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

In recent years, the number of open-source components used by developers to build software has seen immense growth. Millions of open-source libraries are distributed through centralized systems like Maven Central (Java), NPM (JavaScript), and GitHub (Go), and their widespread use means that bugs and vulnerabilities impact large numbers of downstream applications.

Discovering and cataloging such vulnerabilities across entire language ecosystems is a challenge not adequately addressed by existing vulnerability description languages. In this paper we describe the Security Graph Language (SGL), a domain-specific language meant to aid security engineers and software developers in finding bugs and discovering vulnerabilities. SGL supports transparently switching between transactional (OLTP) and analytical (OLAP) modes to enable analysis of large datasets. We implement structural typing for vulnerabilities in SGL that reduces the problem of vulnerability deduplication to type equivalence checking. SGL queries are optimized using query containment and compiled to Gremlin, which enables a variety of graph databases to be used as back-ends. We present several applications of SGL in bug-finding, impact analysis, and suggesting fixes for vulnerabilities introduced via transitive libraries. We also present a case study of GlassFish server where we found 23 zero-day vulnerabilities using SGL. Based on our survey of security researchers and software engineers, over 70% of the security researchers said that they would prefer SGL over OVAL for the task of describing vulnerabilities.

1 INTRODUCTION

Over the past few years, the way we build software has fundamentally changed. The rise of DevOps and use of automation in deployments has made it very cheap and easy to release new versions of software. These days, developers assemble large applications using off-the-shelf components and libraries, which are distributed through centralized repositories, such as Maven Central, NPM, RubyGems, PyPI, and Packagist.

Developers consume these libraries via dependency management tools, like `npm`, `pip`, or `gem`, which automate the busywork of downloading and wrangling dependencies. A single `install` command can download hundreds of libraries, which speaks to the volume of third-party code that is used today, and ease with which it is included in projects.

There are many benefits to using open-source code, among them code reuse, low cost, and the flexibility to customize it one's needs. However, unaudited third-party code is also a means for flaws and vulnerabilities to make their way downstream into applications. Several recent vulnerabilities like Heartbleed [5], FREAK SSL/TLS [7] and GHOST [8] were due to bugs in popular open-source libraries. In addition, the recent Equifax data breach [14] (the largest in history) was due to a known vulnerability in the Apache Struts library.

Discovering and cataloging bugs in open-source libraries at scale is a challenging problem, and one not addressed by existing vulnerability description languages such as OVAL [1]. Granular and automated ways to make sense of the open-source ecosystem are required. To this end, we describe SGL (Security Graph Language), a domain-specific language (DSL) for analyzing large datasets of open-source code and vulnerabilities. It is implemented by compilation to Gremlin, a graph traversal language originating from the Apache TinkerPop project [28]. This allows SGL to use any TinkerPop-based Graph database as a back-end; our own implementation is built on top of DataStax Enterprise Graph. SGL allows users to express complex queries on relations between libraries and vulnerabilities.

Our main contributions are:

- The implementation of an executable DSL capable of both describing vulnerabilities and representing complex queries on vulnerability datasets. In Section 2, we describe the syntax and semantics of the language, and in Section 3, its type system.
- We describe a strategy for vulnerability duplication checking based on structural typing of vulnerability vertices in Section 3. This prevents users from creating duplicate vulnerabilities if they have the same underlying cause (based on an SGL query), providing a means of scalably detecting errors and maintaining quality of vulnerability datasets.
- Automatic switching between OLTP and OLAP modes, as described in Section 4, allows users to write analyses that run consistently and efficiently over large graphs, without requiring familiarity with how they execute. In Section 6, we show from our experiments that OLAP processing is required in order to answer several important queries.
- Optimization of SGL queries based on query containment is done to improve performance. In Section 4 we explain how prior work on regular path query containment is used to rewrite special cases of SGL queries. As shown in our experience in Section 6, this can improve query performance by up to 3 orders of magnitude.

- SGL has several applications for security engineers and software developers. In Section 5 we present examples of SGL queries that automate tasks like finding similar vulnerabilities, estimating the impact of a vulnerability, suggesting fix versions of transitive vulnerabilities, and doing static analysis to find bugs. We also present a case study of GlassFish server where we found several zero-day vulnerabilities using SGL.
- In Section 6, we discuss the results of a survey of 36 security researchers and software engineers which shows that they would prefer SGL for creating vulnerability reports.

2 SYNTAX AND SEMANTICS

SGL is a domain-specific language (DSL) designed for the analysis of large graph-structured datasets, particularly for the domain of open-source security. It operates over a *property graph* [22]: 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. In our case the keys are symbols from some alphabet ρ and the values are JVM primitives or objects. Thus, we have two disjoint alphabets of symbols L and P such that $\ell \in L$ and $\rho \subseteq P$.

