SGL: A domain-specific language for large-scale analysis of open-source code

Darius Foo
SourceClear, Inc.
darius@sourceclear.com

Ang Ming Yi
SourceClear, Inc.
ming@sourceclear.com

Jason Yeo
SourceClear, Inc.
jason.yeo@sourceclear.com

Asankhaya Sharma
SourceClear, Inc.
asankhaya@sourceclear.com

***Abstract*—Today software is built in fundamentally different ways from how it was a decade ago. It is increasingly common for applications to be assembled out of open-source components, resulting in the use of large amounts of third-party code. This third-party code is a means for vulnerabilities to make their way downstream into applications. Recent vulnerabilities such as Heartbleed, FREAK SSL/TLS, GHOST, and the Equifax data breach (due to a flaw in Apache Struts) were ultimately caused by third-party components. We argue that an automated way to audit the open-source ecosystem, catalog existing vulnerabilities, and discover new flaws is essential to using open-source safely. To this end, we describe the Security Graph Language (SGL), a domain-specific language for analyzing graph-structured datasets of open-source code and cataloguing vulnerabilities. SGL allows users to express complex queries on relations between libraries and vulnerabilities in the style of a program analysis language. SGL queries double as an executable representation for vulnerabilities, allowing vulnerabilities to be automatically checked against a database and deduplicated using a canonical representation. We outline a novel optimization for SGL queries based on regular path query containment, improving query performance by up to 3 orders of magnitude. We also demonstrate the effectiveness of SGL in practice to find zero-day vulnerabilities by identifying several flaws in the open-source version of Oracle GlassFish Server.**

I. Introduction

The adoption of DevOps practices and sophisticated deployment tools have made it straightforward to release and consume software packages.

Developers assemble large applications using off-the-shelf open-source components and libraries, which are distributed through centralized repositories such as Maven Central, NPM, RubyGems, and PyPI. Much of the busywork of downloading sources and negotiating package versions is automated by dependency management tools like `npm`, `pip`, or `gem`. A single `install` command can pull in hundreds of libraries, demonstrating how easily large volumes of third-party code can be included in software projects.

There are many benefits to using open-source libraries: low cost, code reuse, and the flexibility to customize it to one's needs. However, unaudited third-party code is also a means for flaws and vulnerabilities to make their way downstream into applications. Recent vulnerabilities – Heartbleed [6], FREAK SSL/TLS [8], and GHOST [9] – were due to bugs in popular open-source libraries. The recent Equifax data breach [12], the largest in history, was also due to a known vulnerability in the Apache Struts web framework.

We believe that an automated way to audit the open-source ecosystem – by cataloguing existing vulnerabilities and bugs, as well as discovering new ones – is essential to being able to use open-source safely.

To this end, we describe the Security Graph Language (SGL), a domain-specific language (DSL) for analyzing graph-structured datasets of open-source code. SGL allows users to express complex queries on relations between libraries and vulnerabilities: for example, taking the transitive closure of a *calls* relation in a call graph to discover if method calls from a library reach a sink known to be associated with a given vulnerability.

SGL doubles as a vulnerability description language. We represent vulnerabilities intrinsically, as SGL *queries*. This allows them to be deduplicated based on the structural types of queries, providing a means of maintaining the quality of vulnerability datasets. This representation also allows vulnerabilities to be easily verified against a given dataset by executing them.

SGL is implemented by compilation to Gremlin, a graph traversal language from the Apache TinkerPop [24] project. This allows SGL to use any TinkerPop-based graph database as a back-end; our own implementation uses the DataStax Enterprise Graph database.

Our main technical contributions are:

- The implementation of a DSL capable of both describing vulnerabilities and representing complex queries on real-world vulnerability datasets.
- A strategy for vulnerability duplication checking based on computing structural types for queries, leading us to the idea of a canonical, executable representation for vulnerabilities.
- A novel means of optimizing special cases of SGL queries for runtime performance based on regular path query containment. This can improve query performance by up to 3 orders of magnitude.

This paper is structured as follows:

- Section II covers the problems SGL solves in greater depth and compares SGL with prior work.
- In Section III, we provide an overview of Gremlin.
- In Section IV, we describe the syntax and semantics of SGL.
- In Section V, we cover SGL's type system and how it supports canonical vulnerability representation.

- In Section VI, we describe how SGL queries may be rewritten and optimized.
- In Section VII we evaluate the performance of rewritten SGL queries and include a case study on identifying zero-day vulnerabilities in the open-source version of Oracle GlassFish Server.
- We conclude by discussing areas for future improvement in Section VIII.

II. Motivation

A. Large-scale program analysis and graph qeuries

SGL is a DSL for program analysis. It is meant to express relations involving open-source libraries, their file contents, and associated vulnerabilities. Examples of domain objects are files and file hashes (library-level), as well as methods, classes, and bytecode hashes (source-level). The full schema is shown in Figure 3.

