Towards Generic Database Management System Fuzzing

Yupeng Yang*, Yongheng Chen*, Rui Zhong^, Jizhou Chen*, and Wenke Lee*

* Georgia Tech ^ paloalto NETWORKS

Background and Motivation

- Database Management Systems (DBMSs) are widely used for data storage, retrieval, and management.
- Both relational (SQL) DBMSs and non-relational (NoSQL) DBMSs have wide adoption in real world for the diverse requirements of various applications.

redis AgensGraph mongoDB PostgreSQL

KeyDB RedisGraph ArangoDB MySQL

......

The security and robustness of these prevalent and critical systems are vital!

Background and Motivation

- Fuzzing can be used to test software systems by injecting random inputs to them.
- Fuzzers targeting **SQL DBMSs** have proven useful and effective over the years.
  - SQLSmith, Squirrel, SQLancer...
- **NoSQL DBMSs** lack an effective fuzzing solution
  - Existing SQL DBMS fuzzers have challenges migrating to NoSQL DBMSs.
  - Generic fuzzers (e.g., AFL) struggle to generate valid inputs to DBMSs.

?

......

Challenges and Limitations

- We discover three major challenges when designing a fuzzer that extends to NoSQL DBMSs.

- C1: It is hard to generalize.
- C2: Semantics can change based on the context.
- C3: Loose data dependencies.

Georgia Tech

C1: It is hard to generalize

- Semantic correctness is vital for exploring deep DBMS logic.
- NoSQL DBMSs have ***diverse*** interfaces, and their semantics vary drastically.

DBMS	Input Format	Examples
redis	*key-value commands*	`HSET key field value [field value ...]`
AgensGraph	*ASCII-art (Cypher)*	`MATCH (p:person {name: 'Tom'})-[r:knows*1..2]->(f:person)` `RETURN f.name, r[1].fromdate;`
mongoDB®	*JSON documents*	`db.products.insertOne( { item: "card", qty: 15 } );`

C2: Semantics can change based on the context

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

Grammar

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

Parsing

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

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

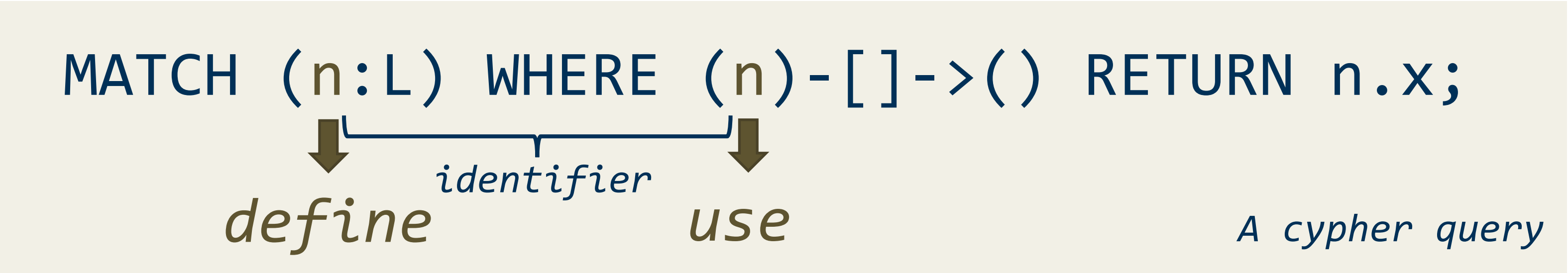
Static Constraint

- Existing works bind "static semantics" to the syntax structures.
- This works well for modeling common SQL semantics.

C2: Semantics can change based on the context

However, for NoSQL, semantics often change based on the context.

- One syntax structure can have different semantics in different syntactic contexts.




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

pattern_part:
  node_pattern;

where_part:
  WHERE node_pattern;

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

- Data types can depend on other values in the context.

C2: Semantics can change based on the context

However, for NoSQL, semantics often change based on the context.

- One syntax structure can have different semantics in different syntactic contexts.