Together, a label and property key set $\ell \times \rho$ may be seen as defining the *type* of a vertex or edge; we elaborate on that in section 3.

SGL is stratified into two sub-languages (analogous to SQL's DML and DDL):

- Security Graph Description Language (SGDL) describes operations on graphs
- Security Graph Query Language (SGQL) describes navigational queries [22] (or traversals) on graphs

We give an informal overview of the important syntactic elements of both sub-languages here.

2.1 Syntax

SGDL. SGDL is a simple imperative language whose programs consist of:

- A set of bindings naming vertices
- A sequence of actions

The following is an example of an SGDL program.

```
let spring = library('java', 'org.springframework',
  'spring-webmvc', '4.0.0.RELEASE')
let oro = library('java', 'oro', 'oro', '2.0.7')
let size = method('org/apache/oro/util/GenericCache',
  'size', '()') in
add spring depends_on oro;
add oro has_method size;
remove spring depends_on oro;
remove oro has_method size
```

It first binds names to vertices representing the `spring` web framework, the regular expression library `oro`, and one of `oro`'s methods.

A couple of imperative `add` actions follow. When executed, they mutate the graph, inserting the three vertices and two edges between them. Edges are written infix and labeled depending on the types of vertices they connect; the edge between library vertices has the label `depends_on`, and the one between libraries and methods has `has_method`.

Finally, there are the `remove` actions, which undo the effects of the `add` actions by deleting the path between the vertices.

SGQL. SGQL is a much richer language. Its programs consist of:

- A set of bindings naming intermediate sequences of *steps*
- A sequence of steps

Consider the following example SGQL program. It queries for the existence of a CVE 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'))
```

As with the earlier example, the first line binds the name `nifi` to the vertex representing the Apache library. The next line is a sequence of four steps. The first indicates the starting point of the navigational query: the vertex representing the library `nifi`.

The second is a *traversal* step named after the edge connecting the `library` vertices to version ranges. CVEs often refer to vulnerable versions of libraries, so these ranges are represented as vertices in SGL's schema. Traversal steps like this may be viewed as functions mapping some set of vertices to another set of vertices, describing a single hop between them. One might think of the program `nifi` containing the single vertex named above, and the program `nifi library_in_version_range` as containing all the version ranges the `nifi` vertex appears in.

The third step maps version ranges to the vulnerabilities they appear in.

The last step is a *meta* step: a step which takes other steps as arguments. The `where` step has the effect of filtering the vertices for which the traversal has a non-empty result set. The arguments continue the traversal, going from the vulnerability to the CVE, and using the `cve` *vertex* step to constrain the result of the query, so only vertices matching the predicates it specifies will be mapped to. In this case, the only predicate is equality on the `identity` property. Thus, this program will return a result only if the `nifi` library is affected by CVE-2017-7665.

Steps are first-class in SGL, so we can also write the following equivalent program:

```
let has_vulnerability =
  library_in_version_range
  version_range_in_vulnerability in
nifi has_vulnerability
  where(describes_cve cve('2017-7665'))
```

This defines a new step, `is_vulnerable_to`, in a style reminiscent of point-free function composition. `is_vulnerable_to` can then be used to map libraries directly to vulnerabilities.

Now, consider the following program that illustrates the remaining syntactic elements of SGQL.

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

Vertex steps (like `library`) support keyword arguments (which allow us to elide arguments; notice `java` is not specified in the `library` vertex here). They also support predicates other than

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

Figure 1: Transitive SGL query

equality, like `within` (which is analogous to SQL's `in`) and `<` (which compares elements lexicographically). `union` is a meta step which allows a traversal to branch depending on the given argument traversals, and *aggregation* steps (such as `count` and `select`, which are analogous to their SQL counterparts, and `dedup`, which is `distinct`).

Traversal steps may also be recursively executed if they map a domain to itself: adding the Kleene star allows us to compute their transitive closure. This query uses all of these elements to count the number of implicit and explicit transitive dependencies of the library.

2.2 Semantics

In this section we describe the semantics of SGL programs via translation to Gremlin. A formal overview of Gremlin is given in [28], and examples of what individual steps do are given in [21].

Due to SGDL's simplicity, it is straightforward to give it a semantics based on graph mutations. Syntactically, we are limited to only being able to `add` and `remove` individual vertices and edges. Operations are transactional and idempotent (depending on the properties vertices are keyed by in the schema).

We focus on SGQL in the remainder of this section. To translate SGQL programs, syntactic sugar is first translated. Bindings are recursively expanded, failing on cyclic references. This is possible because SGQL is a pure language and is referentially transparent. 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.

A secondary function of bindings is that they mark program fragments with Gremlin's `as` step, which allows them to be referenced in a controlled way by other steps, such as `select`. Keyword arguments, like bindings, are compiled away, and predicates map trivially. Vertex steps map to Gremlin `hasLabel` steps, and traversal steps map to `out` or `in`. Meta and aggregate steps map trivially for the most part, modulo syntactic differences – SGQL does not have Gremlin's *modulation steps* and represents each step atomically.

SGQL thus inherits Gremlin's homomorphism-based bag semantics for results [22]. It supports complex graph patterns *without joins* and is not Turing-complete, featuring only transitive closure instead of unrestricted iteration; this gives it a subset of Gremlin's expressiveness. Gremlin steps may perform side effects, bind names, alter control flow, and so on, but SGQL steps only describe navigational paths and aggregation. Figures 1 and 2 illustrate how SGL queries are translated to Gremlin. The single step `depends_on*` is expanded using Gremlin's constructs for iteration.

3 TYPE SYSTEM

3.1 Schema-based checking

A notable difference between SGL and Gremlin is that the former requires an explicit schema. Schema knowledge contributes

