state during query execution which are visible in subsequent clauses [4]. Neglecting



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

Figure 1: A query that causes RedisGraph v2.10.11 to encounter an untrue assertion. Highlighted are data dependencies within the query.

this behaviour results in the query processor not having to handle complicated data dependencies, resulting in more shallow fuzzing. This poses the difficult challenge of correctly maintaining usable state information, which does not impact query validity while enabling the fuzzer to generate more complex queries with data dependencies between clauses. When compared to imperative languages like C [29], Cypher has more ways of manipulating state with information which can be referred to in later parts of a statement [4]. This poses the problem of capturing this information and incorporating it in the generation of later clauses, thereby forcing the GDBMS to handle this intricate data flow.

During the execution of a Cypher query, two origins of state exist. These consist of the current database schema and the query context, holding declared variables. The schema holds references to nodes and their labels, relationships and their type, as well as properties for both of them. Keeping track of the query context entails correctly storing all variable identifiers in scope as well as their associated type. The database schema can be approximated during generation by storing a set of employed node labels and relationship types, as well as all used property identifiers and their exact type. Inexact information leads to ineffective testing. Missing node labels and relationship types lead to fewer data dependencies and reduced query complexity while type errors may arise from incorrect variables and properties, reducing query validity.

To increase the complexity of generated queries, we perform state tracking by using on-the-fly state manipulation. When generating a query, we incrementally generate an abstract syntax tree (AST), where every node represents a Cypher clause or expression. With on-the-fly state manipulation, every such AST node gets granted the ability to update the global state during its generation. This state information then gets used by subsequent clauses through referencing declared variables or graph elements, resulting in data dependencies and thereby increasing query complexity. Figure 1 shows a query which causes RedisGraph to encounter an un-

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

Figure 2: A complex query that triggers an exception bug in Neo4j 5.6.0.

true assertion. Every line connecting two elements in this query represents a data dependency. As previous approaches neglect state information, they do not have the ability to generate such a query and missed this bug.

Query complexity is enhanced by letting the AST grow during generation. Instead of generating an AST skeleton and then filling out the gaps, this approach instead generates nodes which then randomly pick between further nodes that may follow it. This method ensures syntactic correctness while enabling theoretically infinite query sizes. In addition to that, we use a modular approach when defining AST nodes, through which we are able to easily expand our tool to target all language features. The query in Figure 2 shows such a complex query. Previous approaches have limited coverage of the OpenCypher language, thereby they aren't able to generate this query, containing complex clauses like `FOREACH` and the `CALL` subquery which possess non-trivial data flow.

Based on our approach, we implemented Dinkel, an open-source Cypher fuzzer producing complex and valid queries using stateful generation. We have evaluated our tool against three popular GDBMSs (Neo4j [20], RedisGraph [21] and Apache AGE [19]) and found 53 unique, previously unknown bugs. The bugs consist of 17 crashes and 36 exception bugs. So far,33 have been fixed and a further 11 of these bugs have been confirmed.

Overall, we make the following technical contributions:

- A novel approach to Cypher query generation, enabling fuzzing of GDBMSs using complex queries containing intertwined, data-dependent clauses. With a framework allowing for easy target domain expansion using declarative drop-ins for implementation-specific language structures.
- Based on our approach, we implement Dinkel, an easy-to-adapt Cypher fuzzer written in Go.
- We evaluate Dinkel on three widely-used GDBMSs, including Neo4j, RedisGraph and Apache AGE in which Dinkel finds 53 unique bugs. We compare Dinkel to previous approaches of Cypher query generation and evaluate it based on the generated queries' complexity and data dependence among clauses.

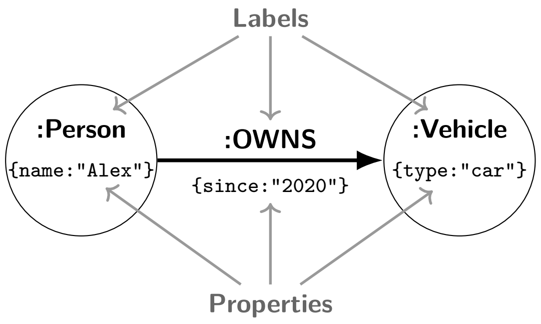


Figure 3: A property graph modelling a person named Alex owning a car.

2 Background

This section introduces the background knowledge required to understand our approach and design decisions.

2.1 GDBMS

A GDBMS is a database system operating on property graphs, storing interconnected data using nodes connected via edges [14]. Nodes and edges, so-called graph entities, may hold additional data through labels and properties. Labels are used to group and classify elements, whereas properties are made up of key-value pairs, providing additional information about an element.

Figure 3 shows an example of such a property graph. It models a person named Alex owning a vehicle of the type car since 2020. The leftmost node, representing Alex, has the label `Person` and a property with name `name` and value `Alex`. This node is connected to another node via an edge of type `OWNS`. On this edge, a property with a key of `since` and value of `2020` is held. This edge goes from the node representing Alex to a node representing a car. This car node possesses the label `Vehicle` and houses a property named `type` with value `car`.

GDBMSs are optimized to operate on large-scale graphs by traversing and modifying them according to rules given by a user's query.

Since graphs often represent continuously developing or strongly interconnected data, it is natural for graph databases to demand few to no schema definitions. However, graph databases often allow defining rules for data integrity through uniqueness or existence constraints. This is notably different from relational databases, which require a schema to be rigidly defined before it may be populated with data [2].