- Data types can depend on other values in the context.

redis commands

```
> HSET k1 k1_field1 "Hello"
```

ASCII string — context

```
> HSET k2 k2_field1 "123"
```

Numeric string — context

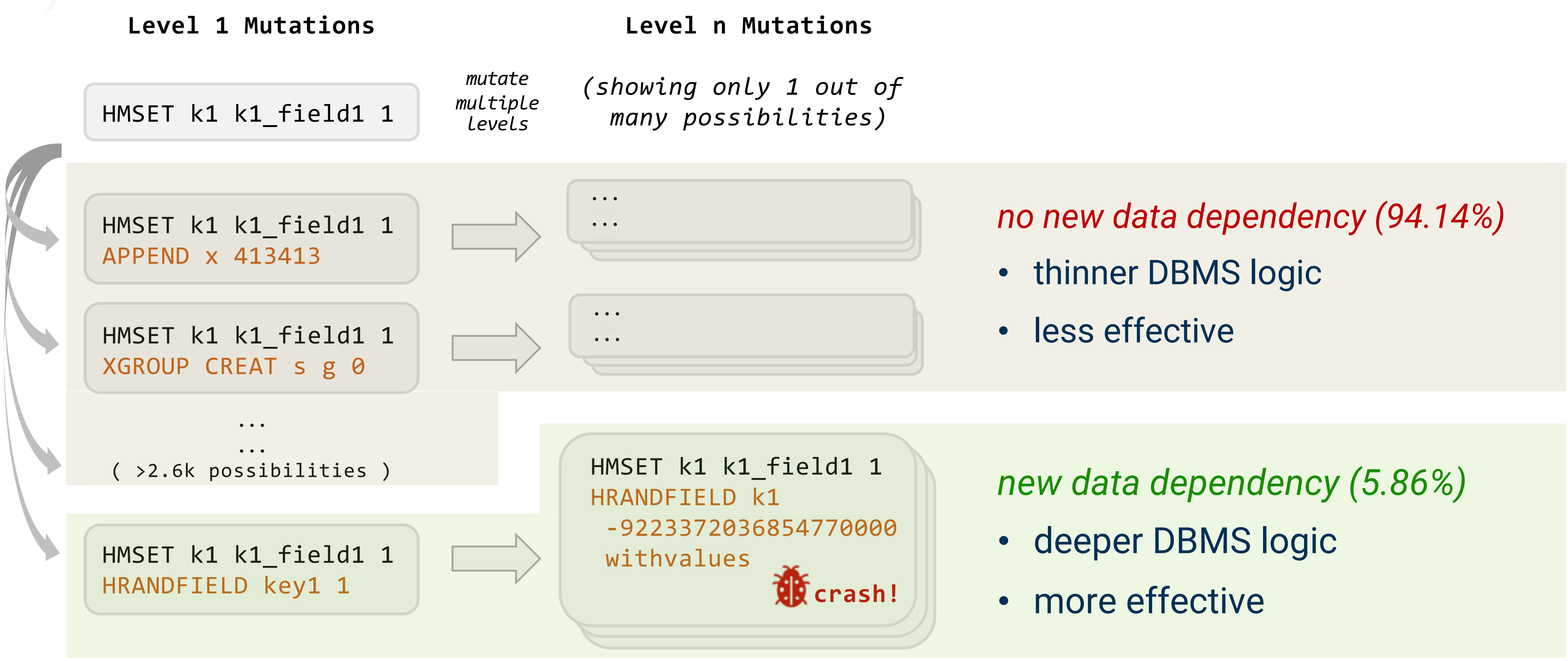
```
> HINCRBY k1 k1_field1 1
```

❌ *Only a numeric string is valid.*

```
(error) value not an integer
```

C3: Loose Data Dependencies

Random mutations tend to generate loose data dependencies.



Random Mutation Running Examples (for redis)

Our Solution

We propose three approaches to tackle the three challenges.

- *Semantics Abstraction*
  - C1: Non-generic
- *Context-sensitive Constraint Resolution*
  - C2: Context-based Semantics
- *Dependency-guided Mutation*
  - C3: Loose Data Dependency

We implemented our approaches into a generic fuzzing framework, BuzzBee, that can fuzz ***both*** SQL and NoSQL DBMSs effectively.

Semantics Abstraction -> C1

To generalize, we model common DBMS operations at a highly abstract level using three basic data operations: *Define*, *Use*, and *Invalidate*.