```
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()
```

Figure 2: Gremlin output for transitive query

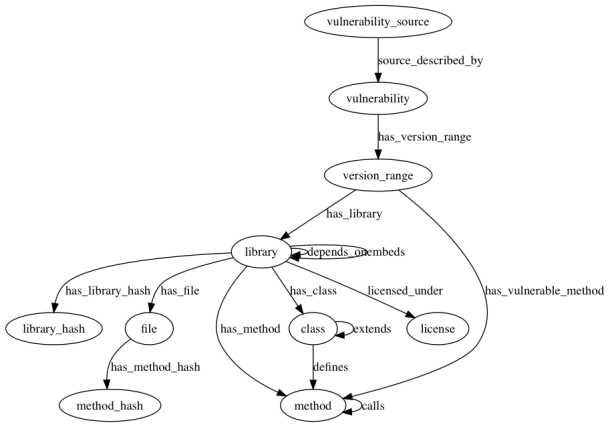

Figure 3: Schema for open-source domain

to many of SGL's functions, including: allowing users to leave out 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 combination of its label and property key set, $\ell \times \rho$. The type of a *query* is defined as the set of vertex or edge types which occur in its result set. Like Gremlin queries, SGL queries may return results of multiple types; we do not restrict the expressiveness of Gremlin here.

An interesting subset of the typing rules for checking that SGL queries conform to a given schema is shown in Figure 4. We assume the presence of an implicit *start* symbol at the beginning of every SGL query. This corresponds to Gremlin's `V()`, representing all graph vertices, and has as its type the set of all vertex types. We indicate a set of types with curly braces, and write function application backwards, in syntax close to Gremlin's: `start count dedup` is the equivalent of Gremlin's `V().count().dedup()`, or `dedup(count(start))` in C-family syntax.

The *vertex* rule for vertex steps builds on this: if t is the type of the vertex represented by the vertex *step* v, v may be seen as a function from some set of types containing at least t to a singleton set of just t. In other words, types which aren't t are filtered away, as Gremlin's `V().hasLabel(...)` would do. The ellipsis within sets of types denotes a *row variable* [30] which may be instantiated to any number of concrete types; ellipses outside have their usual meaning of omission.

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

$$\frac{a : \{t, ...\} \quad edge : \{t, ...\} \to \{u, ...\}}{a\ edge : \{u, ...\}}\ [traversal] \qquad \frac{a : \{...\} \quad count : \{...\} \to Int}{a\ count : Int}\ [aggregation(count)]$$

$$\frac{a : \{t, ...\} \quad b : \{t, ...\} \to \{u, ...\} \quad where : \{t, ...\} \to (\{t, ...\} \to \{u, ...\}) \to \{t, ...\}}{a\ where(b) : \{t, ...\}}\ [meta(where)]$$

$$\frac{a : \{t, ...\} \quad x_1 : u_1, ..., x_n : u_n \quad union : \{t, ...\} \to [\{t, ...\} \to \{u_1, ...\}, ..., \{t, ...\} \to \{u_n, ...\}] \to \{u_1, ..., u_n\}}{a\ union(b) : \{u_1, ..., u_n\}}\ [meta(union)]$$

Figure 4: Typing rules for a subset of SGL

The *traversal* and *aggregation* rules express that both types of steps are functions from their source vertex type to some destination type: either another vertex type, or in the case of `count`, integers. Other aggregation rules are elided. The *where* rule expresses that the argument to the `where` step must have the same domain type as the result of the preceding step, and that the type of the preceding step passes through unchanged. The *union* rule shows that steps of multiple types may all be applied to some bag of vertices as long as the source types of the steps all appear; with the result type being the union of all the destination types.

3.2 Canonical vulnerability representations

One of the use cases that motivated SGL's creation was to have a method for describing *vulnerabilities* that would work in a distributed setting. Centralized vulnerability databases, such as NVD/CVE, are the de facto means of cataloging and describing flaws and bugs in software components. They work by assigning identifiers to instances of vulnerabilities, such as `CVE-2017-1234`, giving researchers a canonical means to reference them.

However, centralization has many flaws. The creation and assignment of CVE-IDs is bottlenecked by having to go through only a handful of numbering authorities. Thousands of vulnerabilities went without ID assignments in 2015 [10], and as much as 55.6% of SourceClear [20]'s vulnerability data lacks a CVE-ID. Vulnerabilities are complex: they 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.

CVEs also suffer from a lack of granularity. They inherit this from another standard, CPE [4], which assigns identifiers to software components and libraries. The unit of a CPE is not well-defined and may be an application or library (`curl` appears, but `libcurl` doesn't; both MySQL and MySQL's `Connector/J` appear), along with auxiliary information such as its vendor and the platforms it runs on. This usually isn't sufficient to pinpoint which subcomponents are vulnerable, and leads to imprecision and false positives when trying to assess if a CVE is relevant to some real-world system.

The CVE itself typically contains a more specific description of the affected components, which brings us to the next issue: the CVE format is not completely machine-readable. CVE status are documented in semi-structured plain text notes, and CPEs do not always corresponding to software that can be mechanically