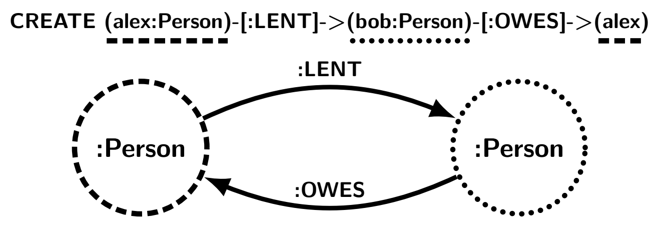


Figure 4: A simple Cypher query and the graph it creates.

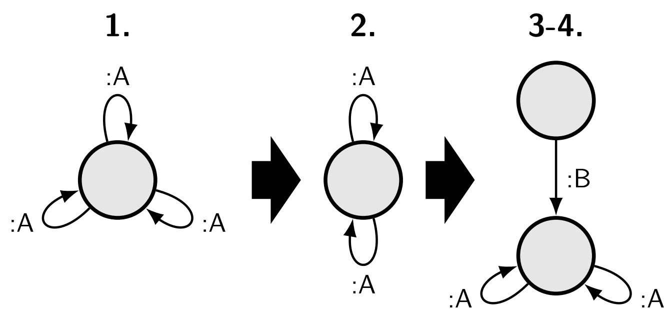


Figure 5: The state after processing each line of the query in Figure 1.

2.2 Cypher

While currently there is no set standard for graph query languages, Cypher is a promising candidate. Its usage in the most popular GDBMS, Neo4j [5], and their efforts in establishing the OpenCypher [12] standard makes it stand above other graph query languages.

Cypher makes use of declarative path-pattern matching and combines this with clauses operating on the query context and graph state. Path patterns are used in queries to reference graph structures and are constructed using a syntax reminiscent of ASCII art. Figure 4 contains a simple Cypher query and the graph it creates. The query creates a node of type `Person`, which gets assigned to the variable `alex`, represented in the graph with the dashed node. This variable is only referenceable in the query itself and thereby its name does not appear in the graph. Alex gets connected to the node `bob`, the dotted node in the graph, via an edge of type `LENT`. The query then creates another node from `bob` back to `alex` of type `OWES`.

Cypher clauses can be put into two main categories consisting of write and read clauses. Write clauses change the graph data by creating or modifying it. Read clauses, meanwhile, query the graph to extract information without modifying it.

Different from traditional query languages like SQL [25], Cypher does not differentiate between data declaration (DDL), manipulation (DML) and query (DQL) language. 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.

The Cypher query in Figure 6 is making use of this behaviour. It first matches a graph, the Cypher equivalent of querying. The next clause then creates a graph pattern, while the last clause modifies the property of one of the nodes just created.

A Cypher statement is constructed by chaining optional read clauses, followed by a return or a chain of write clauses. Cypher statements themselves can further be chained using the WITH clause, after which read clauses are syntactically correct again. Using the WITH clause thus allows write clauses to be followed by more read clauses.

With this procedural approach to query construction, data flow and state visibility are not straightforward. During query execution, clauses can observe all writes performed by previous instructions. The query state can thus change drastically during query processing, with state information captured at the start of the query possibly not matching the one present in subsequent clauses.

Figure 5 shows the database state after processing each line of the query depicted in Figure 1. The displayed graph is the one visible at the end of processing a line, thereby the one the subsequent clause operates on. Notice how the first DELETE clause has the ability to delete the edge created in the previous MERGE clause, causing the graph in the second graph to have an edge missing when compared to the first graph. This behaviour is what gifts Cypher its imperative nature while also complicating a query's data flow and state.

2.3 Existing Approach Limitations

Existing approaches do not capture the same level of state information and can thus not reach a high complexity in their generated queries.

GDSmith [30] restricts itself in query complexity by separating read- and write statements as well as by defining the graph in advance, thus not incorporating changes to the schema during query processing. Additionally, GDSmith only implements the core OpenCypher grammar, which does not include more complex clauses required by the OpenCypher specifications.

GDBMeter [28] intentionally restricts its query complexity by focusing on finding bugs using predicate partitioning [31]. However, GDBMeter also misses bugs by ignoring state information, thus not creating data depen-

```
MATCH (a:Actor {name: "Tom Hanks"})  // Query
CREATE (a)-[:PLAYS_IN]->(b:Movie)    // Create
SET b.title = "Forrest Gump"         // Modify
```

Figure 6: A Cypher query reading, creating and modifying data in a single statement.

dencies within its clauses.

These limitations have all been addressed in our approach and enabled us to find many bugs missed by previous approaches.

3 Approach

3.1 Approach Overview

To address the limitation of state-independent queries, we model the query state during generation and refer back to this information later on. Through the query state, we try to accurately capture data referenceable by clauses and incorporate it in their generation, thereby increasing query complexity.

On-the-fly state manipulation gets used to operate on the query state model. It involves statically inferring state information from clauses during generation. This information then gets added to a global state, allowing subsequent clauses to refer to information previously created within the query.

3.2 Query State Modeling

We differentiate between two parts making up the query state.

The first part contains the abstract graph summary, which is an approximation of the graph stored in the database, statically inferred during query generation. This summary contains graph attributes referenced in the query. We track all graph attributes, which are made up of node labels, relationship types and properties with their respective types.