Next, we constrain the abstract semantics

- When to *Define*, *Use*, or *Invalidate* (scope constraints)
- What *type* to *Define*, *Use*, or *Invalidate* (type constraints)

Constraints:

The semantic rules to avoid a DBMS execution error.

We design an *Annotation System* to let users annotate the abstract semantics and constraints on the input grammar.

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

Input Grammar

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

scope constraint

type constraint

Annotation

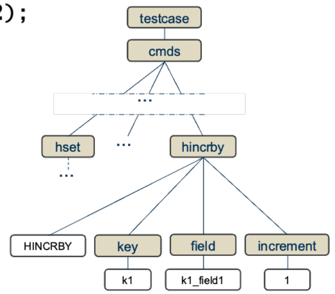
Semantics Abstraction - Internals

- We design an IR to carry the syntactic and semantic information specified by the user.
- We maintain scope trees and symbol tables to track the data.

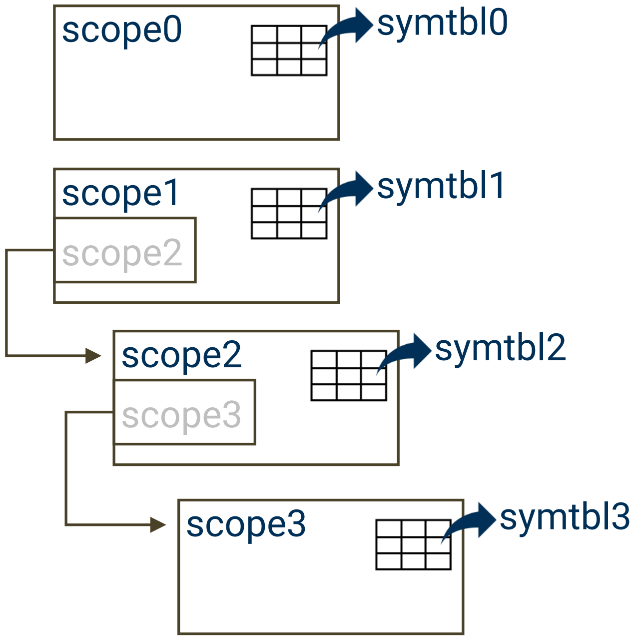
```
HSET k1 k1_field1 "Hello"
HSET k2 k2_field1 "123"
HSET k3 k3_field1 "456"
// DEL k1
HINCRBY k1 k1_field1 1
```

Test Case


```
node3(id=3, type=hset, text=NULL, annotation={},
      children={&node4, &node5, &node7, &node9}, parent=&node2);
...
node33(id=33, type=hincrby, text=NULL, annotation={},
      children={&node34, &node35, &node37, &node39},
      parent=&node32);
node34(id=34, type=terminal, text="HINCRBY", annotation={},
      children={}, parent=&node33);
node35(id=35, type=key, text=NULL,
      annotation={"default":{operation:Use,
                             args:{type:"HSET key"}}},
      children={&node36}, parent=&node33);
```



An IR Program



Scope Trees and Symbol Tables

Context-sensitive Constraint Resolution -> C2

To achieve context sensitivity, we design two features for the *Annotation System* so that users can specify constraints based on the context.

- *Context Query Language (CQL)* for simplicity – targeting common semantics
- *Custom Resolvers* for expressiveness – targeting complex semantics

Context Query Language (CQL)

- CQL is a lightweight language to fetch information from the context.
- To fetch certain information, we need to know:
  - ***where*** to fetch (which part of the context do we care about?)
  - ***what*** to fetch (what property of that part are we interested in?)

```
cql:
    navigator* property ;
```

where

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

```
arg:
    '(' num ')' ;
```

what

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

Grammar of CQL

```
.lsib(1)@text

.parent.rsib(1)@id

.parent.rsib(1).child(0)@id
```

CQL Examples

```
args: {
  type: "HSET numeric field
        of {.lsib(1)@text}"
}
```

CQL in the Annotation

Georgia Tech

Context Query Language (CQL)

```
> HSET k1 k1_field1 "Hello"
> HSET k2 k2_field1 "123"
> ...
> HINCRBY k1 k1_field1 1
```