Source-level granularity is required because vulnerabilities ultimately originate in programs, and an understanding of program semantics is required to accurately analyze their effects. For example, to determine if a software project inherits a vulnerability from an open-source library it uses, one might check if it calls a *vulnerable method* (i.e. a known sink identified by a vulnerability research team for that particular vulnerability). This entails building a call graph [1][2], possibly also performing a data-flow analysis. The purpose of doing this at scale is to discover zero-day vulnerabilities from inter-library relationships.

A useful representation of an entire open-source ecosystem, from its libraries to their methods, is likely too large to fit in memory. Our Java dataset contains data for 79M vertices and 582M edges, computed from 1.4M libraries and totalling about 76GB. This is why SGL is at heart a graph query language, intended to interact with databases of library code and vulnerabilities.

As a general-purpose graph query language, SGL's closest relative is Gremlin [24], from which it inherits its core semantics. As a declarative fragment of Gremlin, it is able to express path queries without joins, but is not Turing-complete. Like Gremlin, it operates at a lower level of abstraction than other graph query languages, like Cypher [14] (Neo4j), Datomic Datalog [10] (Datomic), and GraQL [19] (Grakn), and has limited facilities for aggregation.

As a DSL for program analysis, SGL exists in the same space as LogicBlox LogicQL [20], Semmle QL [18], PQL [23], and Soufflé [22]. Joern [27] is a similar effort which abstracts over source code with a *code property graph* [28] model, but relies on Gremlin directly. Unlike the Datalog-derived languages in this category, SGL is not capable of general computation, and the classes of queries it can express are limited to reachability queries, path enumeration, and simple aggregations like counting and projection. It also does not have as many built-in facilities specifically for program analysis (e.g. syntax for program representation), as it is currently focused on the more general domain of open-source security. We hope to grow the language in these two areas in future.

B. Canonical vulnerability representation

SGL is also intended to give vulnerabilities a canonical, executable representation. This gives it a secondary use as a vulnerability description language that works in a distributed setting.

Vulnerabilities are possible only under specific conditions, require specific versions of components, and have varying and relative severity. Without some form of identifier, it is difficult to pinpoint the particular vulnerability one is discussing and get it the attention it requires.

Centralized vulnerability databases (e.g. NVD) are the de facto means of cataloging flaws and bugs in software components. They assign nominal identifiers (e.g. CVE-2017-1234) to instances of vulnerabilities, giving researchers a way to refer to them.

The CVE format is not completely machine-readable. Components that a CVE pertains to are identified inconsistently, leading to false positives when trying to assess if a CVE is relevant to some real-world system.

DSLs meant to add structure to CVEs exist; OVAL [3] is perhaps the most widely-adopted standard for vulnerability description. It uses information derived from CVEs and CPEs to identify vulnerable system configurations. Other efforts in the space are OASIS AVDL [4], for describing web application exploits, and VuXML [5], which captures vulnerabilities in software packages for FreeBSD.

As a vulnerability description language, SGL is focused on the domain of open-source libraries and vulnerabilities, and on drawing insights from large datasets offline, instead of live systems. It is also generally terser (allowing interactive use; see Section II-C) and contains fewer auxiliary details (such as a component's vendor), but has enough to identify software components unambiguously. Where it does better than its competitors is in its support for automated verification and deduplication.

Every SGL description of a vulnerability is a *graph query*, which can be trivially tested by executing it against a database. This makes the ability to gather the data the only requirement for being able to gain insight from an SGL-described vulnerability.

Furthermore, in SGL, every vulnerability has a structural canonical form (see Section V-B). Anyone can specify a vulnerability, execute it against their own vulnerability dataset, and share it with others. Not relying on a surrogate key, such as a CVE-ID, allows this scheme to work in the absence of a centralized authority.

C. As a domain-specific language

SGL was designed for the domain of open-source security. Its target audience is security researchers, and it abstracts over many graph database implementation details that Gremlin exposes, requiring knowledge of only the property graph model and schema to use. It is also significantly terser than

other vulnerability description languages and is suitable for use in an interactive, exploratory setting. This is essential for researchers as a means of inspecting data for patterns.

TinkerPop 3 provides ways to create internal DSLs [13], but we chose to go with an external DSL so we could remove things from Gremlin (e.g. Turing-completeness), giving us a simpler language grounded in existing theory (i.e. with decidable containment) that would be easier to reason about. For example, we are able to reason abstractly about operations such as transitive closure without having to interpret imperative Gremlin constructs to do so (see Figures 1 and 2, and Section VI).