The second part is the query context, which contains variables in scope and their types. These variables can be property variables with simple datatypes, or structural variables, holding references to nodes or relationships.

An example state can be seen in Figure 7. The set of variables makes up the query context, whereas the abstract graph summary contains the labels and properties sets. After the `CREATE` clause, the query context contains the variables `x` and `y`, whereas the abstract graph summary consists of the label `A` and property name `n0`.

```
                        // Vars  Labels Props
WITH 0 AS x             // {x}   {}     {}
CALL {                  //
  WITH 1 AS y           // {x,y} {}     {}
  CREATE (:A {n0:y})    // {x,y} {A}    {n0}
}                       //
RETURN x                // {x}   {A}    {n0}
```

Figure 7: How the tracked state changes throughout a query. What is omitted from the figure is the additional information attributed to these entities. The fuzzer would hold additional information about the type of `x` and `y`, the fact that the origin of the `A` label is a node as well as the type and origin of the `n0` property.

3.3 On-The-Fly State Manipulation

Since state changes throughout a Cypher query, every clause can have an effect on the abstract graph summary and query context. To accurately keep track of these changes, we store the reference to a global state with every AST node. This state is thereby freely accessible and modifiable by the node during its generation. Depicted in Figure 7 is the tracked state during a query. The sets on the right side of the figure show what the state holds and how it is modified on-the-fly. More specifically, the sets containing the names of tracked variables, labels and properties are shown. Notice how after the `CALL` subquery, the query context shrinks, as the variable defined within the subquery goes out of scope. Meanwhile, the abstract graph summary remains the same. This results in later parts of the query not incorrectly referencing variables outside the current scope, but possibly creating data dependencies through further references of previously defined labels and properties.

AST nodes incorporate this state information to guide their generation. With this approach, every time a node uses state information in its generation, a data dependency gets created. Figure 8 shows an example of how these dependencies get created. Arrows pointing to the global schema populate it with data, while arrows pointing away from it depict the incorporation of state information in the query. Every connecting line below the query represents a data dependency. The depicted query generates two variables, `x` and `y`, which get stored in the query context. The abstract graph summary gets populated by the label `A` and the property name `n0`. Notice how every data incorporation, or down-pointing arrow in the figure, induces a data dependency.

However, some AST nodes have to modify the state after generation to enforce a correct state. For example, the CALL clause in Figure 7 has to ensure that the variable `y` in the subquery is not visible outside its body.

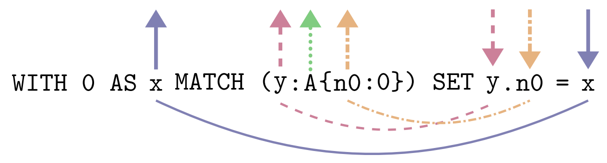


Figure 8: Data coupling within a query through global state.

Referencing this variable outside the subquery is semantically incorrect. Because of this, the variable must be removed from the schema after the `CALL` subquery's AST got generated. This issue is addressed by implementing a method for relevant AST nodes which gets called after its generation is finished, which has the ability to modify the state.

4 Implementation

4.1 Architecture

Our tool's core logic can be split into two parts, the scheduler and the query generator. Figure 9 illustrates how these parts interact. The query generator's job is to generate Cypher queries by building and then translating query ASTs. The scheduler meanwhile is responsible for fetching queries from the query generator which it then sends to the database under test before analyzing the result. A fuzzing run starts off with the scheduler requesting a query from the query generator given a seed and the OpenCypher implementation under test. After receiving a query from the query generator, the scheduler then sends this query to the target GDBMS and waits for the result. The result then gets analyzed and if it indicates that a bug has been triggered, the scheduler generates and outputs a bug report.

4.2 Query Generation

The query generator creates a query by recursively generating the AST associated with the root node and a given schema. An example AST being generated is shown in Figure 10. The query generator asks the AST root node to return its child nodes, which then further get generated. The AST gets traversed in a pre-order approach by the query generator, generating a node's leftmost child and all its descendants before moving on to the next child.

4.3 AST Node Metamodel

The node interface is a struct implementing the Generate method. This Generate method takes in a seed and schema, which the node uses to guide its generation. The method returns a list of further AST nodes, which represent its children in the abstract syntax tree.

Nodes additionally implement the TemplateString method, which returns a format string. The format string contains string verbs, which act as the locations where the node's children get inserted during translation.

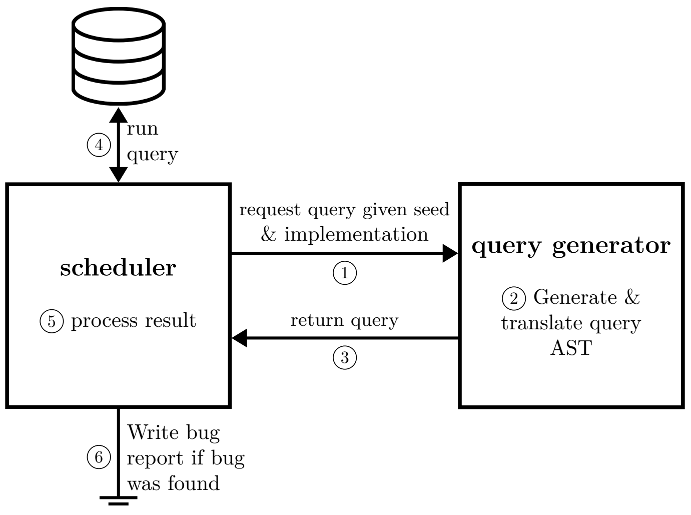

run query
④
scheduler
⑤ process result
request query given seed & implementation
①
return query
③
query generator
② Generate & translate query AST
Write bug report if bug was found
⑥

Figure 9: How the scheduler, query generator and target GDBMS interact during fuzzing.