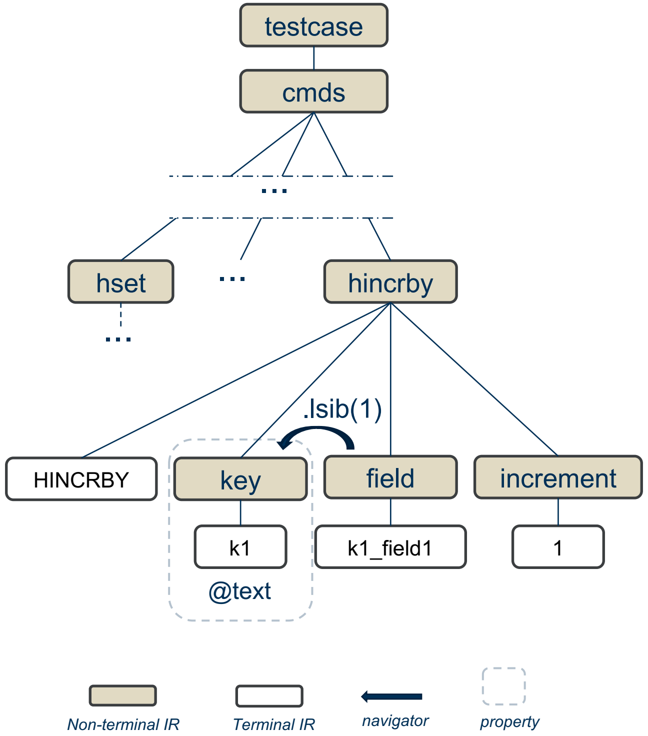
Redis Test Case

```
hincrby:
    HINCRBY key③ field④ increment;
```

Redis Grammar

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

CQL in the Annotation



CQL Querying Process

```
> HSET k1 k1_field1 "Hello"
> HSET k2 k2_field1 "123"
> ...
> HINCRBY k1 k1_field1 1
```

```
{
    operation: Use,
    args: {
        type: "HSET numeric field
               of k1"
    }
}
```

Resolved Constraint

Custom Resolvers

Custom Resolvers are plugins to the *Annotation System*.

- Can be written in high-level languages like C++.
- Have access to all the context information visible to BuzzBee.
- Can express arbitrarily complex semantics to complement CQL.

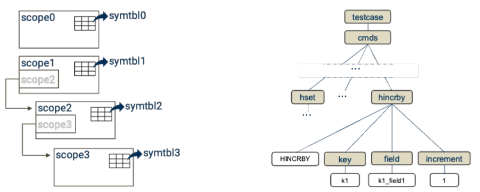
Custom Resolvers

```
hset:
    HSET key(1) (field(2) value)+;
```

Redis Grammar

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

Custom Resolver in Annotation



Symbols *IR Program*

```
hset_field_type_resolver
> ... (custom code)
```

Custom Resolver

type: HSET field of k1

```
> HSET k1 k1_field1 "Hello"
> HSET k2 k2_field1 "123"
```