III. Gremlin

In this section, we give a brief introduction to the Gremlin graph query language before moving on to SGL; the interested reader is referred to [24] for in-depth coverage, and to [16] for examples of Gremlin queries.

Operationally, the Gremlin *implementation* may be viewed as a virtual machine specialized for graph traversal. The machine takes a list of instructions (collectively called the *traversal*) and a *property graph* as inputs, changing its state as it executes the instructions in order. The state of the machine is a bag of pointers to objects, typically graph vertices; one might compare it to the frontier of a graph traversal algorithm. Once the list of instructions is exhausted, the final state of the machine is taken to be the result.

The Gremlin *language* may be viewed as the instruction set of the machine. Each instruction is called a *step*.

The semantics of Gremlin may be characterized more abstractly as homomorphism-based, with bags of results [17]. It supports complex graph patterns *with joins* and is Turing-complete: in addition to describing navigational paths and aggregation, Gremlin steps may perform side effects and alter control flow.

Gremlin operates on *property graphs* [17]: disjoint sets of vertices and directed edges, where each vertex or edge carries a *label* ℓ and is associated with a number of *properties*, or key-value pairs. The property keys ρ are symbols from some alphabet P, and the values may be any data from some domain D. The alphabets of symbols L and P, given $\ell \in L$ and $\rho \subseteq P$, are disjoint.

The canonical implementation of Gremlin is for the Java Virtual Machine (JVM), where L and P are sets of strings, and D is the set of JVM primitives and objects.

The Gremlin language has no concrete syntax: it is an internal DSL by design. The canonical implementation uses Groovy method calls to create chains of steps.

Given a graph containing libraries, vulnerabilities, and methods, consider this Gremlin query, which counts methods of a version of Spring MVC associated with vulnerabilities:

```
g.V().hasLabel("library")
     .has("group", "org.springframework")
     .has("name", "spring-mvc")
     .has("version", "4.0.0.RELEASE")
     .out("has_method")
     .where(in("has_vulnerable_method"))
     .count()
```

`g` begins the query. The first step, `V()`, conceptually places pointers at all vertices in the graph. The `has` steps then filter the vertices pointed to just library vertices with the given property keys and values.

`out("has_method")` then moves the pointers to method vertices connected to the libraries by `has_method` outgoing edges. The state of the machine is now a bag of methods associated with Spring MVC.

We want to count methods which have an incoming edge from a vulnerability, but not actually traverse to the vulnerabilities. `where(in("has_vulnerable_method"))` does this, filtering the result further but not moving the remaining pointers.

Now that the pointers of the traversal are at all the vulnerable method vertices, we can apply the `count()` step.

IV. Syntax and Semantics

SGL works much like Gremlin, in that its programs are chains of steps, and it describes navigational queries [17] over property graphs.

Unlike Gremlin, it is a typed language: together, a label and property key set $\ell \times \rho$ may be seen as defining the *type* of a vertex or edge; this is further described in section V.

A. Syntax

SGL programs consist of a set of bindings naming sequences of steps, followed by a sequence of steps.

```
<sgl_prog> ::= [ <bindings> ] <steps>
<bindings> ::= <binding> <bindings> | <binding>
<binding> ::= 'let' <ident> '=' <steps> 'in'
```

`<ident>` is a Java identifier. A step syntactically resembles a C-family function call, and may either be a vertex or edge traversal step – which ensure that the traversal passes through the given vertex or edge – or a meta step: aggregations such as `count` and `select`, or higher-order steps like `union` and `where`. An identifer may also appear in place of a step, standing for the subprogram it is bound to.

```
<steps> ::= <step> <steps> | <step>
<step> ::= <vertex> | <edge>
         | <meta> | <ident>
<vertex> ::= <ident> '(' <args> ')'
<edge> ::= <edge_label> [ '*' ]
<meta> ::= <meta_name> [ '(' <args> ')' ]
```

Steps are parameterized by predicates which range over literals. We show only the < predicate for brevity. `<literal>` may be assumed to be either strings or integers for simplicity.

```
<predicate> ::= <literal> | '<' <literal>
<keyword> ::= <ident> ':'
<arg> ::= [ <keyword> ] ( <predicate> | <steps> )
<args> ::= <arg> ',' <args> | <arg>
```

The following SGL program queries for the presence of a CVE identifier for Apache NiFi.

```
let nifi = library('java',
  'org.apache.nifi',
  'nifi-web-ui', '1.0.0') in
nifi library_in_version_range
  version_range_in_vulnerability
  where(describes_cve cve('2017-7665'))
```

We name a vertex representing the NiFi library. The next line is a sequence of four steps. Composition of steps is written in a postfix syntax reminiscent of Gremlin's method calls: `library(...) count` is the equivalent of Gremlin's `V().hasLabel('library').count()`, or `count(library(...))` in C-family syntax.