4.4 Model Adaptation

While the targets tested using our tool focus on adhering to the OpenCypher specifications, there are still differences between their implementations. These differences can range from simple functions having different signatures, over containing unimplemented clauses, to introducing completely new clauses. Ignoring these characteristics causes a larger amount of invalid queries and worse target domain coverage.

Simple differences, like unimplemented types and different function signatures, get addressed by having every implementation define a generation config. This config holds a list of disallowed OpenCypher functions, additional implementation-specific functions as well as unimplemented property types. Relevant AST nodes use this config and incorporate its information during their generation.

For more difficult-to-address discrepancies, like differing clause semantics, we employ declarative AST node drop-ins. Every implementation defines a drop-in map with AST node types as the keys and a drop-in function as a value. This drop-in function accepts an AST node, the current schema as well as seed and returns a new AST node. When the query generator encounters an AST

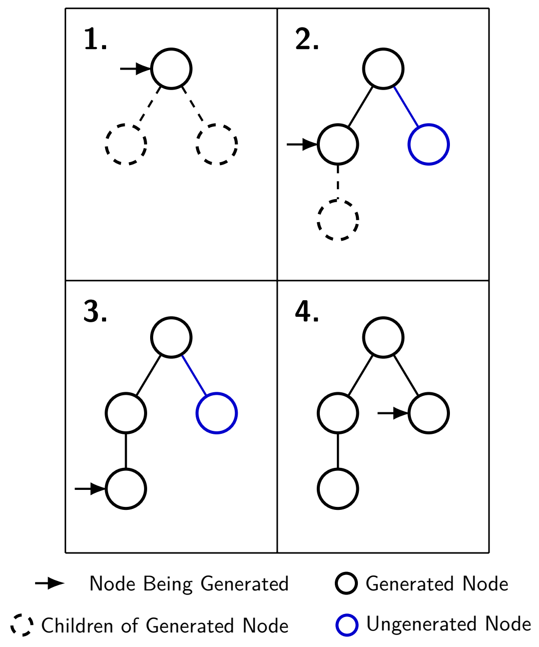


Figure 10: How the query generator generates and traverses a query's AST.

node matching a key from the drop-in map, it replaces it with the returned node from the function associated with the encountered node. Instead of translating the original node, the query generator now generates the drop-in instead. Figure 11 shows how these drop-ins adapt the query AST. The dashed nodes are ones which would have been generated, were no drop-ins defined. However, since the implementation under test defined a drop-in for the node type of `x`, the query generator replaces `x` with its associated drop-in. This causes the query generator to continue AST generation from this drop-in, coloured in purple, instead of `x`.

4.5 Query Reduction

In order not to overburden developers with large queries in bug reports, we reduced all bug-triggering statements to a minimal, viable example.

This was automated by having the fuzzer regenerate the original query's AST and then delete nodes, thereby essentially replacing their translation with an empty string. After every such deletion, the AST is translated to a Cypher query again and sent to the database, checking if the same bug still triggers. If the bug triggers, the modified AST is used for further reduction, otherwise, the fuzzer falls back to the previous AST.

Additionally, AST nodes themselves are able to implement a method returning another node for reduction, thereby instead of deleting the node, replacing it with the given alternative. This approach is effective for nodes whose absence in the query would be a syntactic error. For example, a `UNION` clause could, instead of just deleting one of its subclauses, which would be a syntactic error, return just one subclause instead.

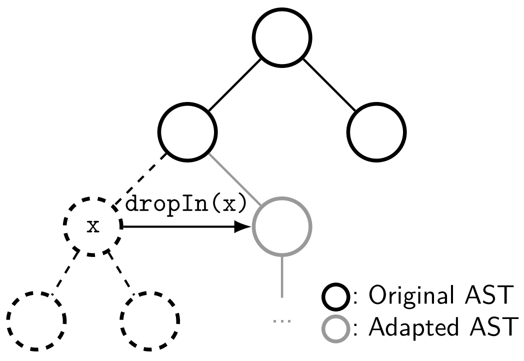


Figure 11: An illustrative example showing how an implementation drop-in can adapt the statement's AST. Dashed elements represent AST nodes which would have been generated, were the drop-in not defined.

With this approach, bug-triggering queries could be efficiently reduced, only requiring little manual work afterwards to minimize the final queries. This manual work consisted of removing a few clauses missed by the reducer or improving the report's readability by replacing constants (e.g. a random integer to a simple zero), removing whitespace and renaming variables.

5 Evaluation

5.1 Setup

We ran our tests on a machine running Ubuntu 20.04 on a 64-Core AMD EPYC 7742 processor running at 2.25GHz with 256GB of RAM.

We evaluated our tool against Neo4j version 5.6.0, which was the newest version when Dinkel started being developed.

In this section, we differentiate between three types of queries.

Valid queries are queries for which the GDBMS successfully produces a result.