type: HSET **numeric** field of k2

```
> ...
> HINCRBY k1 k1_field1 1
```

Redis Test Case

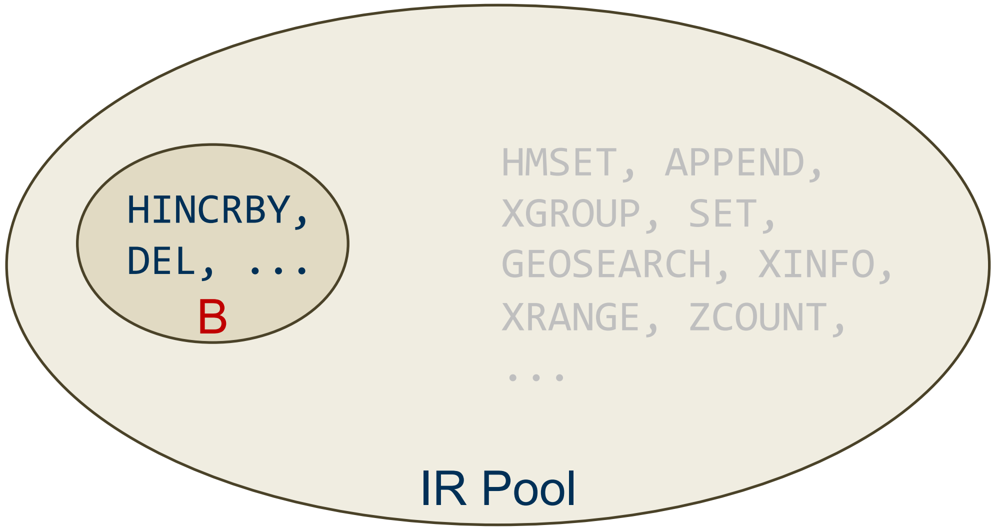
Georgia Tech

Dependency-guided Mutation -> C3

We add **guidance** to the *replacement* and *insertion* mutations.

```
> HSET k1 k1_field1 "Hello"
> HSET k2 k2_field1 "123"
> ...                         Mutation point A
> HINCRBY k1 k1_field1 1
```

1. Get all symbols available at A.
   - k1, k1_field1, k2, k2_field1
2. Favor B from the IR Pool, which can use the available symbols.

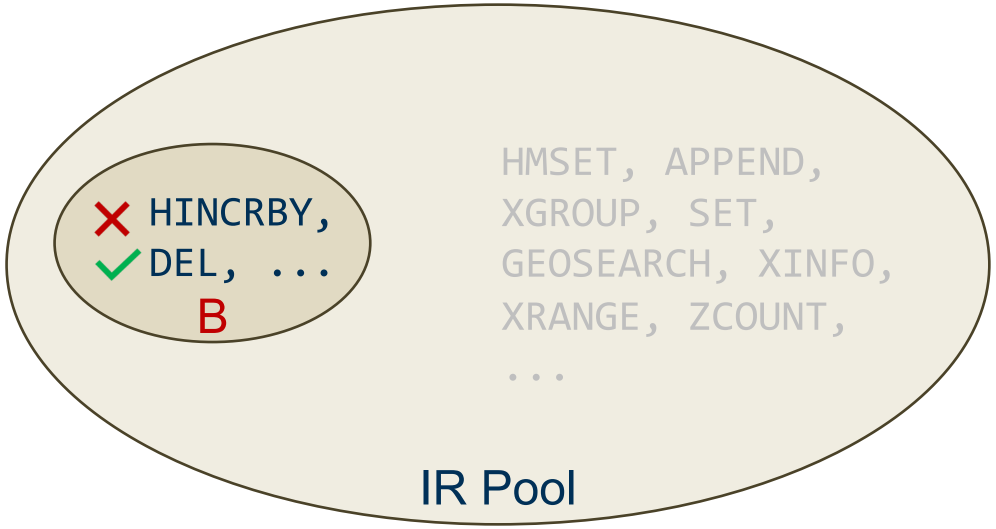


IR Pool stores the mutation candidate IRs.

Dependency-guided Mutation -> C3

We also introduce a finer-grained **prioritization** to cover more behaviors.

```
> HSET k1 k1_field1 "Hello"
> HSET k2 k2_field1 "123"
> ...                       Mutation point A
> HINCRBY k1 k1_field1 1
```



1. Get all symbols available at A.
   - k1, k1_field1, k2, k2_field1
2. Favor B from the IR Pool, which can use the available symbols.
3. Prioritizes IRs in B that do not exist in the test case.
   - DEL will be chosen over HINCRBY.

Georgia Tech

Implementation & Evaluation

- Implemented BuzzBee mainly in C++ and Python (9,130 LoC)

- Applied to **8** real-world DBMSs covering **4** major data models.
  - redis, KeyDB, RedisGraph, AgensGraph, MongoDB, ArangoDB, PostgreSQL, MySQL
- Discovered **40** bugs in the latest versions (with 4 CVEs).
- Outperformed generic fuzzers in NoSQL DBMSs
  - Up to 76.9% cov increase in NoSQL DBMSs
  - Discovered >30 bugs that generic fuzzers could not discover
- Achieved comparable results with SQL fuzzers
  - Achieved 92.7% cov of Squirrel
  - Found a similar # of bugs

Thanks / Q&A

Towards Generic Database Management System Fuzzing

Yupeng Yang*, Yongheng Chen*, Rui Zhong^, Jizhou Chen*, and Wenke Lee*

* Georgia Tech   ^ paloalto NETWORKS