The first step starts the query, putting a pointer at the `nifi` vertex. The second moves the pointer to the set of vertices representing NiFi's *version ranges* – CVEs often refer to multiple vulnerable versions of libraries, and these are reified as range vertices in SGL's schema.

The third step moves from version ranges to the vulnerabilities they appear in. The last filters away vulnerabilities which do not pertain to CVE-2017-7665: those which do not have an outgoing edge of type `describes_cve`, and those which do but do not have the right CVE-ID. Concatenation of steps here serves as a means of conjunction, as in Gremlin.

```
library(
  coord1: 'org.springframework',
  coord2: within('spring-web', 'spring-mvc'),
  version: < '5.0.0.RELEASE')
  union(
    identity, dependent_on*,
    embedded_in*) dedup count
```

Vertex steps support keyword arguments, allowing the elision of irrelevant arguments; this is analogous to leaving out `has` clauses in Gremlin. The `<` and `within` predicates are demonstrated, as are the meta steps `union` – which allows a traversal to branch depending on the given argument traversals – and `count` and `dedup`, which are analogous to their SQL counterparts. `identity` is the identity traversal.

Adding the Kleene star computes the transitive closure of an edge traversal step with the same source and destination types. Putting everything together, this query counts the implicit and explicit transitive dependencies of Spring MVC.

B. Semantics

We describe the semantics of SGL programs via translation to Gremlin. The translation begins by recursively expanding bindings and removing keyword arguments.

Bindings may be seen as names for subprograms, rather than for runtime values; as such, they do not allow the results of queries to depend on the results of other queries.

Predicates map to Gremlin predicates. Vertex steps map to Gremlin's `has` and `hasLabel` steps, and edge traversal steps map to `out` or `in`. Meta steps map to individual Gremlin steps.

SGL inherits Gremlin's homomorphism-based bag semantics, but *without joins*. It is *not* Turing-complete, featuring only transitive closure instead of unrestricted iteration, thus SGL steps describe only navigational paths and aggregation without side effects. SGL may be seen as a declarative fragment of Gremlin in the sense that every SGL expression is referentially transparent.

```
let spring = library('java',
  'org.springframework',
  'spring-webmvc',
  '4.3.8.RELEASE') in
spring dependsOn*
```

Fig. 1. Transitive SGL query

```
g.V()
 .hasLabel('library')
 .has('language', 'java')
 .has('group', 'org.springframework')
 .has('artifact', 'spring-webmvc')
 .has('version', '4.3.8.RELEASE')
 .repeat(out('depends_on').dedup())
 .emit()
 .dedup()
```

Fig. 2. Gremlin output for transitive query

Figures 1 and 2 shows an SGL query translated to Gremlin, including the translation of the Kleene star to imperative iteration.

V. Type system

A. Schema-based checking

A notable difference between SGL and Gremlin is that the former is typed and requires an explicit schema. Schema knowledge contributes to many of SGL's functions, including allowing elision of information that is implicit but unambiguous, error-checking, type-checking, and query rewriting and optimization. The schema we use for the domain of open-source libraries and vulnerabilities is shown in Figure 3.

We define the *type* of a particular vertex or edge to be the product of its label and property key set, $\ell \times \rho$. The type of a *query* is the set of vertex or edge types which occur in its result set.

An interesting subset of the typing rules for checking that SGL queries conform to a given schema is shown in Figure 4.

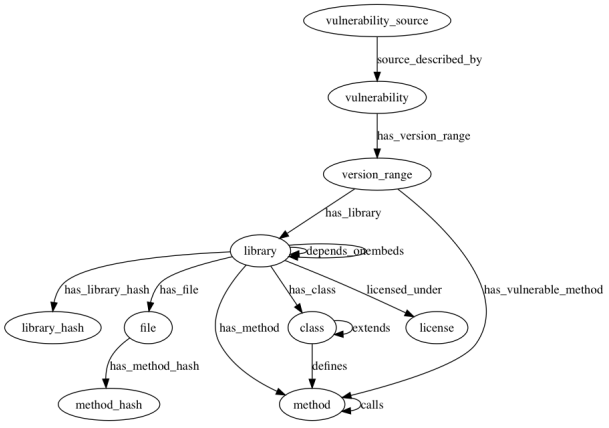

Fig. 3. Schema for open-source domain

$$\frac{t_1, ..., t_n : Type}{start : \{t_1, ..., t_n\}}\ [start] \qquad \frac{start : \{t, ...\} \quad v : \{t, ...\} \to \{t\}}{start\ v : \{t\}}\ [vertex]$$