```
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', '()')))
```

Figure 5: Vulnerability query

installed by some standardized package manager. Thus it is difficult to machine-audit or verify CVEs, and tasks like deduplication and ensuring that cross-references are correct must be done manually.

There has been a proliferation of standards meant to address this issue, including OVAL [1], Oasis AVDL [2], and VuXML [3]. We discuss them in Section 8. To address the problem of a machine-readable standard for describing *vulnerabilities*, SGL supports two key operations - automatic verification and deduplication. Moreover, deduplication does not depend on the use of a *surrogate key* such as an identifier from a centralized numbering authority (CVE).

Automatic verification. As a motivating example consider a simple XXE vulnerability where a web framework does not properly whitelist allowed inputs. And, thus, 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; we can then assume that all code paths which go through that method trigger the vulnerability, depending on data flow. Having additional information is useful (such as severity, or a human-readable title), but this is the minimum one would require to identify instances of the vulnerability given the source code.

We define an SGL *vulnerability* to be a subgraph containing all of this information: libraries, methods, 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 (rendered pictorially in Figure 6 with the schema in Figure 3). This may be seen as an *extensional* description of the vulnerability. The `cwe` property contains its CWE identifier [12], a taxonomy of software weaknesses and flaws.

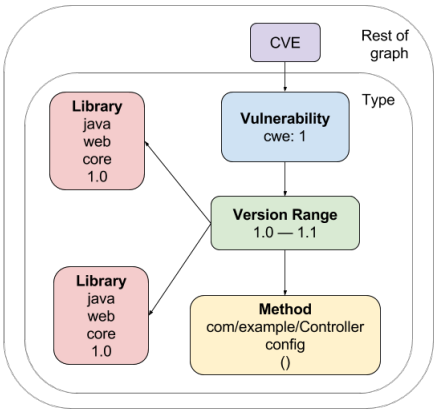

Figure 6: Vulnerability subgraph

We can now insert vulnerabilities into the graph database and perform queries which use it. Executing the query representing the vulnerability would be tantamount to verifying it – a non-empty result set would indicate that the vulnerability and all its associated relationships are correct. 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, we have the full expressiveness of SGL available to perform further analysis on vulnerabilities, for example: traversing to vulnerable methods and finding their callers in the hope of identifying similar vulnerabilities. Several such uses cases are given in Section 5.1.

Automatic deduplication. Representing this 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* τ for the query, serialize it, and store it as a property in the vulnerability vertex. These types are different from those mentioned in Section 3.1 (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 τ if and only if 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 outline an algorithm for computing the structural type of a vulnerability query. For purposes of vulnerability description, we require that everything be specified explicitly, so we disallow *intensional* queries and use a subset of SGL with the following restrictions:

- Bindings are allowed
- Only the required vertices and edges are allowed: `vulnerability`, `version_range`, `method`, `library`, `has_version_range`, `has_library`, `has_vulnerable_method`
- Only the predicates `eq` and `within` are allowed
- Queries must begin at a `vulnerability` vertex

The query in Figure 5 satisfies these requirements.

We perform the same type and semantic checks as we do on standard SGL queries.

We then compute a structural type for the query, which is a property graph with the same schema. It has exactly the same structure as the query, and each vertex in it is indexed by the property values of the corresponding vertices in the query.

This process can be implemented as a series of syntactic transformations:

- Bindings are expanded
- Values in `within` predicates are expanded into their cross product: for example, `library(within('a', 'b'), within('c', 'd'))` becomes 4 library clauses
- Keyword arguments are used for all arguments so ordering does not matter
- All arguments are given in lexicographical sorted order

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.

The final query is rendered as a string and stored in the vertex property `query`. As it is the canonical way to represent vulnerabilities, string equality is sufficient to tell if two vulnerabilities are equivalent. This gives researchers a decentralized identifier to use and enables community-submitted vulnerabilities to be verified and fed into the graph database.

4 QUERY ANALYSIS AND OPTIMIZATION

4.1 Rewriting queries

Query equivalence in the general case is essential to applications such as query rewriting and optimization. In this section, we describe specific and commonly-occurring patterns of queries we are able to rewrite based on equivalence.

Query containment. We can formulate query equivalence in terms of query *containment*: when viewing queries A and B in terms of the result sets R_A and R_B they return when executed, we may say that $A \equiv B$ if $R_A \subseteq R_B$ and $R_B \subseteq R_A$.

Regular path queries are considered to be the basic querying mechanism for graph databases [22]. We take the result of a regular path query when executed to be the set of vertices traversed.

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 [29] but decidable. We describe how a subset of SGL is reduced to a subset of RQ to use its containment algorithm.

The SGL subset we consider contains only vertex steps, traversal steps, and the meta step `where`, with arbitrary 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` functions as a means of projection, allowing 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 (where a query's result is the ending bag of vertices) by selecting only the ending set of vertices and adding dedup.