Invalid queries are queries which cause an exception indicating a syntactic or semantic error to be raised.

Bug-triggering queries are all queries crashing the target system and exception-raising queries which do not indicate an invalid query.

Queries were set to time out after 15 seconds and were forcefully killed after 30 seconds, as execution could get stuck even after the 15-second timeout if the target instance was running out of memory.

GDBMS	Exceptions	Crashes	Total
Neo4j	25	0	25
RedisGraph	3	14	17
Apache AGE	8	3	11
Total	36	17	53

Table 1: The distribution of exceptions and crashes among the tested GDBMSs. The last row holds the sum of all previous rows.

We excluded queries resulting in timeouts from our evaluation, as we could not test whether they trigger a bug or contain linguistic errors. On average, queries resulting in a timeout made up about 1.5% of all queries. These timeouts were mainly caused by iterating over immensely large arrays or slowing down the target system by using most of its assigned memory.

5.2 Bug Detection Novelty

In total, Dinkel found 53 unique, previously unknown bugs. The bugs consist of 17 crashes and 36 exception bugs. So far, 11 of these bugs have been confirmed and 33 have been fixed. Table 1 shows how these numbers are distributed among specific GDBMSs.

Of these listed bugs, two were regression bugs found on the release dates of their respective version. One of them affected Neo4j 5.6.0, whereas the second one affected 5.7.0. Our tool can thereby efficiently be used by GDBMS developers for finding recently introduced bugs before they reach their clients.

We analyse the two representative bugs depicted in Figure 1 and Figure 2 to explain how these bugs could only be found using Dinkel's novel techniques.

The data dependencies required to trigger the bug listed in Figure 1 prevented previous approaches from finding it. Dinkel is the first Cypher fuzzer with the ability to create such dynamic data dependencies spanning multiple clauses. GDSmith [30], while capturing and reusing variables, does so only for read clauses and cannot create new dependencies between graph elements without directly referencing the variable that captures them. Because of this, the dependencies induced by referencing the newly created label `A` could not have been created by GDSmith.

The level of query complexity required to trigger the bug in Figure 2 cannot be reached by other Cypher fuzzers. Previous approaches are not able to capture and incorporate the non-trivial dataflow and query state present in the query. This kept them from generating the employed clauses, like the `CALL` or `FOREACH` clause.

5.3 Runtime Testing Evaluation

We ran Dinkel against Neo4j 5.6.0 for 24 hours and collected all generated queries. In this section, we summarize the findings from this test run and explain some obtained metrics.

5.3.1 Generated Query Validity

Of the generated queries, 87.5% were valid, while 12.5% were invalid.

Query invalidity is caused by one or more of four categories.

Arithmetic Exceptions. Since we do not know the concrete values of most expressions in a query during runtime, adhering to arithmetic constraints is not possible. Queries can thereby fail due to arithmetic exceptions, caused by numerical over- and underflow, divisions by zero or similar edge cases.

OOM Exceptions. Similarly to arithmetic exceptions, it is impossible to predict the amount of memory used by a query. Additionally, built-in methods returning lists, for example, the `range` function, can blow up memory usage drastically. Such an increase in memory consumption causes an exception in the GDBMS, which then terminates the query, causing an exception to be raised in the client.

Invalid Grouping Keys in Aggregate Expressions. Cypher provides the ability to apply aggregating functions on values, such as `SUM` or `COUNT`. However, queries containing such invocations may throw a semantic exception due to the presence of implicit grouping keys [1]. This means that for every such aggregation function invocation, the query must explicitly state over which values the function should group its expressions. Adhering to this rule from the perspective of a fuzzer is costly, difficult and may limit its target domain. Additionally, these exceptions only account for a fraction of invalid queries, thereby not greatly impacting the fuzzer's efficiency.

Abstract Graph Summary Unsoundness. The abstract graph summary only grows during query generation. When a graph element gets deleted, it still resides in our tracked state. Referencing this element in some updating clauses (e.g. `CREATE` and `MERGE`) causes an exception to be raised, as the element no longer exists. However, removing the ability to reference these elements would lead to missed bugs. The query in Figure 1 for example deletes the edge `y` twice, triggering a bug. This query is semantically correct however, as deletion of a non-existent element is a valid operation.

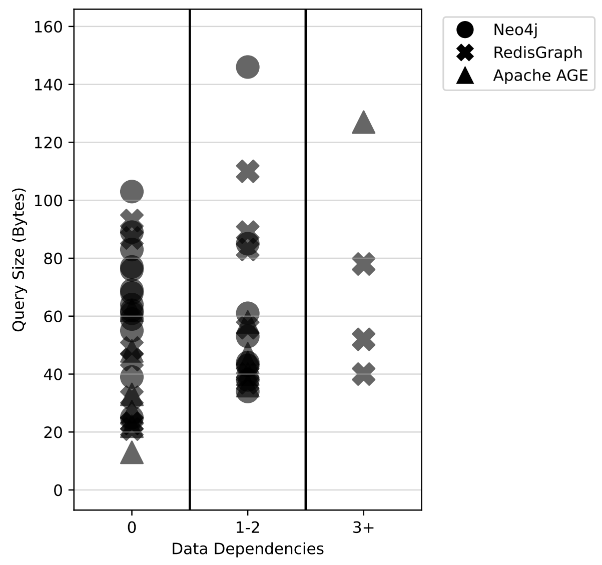


Figure 12: Query sizes and data dependencies of reduced bug-triggering queries.