$$\frac{v : \{t, ...\} \quad e : \{t, ...\} \to \{u\}}{v\ e : \{u\}}\ [traversal] \qquad \frac{v : \{...\} \quad count : \{...\} \to Int}{v\ count : Int}\ [count]$$

$$\frac{v : \{t, ...\} \quad e : \{t, ...\} \to \{u\} \quad where : \{t, ...\} \to (\{t, ...\} \to \{u\}) \to \{t, ...\}}{v\ where(e) : \{t, ...\}}\ [where]$$

$$\frac{\begin{array}{c} e_1 : \{t, ...\} \to \{u_1\}, ..., e_n : \{t, ...\} \to \{u_n\} \\ v : \{t, ...\} \quad union : \{t, ...\} \to [\{t, ...\} \to \{u_1\}, ..., \{t, ...\} \to \{u_n\}] \to \{u_1, ..., u_n\} \end{array}}{v\ union(e_1, ..., e_n) : \{u_1, ..., u_n\}}\ [union]$$

Fig. 4. Typing rules for a subset of SGL

```
vulnerability(cwe: 1)
  has_version_range union(
    version_range(from: '1.0', to: '1.1'))
  union(
    has_library union(
      library('java', 'web', 'core', '1.0'),
      library('java', 'web', 'core', '1.1')),
    has_vulnerable_method union(
      method('com/example/Controller',
        'config', '()')))
```

Fig. 5. Vulnerability query

Premises defining type variables like t and u are elided from all rules after *start*. $\{t, ...\}$ denotes an *open* set of types containing *at least* t.

B. Canonical vulnerability representations

As mentioned earlier, vulnerabilities in SGL are represented as queries, allowing them to be automatically checked by executing them.

Queries are given a structural type which doubles as a normal form for vulnerabilities, allowing them to be automatically deduplicated. This section describes these properties with an extended example.

Automated verification: Consider a simple XML external entity (XXE) vulnerability, where a web framework does not properly whitelist allowed inputs. A particular *version range* of that library is vulnerable, e.g. 1.0 to 1.1. We can describe it more precisely by listing the specific function or method responsible for not performing the desired validation, the *sink*; we can then assume that all code paths which go through that method trigger the vulnerability, modulo data flow. Having additional information is useful (e.g. severity, or a human-readable title), but this is the minimum required to identify instances of the vulnerability.

Assuming the existence of all this information in a graph, an SGL vulnerability is a subgraph containing the relevant libraries, methods, and version ranges. Vulnerability sources, such as a CVE-ID, are deliberately left out, as they do not intrinsically identify the vulnerability – they may still be present in the graph for queries, but do not contribute to the vulnerability's canonical representation.

We represent the vulnerability with the SGL query in Figure 5. This may be seen as an *extensional* description of the vulnerability. The `cwe` property contains its Common Weaknesses Enumeration (CWE) identifier, a taxonomy of software weaknesses and flaws.

While CWEs also come from a centralized source, the CWE list is versioned and added to far less frequently than CVEs are (with 31 versions in the past 8 years); it thus may be assumed to be static with respect to a given version of SGL. This ensures that changes to it do not bottleneck the addition of vulnerabilities expressed in SGL.

Executing the query representing a vulnerability would then be tantamount to verifying it – a non-empty result set indicating that the vulnerability and all its associated relationships are present. This is valid as long as the *graph* is correct; to ensure that, we build the graph in a reproducible way from publicly-available data. Additionally, the full expressiveness of SGL is available to perform further analysis on vulnerabilities once they are determined to exist.

Automated deduplication: Representing a vulnerability as a vertex with a single `cwe` property does not yet address the second requirement of deduplication. Since vulnerabilities are intrinsically defined by their surrounding subgraph of libraries and methods, we need at least some representation of that in the vertex in order to tell if it is different from another with the same CWE.

To deal with this, we compute a *structural type* for the query, serialize it, and store it as a property in the vulnerability vertex. These types are different from those mentioned in Section V-A (which classify queries by the *kinds of results* they produce) in that they classify queries by *structure*. That is, two queries have the same type iff their result sets are identical (and the vulnerability subgraphs they represent are isomorphic). This reduces the problem of deduplicating subgraphs in a graph database to that of checking if two SGL queries have the same structural type.

We compute the structural type of a vulnerability query as follows. For purposes of vulnerability description, we require that everything be specified explicitly, so we disallow *intensional* queries and use a subset of SGL with only the following constructs:

- Bindings
- `vulnerability`, `version_range`, `method`, `library`, `has_version_range`, `has_library`, `has_vulnerable_method` vertices and edges
- `eq` and `within` predicates
- Queries that begin at a `vulnerability` vertex

The query in Figure 5 satisfies these conditions.

We perform the same type and semantic checks as we do on standard SGL queries. To arrive at the structural type from the query, we expand certain syntactic elements and at the same time preserve semantic equality.