Intuitively, consider a directed graph with vertices V, and two sets of vertices $v_s \subseteq V$ and $v_e \subseteq V$. 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 means that the result of a query is unchanged under reversal if the starting and ending points are known.

Since we only wish to 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. Thus, we can make use of the RPQ containment algorithm from Section 3.2 of [29].

The idea behind regular path queries is that we may view a query q as being 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 R_q.

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

Furthermore, the set of queries for which starting and ending points are known is closed under concatenation. 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 R_{q_1} and R_{q_2} which pass through q_{1e} or q_{2s}. We use this to rewrite queries for improved performance.

Definition 4.1. The *redundancy* of a query is the number of vertices reachable from some starting set v_s that do not eventually have a path to some ending set v_e without going back along an edge already traversed.

Intuitively, redundancy measures the number of dead-ended paths. Given some 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 stratify edges by cardinality:

- Edges which are one-to-one. There are no examples of this in our schema, as one-to-one relationships between vertices may be expressed by combining them into a single vertex.
- Edges which are many-to-one. An example is `has_method`, which connects `library` and `method` vertices.
- Edges which are many-to-many. An example is `depends_on`, which connects `library` vertices.

Many-to-one edges may be seen as parent-child relationships. Traversing from a child to its parent has a redundancy of 0 and is thus 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 dataset of 1.4M Java libraries.

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 1: Degree centrality

For many-to-many edges, we approximate the redundancy in both directions using degree centrality. The direction with the lower cardinality will lower redundancy, assuming that the same percentage of vertices traversed is redundant. In- and out-degree figures from our dataset are shown in Table 1.

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 would then be 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. Note that, this is a schema-specific optimization that is not possible in Gremlin. We evaluate the performance improvements in Section 6.1.

4.2 Automatic switching between OLAP and OLTP

Graph compute engines are adapted for different workloads. Apache TinkerPop, Gremlin's parent project, provides two main modes in which queries may be executed.

OLTP is characterized by transactional operations and/or high query volume. Queries run in this mode are typically simple, and written in a such a way as to be answerable in constant time, e.g. with a depth bound, or with guarantees that a constant number of nodes will be accessed. For example, "what are the libraries affected by some specific vulnerability?" In TinkerPop terms, OLTP queries must have a fixed, known starting point, which enables them to be answered without scanning a potentially large database cluster. Traversals are performed by keeping the entire frontier of the graph in memory. Thus, the full expressiveness of Gremlin is available for use. User-facing web applications are among the common uses of transactional queries.

OLAP is characterized by potentially long-running queries which operate over large amounts of data. In a distributed graph database, *scan* and *deep* queries are ideal candidates for this mode. Scan queries are those with starting points of variable size; as such, they may conceptually begin from all vertices in a database: for example, "what are the libraries affected by any vulnerability?" One might imagine starting from *all* vulnerabilities and traversing to the

affected libraries. Deep queries have a large depth bound, or none at all: for example, "find every library which calls a method in this library." One might imagine starting from the library, traversing to the methods within it, then taking the transitive closure of the `calls` edge.

OLAP queries are typically evaluated with a computation engine such as Apache Spark [33], as the traversal frontier may be too large to fit in memory on a single node. Spark would split the work up into tasks and schedule them across a cluster. Tradeoffs are made in expressiveness: because Gremlin's interpretation has to formulated in terms of Spark RDD operations, some complex Gremlin steps cannot be executed; an example is a meta-step like `where`, which is able to filter the result of a traversal by performing another traversal. Big data processing and analytics platforms are among the common uses.

One of the key design goals of SGL is abstraction. Users of SGL only need to care about the answers to their queries and not worry about the actual computation. As such, we include the following heuristics for switching between OLTP and OLAP:

- A deep query contains a transitive step (e.g. `calls*`), or a succession of steps with length above a certain threshold and/or an estimate of accumulated cardinality above a certain threshold.
- A scan query does not begin at a known point. An example is `library(_) count`, which returns the total number of libraries in the database; the wildcard indicates that it begins at *all* libraries, thus scanning the entire cluster.
- A query must be run in OLTP mode when the additional expressiveness of constructs like `where` is required. This overrides other heuristics.

We evaluate the effectiveness of these heuristics, and of switching to OLAP in general, in Section 6.2.

5 APPLICATIONS

5.1 Use cases

Finding similar vulnerabilities. once a pattern for a vulnerability is expressed in SGL, predicates can be varied slightly or removed for strictly more general queries. Relaxing version constraints allows a user to find similar vulnerabilities.

Consider the GlassFish query from earlier.