5.3.2 Query Complexity

To convey the query complexity Dinkel can reach and the complexity required to trigger bugs, we analyse the complexity of queries gathered from the test run as well as all the reduced bug-triggering queries found by Dinkel.

We measure a query's complexity according to its data dependencies and size. Data dependencies are calculated by subtracting the number of times a symbol gets defined from the number of times it appears in the query. As Dinkel only ever defines a symbol once, this means that data dependencies are just the amount of symbols referenced minus 1.

For queries generated during the test run, the average amount of data dependencies per query reached 41 with an average query size of 1473 bytes.

We ran the same analysis for the 53 unique bugs found by Dinkel. Query sizes and data dependencies for these reduced queries are plotted in Figure 12.

Around half of all bugs require no data dependencies to trigger, and the other half mainly require 1-2. Very few, four, in this case, bugs trigger only if more than three data dependencies are present. This result is to be expected, following the Small Scope Hypothesis [22].

5.3.3 Bug-Triggering Feature Distribution

In order to determine which Cypher clauses were the root cause for most bugs, we analysed their distribution over all 53 bug-triggering queries generated by Dinkel. Figure 13 displays this distribution.

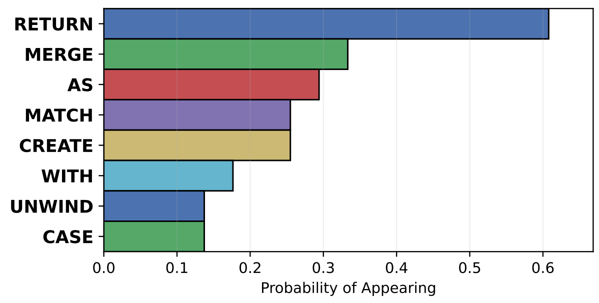


Figure 13: Feature distribution of bug-triggering queries.

The data shows that the `RETURN` keyword is present in most bug-triggering queries, appearing in roughly 60% of them. This is unsurprising, as `RETURN` is required to appear at the end of a query containing only reading clauses, otherwise, the query would be syntactically incorrect. `MERGE` and `CREATE` are the two clauses inducing the creation of graph elements and could often be used interchangeably in our specific queries while still triggering the same bug. We can deduce from this that graph data is often required to be present for bugs to trigger, The remaining keywords mainly induce non-trivial dataflow in the query, such as `WITH`, `UWNIND` and `CASE`, alluding to the fact that bugs often appear when the database system has to handle such flows.

5.4 Sensitivity Analysis

In order to see the impact different features had on the fuzzer's efficiency and metrics, we reran the evaluation for modified versions of our tool.

We conducted three more test runs with the same setup as before, but blocked certain features of the fuzzer:

- `dinkel` represents the original version, with all features enabled.
- $\texttt{dinkel}_{\texttt{!ags}}$ had the Abstract Graph Summary feature disabled. Generated queries did thus not have the ability to keep track of the graph summary, causing labels and properties to not get referenced after their first appearance in a query.
- In $\texttt{dinkel}_{\texttt{!qc}}$, we disabled the query context. Queries generated by this version can thereby not reference previously defined variables within the query.
- $\texttt{dinkel}_{\texttt{!qc,!ags}}$ had neither access to the query context, nor the abstract graph summary. Thus, this variant is unable to create queries with any data dependencies, as the fuzzer had no access to a query state.

Version	Query Count	Data Dependencies	Bug Count
dinkel	78’719	35.49	15
$\texttt{dinkel}_{\texttt{!ags}}$	88’444	9.62	14
$\texttt{dinkel}_{\texttt{!qc}}$	86’972	32.51	7
$\texttt{dinkel}_{\texttt{!qc,!ags}}$	78’589	0	7

Table 2: The results of the sensitivity analysis.

Table 2 displays the results gathered from these test runs, clearly showing that AGS has the biggest influence on the average data dependency count. This is most likely due to the fact that the elements AGS tracks, properties and labels, are not subject to scope constrictions and they can often appear chained together in large numbers. An example of such a query displaying this can be seen in Figure 14, where the graph pattern in the `EXISTS` subquery contains two labels, namely `B` and `C`. Generated queries can contain such label matches of much larger size by chaining them, similarly with properties.

During the 24-hour test runs, AGS didn’t seem to have a large effect on the amount of unique bugs found. This is most likely due to the fact that bugs which require AGS to trigger can take longer to generate, as they require complex graph structures to trigger.

Take for example the bug-triggering query listed in Figure 1, where QC and AGS work together to create different data dependencies, ultimately triggering a bug.

Additionally, AGS has a larger effect on how data flow occurs during the query execution, which may have a larger effect on logic bugs, rather than exception bugs. We believe this is the case, as AGS usually has less of an effect on how a query is processed, but rather what is being processed. This allows for exploiting mistakes in data processing, but not in the runtime, which would cause exceptions or crashes.

An example bug that could only be found using AGS is listed in Figure 14. Running this query in Neo4j 5.8.0 raises an exception, stating that properties cannot be accessed due to the relationship cursor being null. This is unexpected, as the query is both syntactically and semantically correct since nodes and relationships can share property names. The query first creates a graph pattern, where a node contains a property with a key of `x`. This property then gets added to the AGS and subsequently reused in the following `EXISTS` subquery. Notice how originally, the property was registered in a node, but referenced in an edge later. With AGS disabled, Dinkel would have no possibility of capturing this property and, by extension, induce the important data dependency, thereby missing the bug.