In this particular case, rendering the type in SGL syntax then gives us the same query in Figure 5. The type may also be thought of as a syntactic *normal form*, or as a canonical way to serialize the graph. It is rendered as a string and stored in the vertex property `query`. This gives researchers a decentralized representation and enables community-submitted vulnerabilities to be verified and stored.

One weakness with this scheme is that vulnerabilities represented as queries are based on a snapshot of the state of the ecosystem at a point in time. Library versions may generally be assumed to be immutable, but new versions of libraries being released would render existing version ranges incomplete. This implies that vulnerabilities must be maintained, as CVEs must, and may give rise to false negatives when automatically checked.

To solve this, one must ensure that SGL-described vulnerabilities are only taken to be complete when fixed versions of libraries are released, i.e. when all version ranges are closed. As all version ranges are listed explicitly, an SGL-described vulnerability should also be taken to be out of date if, on use, it does not contain a version of a library that one knows to exist. That would be cause for a review by a researcher.

VI. Query analysis and optimization

A. Rewriting queries

Being able to check if two queries are equivalent is essential to applications such as query rewriting and optimization. In this section, we describe a reduction of SGL to regular path queries (RPQ) [25] that enables us to rewrite queries for improved performance.

Preliminaries: We review the definitions of query containment and equivalence. The result set of a query q when executed against a database instance I is denoted $q(I)$.

Definition VI.1. Query q_1 is *contained* in q_2, denoted $q_1 \subseteq q_2$, if $\forall I.\, q_1(I) \subseteq q_2(I)$.

Query equivalence is formulated in terms of containment:

Definition VI.2. Two queries q_1 and q_2 are *equivalent*, denoted $q_1 \equiv q_2$, if $\forall I.\, q_1(I) \subseteq q_2(I) \wedge q_2(I) \subseteq q_1(I)$.

Regular path queries: Regular path queries are considered to be the basic querying mechanism for graph databases [17]. RQ is the set of regular path queries limited to the operations of selection, projection, conjunction, disjunction, and transitive closure. Query containment for RQ is 2EXPSPACE-complete [25] but decidable.

We describe how a subset of SGL is reduced to a subset of RQ, allowing the use of its containment algorithm.

The SGL subset we consider contains only vertex steps, edge traversal steps, and the meta step `where`, with vertex predicates. This is sufficient to express the pattern we wish to show equivalence for: queries with known starting and ending points, e.g. `library(...) dependent_on* has_method calls* method(...)`, where the ellipses stand for some nonempty set of properties with arbitrary predicates as values. We chose this subset because the pattern of trying to determine the path between sets of sources and sinks arises commonly in practice and is a good starting point for developing further optimizations.

This subset is exactly RQ without disjunction. `where` allows us to remove vertices from the final result set. We use the semantics of RQ (where a query's result is the set of vertices along the whole path traversed) for reasoning, but can easily recover Gremlin's semantics by selecting only the ending bag of vertices and adding `dedup`.

Consider a directed graph with vertices V, and two sets of vertices $v_s, v_e \subseteq V$. Suppose there exists some path of alternating edges and vertices $e_1 v_1 e_2 v_2 ... e_{n-1} v_{n-1} e_n$ between every vertex in $v_s \times v_e$. No matter the direction of the intermediate edges e, the intermediate vertices v remain the same along the path. Otherwise, v_s and v_e are unconnected, and the set v is empty. In either case, the set of vertices selected by a regular path query from v_s to v_e remains the same no matter the direction of the edges along the way. This implies that the result of a query is unchanged under reversal if the starting and ending points are known.

We now prove that a query is contained in its reversal. We do not consider the more general case of *folded* queries – queries for which edges followed by their inverse may be collapsed into identity edges – assuming that they do not occur in handwritten queries. Thus, we can make use of the RPQ containment algorithm from Section 3.2 of [25].

Proof. A regular path query q is isomorphic to a regular expression r_q over an alphabet of edge symbols. Since the starting and ending points of q are known, there is a function f_e mapping the strings generated by r_q to some set of concrete paths in the actual graph. This may be seen as a means of relating q to its result set $q(I)$.

Regular languages are closed under reversal, so $reverse(r_q)$ is also a regular expression, isomorphic to $reverse(q)$. Given that the edges of q are unchanged under reversal, f_e remains the same. Thus q and $reverse(q)$ have the same result set. □

Corollary VI.1. *Given queries q_1 and q_2 for which q_1's ending set q_{1e} agrees with q_2's starting set q_{2s}, the result set of their concatenation is the union of vertices in paths in $q_1(I)$ and $q_2(I)$ which pass through $q_{1e} \cup q_{2s}$. Thus the set of queries for which starting and ending points are known is closed under concatenation.*

Edge	Avg out-degree	Avg in-degree
depends_on	4.0	4.1
has_file	43.5	1.0
has_method	1508.2	8.9
calls	27.2	30.6
embeds	54.9	22.0
defines	14.4	1.8
has_library_hash	1.0	2.6
has_method_hash	4.9	18.6
has_library	16.4	1.9
has_vulnerable_method	1.8	2.1
has_version_range	2.9	1.2
has_class	217.0	11.1
extends	1.0	1.0

TABLE I
DEGREE CENTRALITY

We use these properties to rewrite queries for improved performance.