```
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)
```

We can generalize it by removing the `method_name` keyword argument to find all `readObject` methods which interact with something from `java/io/File`, or drop `get_path` to find all the methods it calls.

Finding inherited and embedded vulnerabilities. SGL's schema contains `embeds` edges, which express implicit dependencies between libraries: libraries which aren't explicitly declared in a manifest file such as `pom.xml`, but appear in another library due to JAR shading, copy-paste, or some other form of library vendoring.

```
let vulnerable_library = library(...) in
vulnerable_library union(
  dependent_on*, embedded_in*)
```

Given a `vulnerable_library`, taking the transitive closure of its implicit *and* explicit dependencies gives us the full picture of its impact on the ecosystem. We can use this information to estimate its severity.

Estimating impact. The transitive closure mentioned in the previous point also gives us a rough approximation of the impact of a vulnerability. We can use this information to write a single query that captures the impact of every known vulnerability in the ecosystem:

```
vulnerability(_) has_version_range has_library
  union(dependent_on*, embedded_in*)
```

Another impact analysis worth performing is to list the most vulnerable libraries:

```
let has_vulnerability =
  library_in_version_range
  version_range_in_vulnerability in
library(_) project(lib: identity,
  degree: has_vulnerability count)
  order_by(degree desc)
```

Suggesting safe dependency upgrades. For remediating known vulnerabilities often the suggest fix is to upgrade to a fix or safe version. However, if the vulnerability is in a transitive dependency, constraining it may break the invariants of downstream dependencies which rely on the vulnerable version. In Java, this may cause anything from a compilation error in the best case to an obscure runtime error in the worst.

Using SGL we can ask for a version of a *direct* dependency which has the fixed version of the transitive dependency, allowing us to suggest a safer fix:

```
let desired_transitive = library(...) in
let direct_depedency1 = library(...) in
let direct_depedency2 = library(...) in
desired_transitive dependent_on*
  union(
    direct_dependency1,
    direct_dependency2)
```

API analysis. SGL enables users to find uses of unsafe (`sun.misc.Unsafe`) or interesting APIs (cryptography, network I/O), similar to rule-based static analysis tools such as FindBugs [6].