The bug-triggering query displayed in Figure 15 can only be triggered if the query context is enabled. When this query is executed, it results in an exception, complaining that the `x` inside the `EXISTS` subquery is shadowing a variable of the same name. However, as the inner `x` is simply referencing the previously defined one and not overwriting it, this is unexpected and thus points to a bug. This query could not be generated, were the query context disabled. Notice the data dependency created by the first node `x` being matched, which is later referenced in the `WITH` clause within the `EXISTS` subquery. The query context correctly captures the variable `x` and keeps its scope when entering the subquery. If the query context was disabled, this variable could not have been referenced again, as there would be no internal record of it.

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

Figure 14: A query that triggers an exception bug in Neo4j 5.8.0.

```
MATCH (x)-->({n0:EXISTS {
    WITH toBoolean(sum(0)) AS n1, x AS n2
    RETURN 0
} })
RETURN 0
```

Figure 15: A query that triggers an exception bug in Neo4j 5.8.0.

5.5 Comparison

To better understand the novelty Dinkel brings to the table, we analyse three exemplary bugs and determine why they could not be found by previous approaches.

Case Study 1: The query displayed in Figure 16 triggers a crash in RedisGraph v2.10.11. It contains language features which induce a non-trivial query context, such as the `none` predicate and list comprehension. These complex clauses cannot be generated by previous approaches due to limited domain coverage and their inability to capture the non-trivial variable scopes. For example, referencing `n2` outside the list comprehension or `n1` outside the `none` predicate would be semantically incorrect. Dinkel’s novel approach to clause generation and query context capturing, however, allows for this query to be generated, ultimately leading to a new bug being found. Also, notice the two data dependencies induced by `x` and `y`.

Case Study 2: The query displayed in Figure 17 trig-

```
CREATE x = ()-[y:A]->(),
    ({ n0:none(
         n1 IN [n2 IN [0] | x]
         WHERE false)
    })
MERGE ()<-[:B]-() ON CREATE SET y = {}
```

Figure 16: A query that causes RedisGraph v2.10.11 to crash.

gers an unexpected exception in Neo4j 5.6.0, reporting that the symbol `x` was not found. However, this query is correct, with the two references to `x` being distinct, thus no exception should be raised. This query contains the `FOREACH` clause, which entails difficult-to-capture data flow, with the variables defined within the body of the `FOREACH` clause not being visible outside the clause. Additionally, in order for the bug to trigger, the data dependency induced by the `A` label is required. No previous approaches can capture this data flow and create these data dependencies on-the-fly, leading this bug to be missed by them.

6 Related Work

6.1 Fuzzing

Fuzzing is a technique for automated testing. It involves randomly generating inputs for a program and observing the program's behaviour to check whether a bug has been triggered. Fuzzing tools can be domain-specific and target a specific domain [27, 28, 30–32], or they can be general purpose and produce test cases for arbitrary programs [18, 26].

Using dinkel we employ fuzzing on GDBMSs using the Cypher query language. Dinkel is thereby domain-specific, focusing on generating Cypher queries with complex to capture data flow and context changes.

Stateful Fuzzing uses state information to guide fuzzing, with the aim of making automated testing more exhaustive and efficient. Tools using stateful fuzzing [23, 24] have shown this technique's impact on the exhaustiveness of testing as well as its efficiency. RESTler [23], uses stateful fuzzing by inferring dependencies between restful API calls. The tool managed to find 38 bugs in two Microsoft services as well as GitLab. Stateful Greybox Fuzzing [24] rely on state information to guide the fuzzing of network protocols. The proposed technique automatically detects state variables with which it creates a state map of explored states which it then further uses to guide fuzzing. This stateful fuzzer resulted in double the fuzzing efficiency when compared to stateless greybox fuzzers and managed to find multiple

```
FOREACH ( n0 IN [] | CREATE (x) )
MERGE ()<-[x:A]-()<-[:A]->()
```

Figure 17: A query that triggers an exception bug in Neo4j 5.6.0.

security vulnerabilities, resulting in 8 assigned CVEs.

To our knowledge, dinkel is the first GDBMS fuzzer incorporating rich state information for generating complex queries. We use static inference to keep track of variables in scope and to approximate the generated graph. Dinkel still achieves high query validity, due to the semantics of the Cypher query language being more lenient when referencing nonexistent graph elements.

6.2 DBMS Testing

RDBMS Testing focuses on testing relational databases, which are databases operating on tables. Testing can focus on finding queries triggering logic bugs, finding security vulnerabilities and breaking invariants. Previous work using stateful RDBMS fuzzing [27, 32] has shown how such testing is crucial and powerful, finding and disclosing several critical security vulnerabilities. Squirrel [32] uses static inference to deduce state information. However, around half of its generated queries were invalid due to wrong state inferences stemming from incorrect inferences. DynSQL [27] fetches the state information directly from the database system before generating another SQL statement. This ensures state correctness, leading to a high query validity rate and thereby higher fuzzing efficiency.

Since dinkel is not focused on testing RDBMSs, it requires an adjusted approach and model to query generation, while still basing some core ideas on the previously mentioned work.