Definition VI.3. The *redundancy* of a query with known starting and ending sets v_s and v_e is the number of vertices reachable from v_s that do not eventually have a path to v_e without going back along an edge already traversed.

Intuitively, redundancy measures the number of dead-ended paths. Given a query and its reversal, we would prefer the query with the lower redundancy as it would execute faster and with lower memory usage.

To estimate redundancy, we consider different edge multiplicities:

- One-to-one edges: there are no examples of this in our schema, as one-to-one relationships between two vertices may be collapsed into a single vertex.
- Many-to-one edges: e.g. `has_file`, connecting `library` and `file` vertices. These may be seen as parent-child relationships. Traversing from a child to its parent has a redundancy of 0 and is always preferable to going in the other direction. Cardinality in the reverse direction may be up to 2 orders of magnitude higher, as observed in our Java dataset.
- Many-to-many edges: e.g. `depends_on`, connecting `library` vertices. For these edges, we approximate the redundancy in both directions using degree centrality. The direction with the lower cardinality will have lower redundancy, assuming that the same percentage of vertices traversed are redundant. In- and out-degree figures from our dataset are shown in Table I.

Thus, in cases where a query can be reversed and only contains the `calls` edge, `calls` is always preferable to `called_by` as the former has lower redundancy.

The total estimated redundancy is the product of the cardinalities of all the edges along the way. We use this to rewrite queries into their reverse if it would improve the estimate. This is a schema-specific optimization only doable with runtime profiling in Gremlin. We evaluate the performance improvements in Section VII-A.

Query	$estimate_o$	$estimate_r$	$runtime_o$	$runtime_r$
GlassFish readPath	391.2	55.7	105.8	0.6
HTTP Redirect	831.7	831.7	1.3	3.8
Spring-Jackson dependent_on*	4.1	4.0	4.6	0.3
BeanUtils injection sink	41015.8	271.8	$>$ 575.4	0.3

TABLE II
REWRITING BENCHMARKS

VII. EVALUATION

A. Query-rewriting benchmarks

Consider the following query, which checks if there is a path from a GlassFish class to the `File getPath` method.

```
let glassfish_class =
  class(regex 'org.glassfish.*') in
let read_object =
  method(method_name:'readObject') in
let get_path = method(
    class_name:'java/io/File',
    method_name:'getPath') in
glassfish_class defines
  read_object where(calls get_path)
```

This query has the key property which makes it amenable to optimization: it is the conjunction of two reversible queries, which means it is also reversible. The reverse of the query is as follows.

```
get_path called_by read_object
  where(defined_by glassfish_class)
```

From Table I, the estimated redundancies of the query in both directions are 55.7 and 391.2. This indicates that its reverse should execute faster. In fact, the original query executes in 105.8s on average, while its reverse executes in 0.6s.

Other queries we benchmarked are shown in Table II. The subscripts *o* and *r* stand for "original" and "reversed". Rewriting queries in this way can reduce running time by up to 3 orders of magnitude. The biggest savings come from eliminating many-to-one edges. When estimated redundancies are close, however, the rewrites are not nearly as effective. The redundancy estimate also does not contain meaningful information when a query is equal to its reversal. The HTTP Redirect example illustrates this, its edges being `calls called_by`.

B. Case study: GlassFish zero-days

Much of SGL's utility comes from being able to find new, *zero-day* vulnerabilities. In this section, we present 2 specific findings from analysis of the open-source edition of Oracle GlassFish Server. We verified and disclosed a total of 23 issues to Oracle, which have since been patched.

Security Manager Bypass: CVE-2016-0763 is a vulnerability reported on the Apache Tomcat Application Server. An SGL query that finds similar issues in other servers is as follows.

```
let set_global_context = method(
  class_name:'org/apache/naming/factory/
    ResourceLinkFactory',
  method_name:'setGlobalContext') in
let get_security_manager = method(
  class_name:'java/lang/System',
  method_name:'getSecurityManager') in
let check_permission = method(
  class_name:'java/lang/SecurityManager',
  method_name:'checkPermission') in
set_global_context called_by
  not(union(
    calls get_security_manager,
    calls check_permission))
  method_in_library
```

Executing this query returns all affected libraries, which included GlassFish Open Source Edition versions 3.0-4.1.2, and Oracle GlassFish Server 3.0.1-3.1.2.2 before Critical Patch Update (CPU) - October 2017 [15]. Following our disclosure, Oracle issued a new CVE for Oracle GlassFish Server, CVE-2017-10385.