```
let iv_parameter_spec = method(class_name:
  'javax/crypto/spec/IvParameterSpec') in
let random = method(class_name:
  'java/util/Random') in
iv_parameter_spec called_by where(calls random)
```

For example, the above query returns potentially insecure use of `java/util/Random` by finding callers of the method that also call `IvParameterSpec`.

5.2 Case study: GlassFish zero-days

The real power of SGL comes from the ability to analyze and find new, *zero-day* vulnerabilities. In this section, we present 2 specific findings from the analysis we did on the GlassFish Open Source Edition. We verified and disclosed to Oracle a total of 23 issues [16] that were fixed in the latest version of the GlassFish server.

Security Manager Bypass. CVE-2016-0763 is a vulnerability reported on the Apache Tomcat Application Server, we formed an SGL query that looks for similar issues in other Application Servers such as GlassFish.

```
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 [19]. 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 then found and verified that GlassFish Open Source Edition 3.0-4.1.2 was affected by this issue.

6 EVALUATION

6.1 Query-rewriting benchmarks

Consider a query from the GlassFish case study. Intuitively, its intent is to check if there is a path from a GlassFish class to the `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 a key property which makes it amenable to optimization: it is the conjunction of two reversible queries, which mean it is also reversible. The reverse of the query is as follows.

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

Query	$estimate_o$	$estimate_r$	$runtime_o$ (s)	$runtime_r$ (s)
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 2: Rewriting benchmarks

Query	OLAP	OLTP
Deep queries	✓	✗
Scans, e.g. `library(_)`	✓	✗
`library(...) where(...)`	✗	✓
`library(...) union(...)`	✗	✓

Table 3: OLAP/OLTP differences

From Table 1, the estimated redundancies of the queries are 55.7 and 391.2, respectively. This indicates that the query's 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 2. 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 and its reversal contains exactly the same edges. The HTTP Redirect example illustrates this, its edges being `calls called_by`.

6.2 OLTP/OLAP switching

By default, TinkerPop executes queries in OLTP mode. OLTP is not suited to all kinds of workloads. Queries which traverse deeply and build up large graph frontiers or scan large portions of a database cluster are more effectively executed in OLAP mode. On the other hand, queries which use language features not supported in OLAP must execute in OLTP by necessity.

Table 3 summarizes the differences. Deep queries perform significantly better when executed in OLAP (not being subject to timeouts from the underlying database or causing out-of-memory errors), but the rest *require* a respective mode. Switching automatically allows users to work at a higher level of abstraction and write queries without having to reason about how exactly they execute.

6.3 Automatic Vulnerability Deduplication

SGL's automated vulnerability deduplication improves vulnerability datasets by preventing duplicate issues from being added.

Deduplication of CVE-2012-2098. As an example, consider CVE-2012-2098. It describes an issue of Denial of Service (DoS) through CPU consumption. Since the issue has been found and reported in multiple components, security researchers at SourceClear reviewed it multiple times independently, and there were at least 3 attempts to add it to our dataset as a vulnerability.

The vulnerability is expressed in SGL as follows:

```
vulnerability(cwe: 310)
  has_version_range union(
    version_range(coord1: 'org.apache.commons',
      coord2: 'commons-compress',
      from: '1.0', to: '1.4')
  union(
    has_library union(
      library('java', 'org.apache.commons',
      'commons-compress', '1.0'),
      ...,
      library('java', 'org.apache.commons',
      'commons-compress', '1.4')),
    has_vulnerable_method union(
      method('org/apache/commons/compress/
        compressors/bzip2/BZip2CompressorOutputStream',
        'blockSort', '()')))
```

When expressed in SGL this vulnerability is uniquely identified and any attempt to add it again would lead to a error based on structural typing of vulnerability vertex. In total, there were at least 84 additional preventable duplicate vulnerabilities that were rejected. This represents almost 10% of the total number of vulnerabilities we have in our dataset for Java.

6.4 Survey

In order to evaluate SGL's suitability for vulnerablity description, we performed a survey of security practitioners on several security groups (sourced from LinkedIn and security communities like OWASP). We received 36 responses that contained a good mix of security researchers (60%) and software engineers (40%).

Questions. For each question, survey participants were presented with illustrations of both the Open Vulnerability and Assessment Language (OVAL) and the SGL formats for comparison, and were asked to compare between them for comprehensibility, precision, and conciseness.

- Which is better for specifying the affected application(s)
- Which is better for specifying the affected version(s)
- Which is better for specifying other information or metadata
- Which would be the preferred choice to create and report a vulnerability?
- Which would be the preferred choice to integrate into an existing vulnerability scanner product?
- Which language would allow better discovery of vulnerabilities?
- Which language would be able to scale better in terms of the amount of vulnerabilities?