GDBMS Testing, meanwhile, performs testing on graph databases. Previous work [28, 30] has shown the necessity for this field, successfully finding bugs in tested systems. GDSmith [30] focuses on testing the core grammar of the Cypher query language, which results in a smaller domain coverage. GDBMeter [28], meanwhile, does not incorporate state information and intentionally restricts its query domain by testing for logic bugs with predicate partitioning.

Dinkel has a much broader domain coverage and enhances its queries using state information. It thereby taps into the deeper and more complex logic of the systems under test, causing more bugs to surface.

7 Conclusion

In this paper, we introduce dinkel, a stateful Cypher fuzzer. Dinkel employs on-the-fly state manipulation and abstract graph summaries to generate complex and valid queries with numerous data dependencies. We have evaluated dinkel on the three most popular GDBMSs using Cypher and were able to find 53 previously unknown bugs. Additionally, we analyze two representative bug-triggering queries found by dinkel which cannot be generated by previous approaches.

Acknowledgments

We thank all the developers of our target GDBMSs for looking into our bug reports and providing valuable feedback.

We thank our rubber duckies who were always faithfully by our side when problems arose during development.

Availability

Dinkel is open source and subject to the MIT license. It's repository is accessible at https://github.com/DominicWuest/dinkel.

References

[1] Aggregating functions - grouping keys. https://neo4j.com/docs/cypher-manual/current/functions/aggregating/#grouping-keys. Accessed: Jul 27, 2023.

[2] Comparing SQL to Cypher. https://neo4j.com/developer/cypher/guide-sql-to-cypher. Accessed: Jul 25, 2023.

[3] Creating a Modern Solution to Finding What's Lost with Neo4j. https://neo4j.com/case-studies/notlost. Accessed: May 4, 2023.

[4] Cypher clause composition. https://neo4j.com/docs/cypher-manual/current/clauses/clause_composition. Accessed: Jul 24, 2023.

[5] Cypher query language. https://neo4j.com/developer/cypher. Accessed: Jul 25, 2023.

[6] DB-Engines Graph DBMS Ranking. https://db-engines.com/en/ranking_trend/graph+dbms. Accessed: May 3, 2023.

[7] Faster Fraud Investigations with Neo4j. https://neo4j.com/case-studies/zurich-insurance. Accessed: May 4, 2023.

[8] How Orange uses Neo4j for IT Supervision and Network Security - Interview of Nicolas Rouyer. https://www.youtube.com/watch?v=aua9yzcENgI. Accessed: May 4, 2023.

[9] NBC News Analyzes Hundreds of Thousands of Russian Troll Tweets Using Neo4j. https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-case-study-NBC-News-EN-US.pdf. Accessed: May 3, 2023.

[10] Neo4j Enables Pulitzer Prize-Winning Investigation into Global Tax Evasion. https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-case-study-ICIJ-EN-US.pdf. Accessed: May 4, 2023.

[11] Novartis Captures the Latest Biological Knowledge for Drug Discovery. https://go.neo4j.com/rs/710-RRC-335/images/Neo4j-case-study-Novartis-EN-US.pdf. Accessed: May 4, 2023.

[12] OpenCypher. https://opencypher.org. Accessed: May 19, 2023.

[13] What is a denial of service attack (DoS) ? https://www.paloaltonetworks.com/cyberpedia/what-is-a-denial-of-service-attack-dos. Accessed: May 4, 2023.

[14] What is a GDBMS. https://database.guide/what-is-a-graph-database. Accessed: Jul 24, 2023.

[15] Who uses Neo4j. https://neo4j.com/who-uses-neo4j. Accessed: May 3, 2023.

[16] CVE-2018-1000820. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-1000820, 2018.

[17] CVE-2020-35668. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-35668, 2020.

[18] American Fuzzy Lop. https://github.com/google/AFL, 2023.

[19] Apache AGE. https://github.com/apache/age, 2023.

[20] Neo4j. https://github.com/neo4j/neo4j, 2023.

[21] RedisGraph. https://github.com/RedisGraph/RedisGraph, 2023.

[22] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "small scope hypothesis". 10 2002.

[23] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. Restler: Stateful rest api fuzzing. In *ICSE 2019*, November 2019.

[24] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing, 2022.

[25] Chris J Date and Hugh Darwen. *A Guide to the SQL Standard*, volume 3. Addison-Wesley New York, 1987.

[26] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. GREYONE: Data flow sensitive fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2577–2594. USENIX Association, August 2020.

[27] Zu-Ming Jiang. DynSQL: Stateful Fuzzing for Database Management Systems with Complex and Valid SQL Query Generation. 2023. https://www.usenix.org/system/files/sec23summer_60-jiang_zu_ming-prepub.pdf.

[28] Matteo Kamm. Testing graph databases using predicate partitioning. Master thesis, ETH Zurich, Zurich, 2022.

[29] Brian W Kernighan and Dennis M Ritchie. *The C programming language*. 2006.

[30] Wei Lin, Ziyue Hua, Luyao Ren, Zongyang Li, Lu Zhang, and Tao Xie. Gdsmith: Detecting bugs in graph database engines, 2022.

[31] Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[32] Rui Zhong, Yongheng Chen, Hong Hu, Hangfan Zhang, Wenke Lee, and Dinghao Wu. SQUIRREL: testing database management systems with language validity and coverage feedback. *CoRR*, abs/2006.02398, 2020.