Denial of Service (DoS) via File Upload Requests: CVE-2016-3092 is a vulnerability reported on the Apache Commons FileUpload library. The following is an SGL query that looks for usages of `MultipartStream` which do not throw `IllegalArgumentException`.

```
let multipartstream_init = method(
  class_name:'org/apache/catalina/
    fileupload/MultipartStream',
  method_name:'<init>',
  descriptor:'(Ljava/io/InputStream;
    [BILorg/apache/catalina/
    fileupload/MultipartStream$
    ProgressNotifier;)') in
let illegal_argument_exception = method(
  class_name:
    'java/lang/IllegalArgumentException',
  method_name:'<init>',
  descriptor:'(Ljava/lang/String;)' in
multipartstream_init
  not(calls illegal_argument_exception)
```

We verified that GlassFish Open Source Edition 3.0-4.1.2 was affected by this issue. It was fixed in 5.0 and no CVE was issued.

VIII. Conclusion

We presented SGL, a graph query language specialized for large-scale program analysis. SGL doubles as an vulnerability description language, supporting automated checking and deduplication of vulnerabilities represented in it. We gave an overview of its syntax and semantics, then covered its type system and how that contributes to canonical representations for vulnerabilities. We also explained optimizations on reversible SGL queries, allowing queries to execute up to three orders of magnitude faster. Finally, we described the zero-day vulnerabilities in the open-source edition of Oracle GlassFish Server that we found with it.

In future, we intend to work on broadening the space of graph queries and aggregations expressible in SGL, as well as on improvements to its facilities for program analysis. There is also room for additional query optimization and rewrites: in particular, indexing strategies for fast reachability queries [11] and optimizations to the underlying TinkerPop implementation.

References

[1] Dean, Jeffrey, David Grove, and Craig Chambers. "Optimization of object-oriented programs using static class hierarchy analysis." European Conference on Object-Oriented Programming. Springer, Berlin, Heidelberg, 1995.

[2] Sundaresan, Vijay, et al. Practical virtual method call resolution for Java. Vol. 35. No. 10. ACM, 2000.

[3] Oval - open vulnerability and assessment language, 2002.

[4] Oasis application vulnerability description language (avdl) tc — oasis, 2003.

[5] Vuxml, 2004.

[6] Heartbleed, 2014.

[7] Findbugs, 2015.

[8] Freak ssl/tls vulnerability - us-cert, 2015.

[9] Ghost: glibc vulnerability (cve-2015-0235), 2015.

[10] Datomic - home, 2016.

[11] Veloso, Renł Rodrigues, et al. "Reachability Queries in Very Large Graphs: A Fast Refined Online Search Approach." EDBT. 2014.

[12] Equifax: Cybersecurity incident & important consumer information, 2017.

[13] Gremlin dsls in java with dse graph, 2017.

[14] opencypher, 2017.

[15] Oracle critical patch update advisory - october 2017, 2017.

[16] Tinkerpop 3 documentation, 2017.

[17] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. Foundations of modern graph query languages. *CoRR*, abs/1610.06264, 2016.

[18] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. Ql: Object-oriented queries on relational data. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[19] D. Chavarra-Miranda, V. G. Castellana, A. Morari, D. Haglin, and J. Feo. Graql: A query language for high-performance attributed graph databases. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1453–1462, May 2016.

[20] Terry Halpin and Spencer Rugaber. *LogiQL: A Query Language for Smart Databases*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2014.

[21] Leonid Libkin and Domagoj Vrgoč. Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, pages 74–85, New York, NY, USA, 2012. ACM.

[22] Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On Synthesis of Program Analyzers In *CAV*, 2016.

[23] Michael Martin, Benjamin Livshits, and Monica S Lam. Finding application errors and security flaws using pql: a program query language. In *ACM SIGPLAN Notices*, volume 40, pages 365–383. ACM, 2005.

[24] Marko A. Rodriguez. The gremlin graph traversal machine and language. *CoRR*, abs/1508.03843, 2015.

[25] Moshe Y. Vardi. A theory of regular queries. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '16, pages 1–9, New York, NY, USA, 2016. ACM.

[26] Peter T. Wood. Query languages for graph databases. *SIGMOD Rec.*, 41(1):50–60, April 2012.

[27] Fabian Yamaguchi. Pattern-based vulnerability discovery. 2015.

[28] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 590–604, Washington, DC, USA, 2014. IEEE Computer Society.

[29] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.