Results. Overall, around 58.2% of the `Software Engineers` preferred OVAL and around 70.2% of the `Security Researchers` preferred SGL. In terms of conciseness of specifying the affected version ranges, 66.6% of respondents prefer SGL over OVAL, and about 61.1% of them think that SGL would be able to scale better in terms of amount of vulnerabilities. Also, about 55.5% of the participants would prefer to report a vulnerability with SGL.

Edge	Data source
depends_on	`pom.xml` files using Aether [11]
calls has_vulnerable_method	Call graphs from bytecode
has_method defines has_library has_class extends	Bytecode
embeds has_library_hash	Bytecode hashing
has_file	Source code
has_method_hash	Source code hashing [26]
has_version_range	Curated

Table 4: Implementation methods

7 DISCUSSION

7.1 Implementation

The language infrastructure for SGL comprises a parser, compiler, and other supporting modules for analysis and optimization. The database infrastructure consists of DataStax Enterprise Graph [13] as the storage back-end, and an ETL pipeline which processes source code and bytecode. All of this runs on AWS infrastructure (EC2, S3, SNS) but is not tied to it. We currently store data for 79M vertices and 582M edges. Vertices represent libraries, methods, classes, files, source code hashes, and bytecode hashes, and are computed from open-source data as shown in Table 4 (details are beyond the scope of this paper). This was computed from 1.4M Java libraries and totals about 76GB of data.

7.2 Design choices

SGL was designed to precisely capture the domain of open-source security, and allow users to draw insights from data with only the property graph model and schema in mind. This entails abstracting over implementation details and inconsistencies. TinkerPop 3 provides ways to create internal DSLs [15], but we chose to go with an external DSL primarily so we could remove things from Gremlin (e.g. Turing-completeness), giving us a simpler language grounded in existing theory (e.g. decideable containment) that is more amenable to static analysis.

7.3 Threats to validity

For some of the features in SGL, such as impact analysis, and threat analysis to be highly effective, SGL depends on the amount of vulnerabilities to be discovered and expressed in SGL. If SGL is not seen to be easy to implement, or easy to use, language, it would not be as effective as it was designed to be.

The results from the survey, has validated a potential issue in the state of reporting vulnerabilities and the way vulnerability data is stored. In the survey, we have categorized the responses between `Security Researchers` and `Software Engineers`. More `Security Researchers` expressed interest in SGL rather than the OVAL format, and more `Software Engineers` have, expressed their interest in favor of the OVAL format.

We speculate that `Software Engineers` would prefer the OVAL format due to the familiarity of the XML format, used for expressing information. When viewed as a comparison between XML, and a new DSL, it is completely valid to stick to a known format such as XML as compared to a new DSL, such as SGL, which requires an additional level of implementation and understanding in order to integrate into existing software and pipelines.

8 RELATED WORK

As a DSL, SGL exists at the intersection of vulnerability description languages, graph query languages, and program analysis languages. We briefly compare the features of these related languages in this section.

Perhaps the most widely-adopted standard for vulnerability description is OVAL [1]. It is an XML-based language for describing vulnerabilities in terms of information derived from CVEs and CPEs. It is also executable, meant to be executed against system configurations to determine if they are vulnerable. A reference implementation is provided in the form of an interpreter; commercial offerings such as Joval [17] also exist. Other efforts in the space are OASIS AVDL [2], another XML-based language for describing exploits for web applications, and VuXML [3], which captures vulnerabilities in software packages for the FreeBSD operating system. Compared to OVAL, SGL is more focused on the domain of open-source libraries and vulnerabilities, and on drawing insights from large external datasets, compared to local scans to check for vulnerable configurations.

SGL may also be contrasted with general-purpose graph query languages as it can also express arbitrary graph traversals. Its closest relative is Gremlin [28], from which it inherits its imperative semantics. Other languages in the space are declarative: Cypher [18] is the query language for the Neo4j graph database, and there has been a proliferation of Datalog-based graph query languages recently in Datomic Datalog [9] and GraQL [24].

Other DSLs and platforms for program analysis are LogicBlox LogicQL [25], Semmle QL [23], PQL [27]. Joern [31] is a particularly similar effort which abstracts over source code with a code property graph [32] model and utilizes a graph database for queries. Unlike many of the languages in this category, SGL does not capture data flow and is not purely declarative. We hope to adopt some form of these qualities in future.

9 CONCLUSION

We presented SGL, a DSL designed for the analysis of large graph-structured datasets in the domain of open-source security. We gave a thorough overview of its syntax, semantics, type system, and the analysis techniques we use to optimize queries. SGL can be used as a machine- readable means of describing vulnerabilities, supporting automated checking and deduplication. We described the zero-day vulnerabilities we found in GlassFish with it. Finally, we discussed promising use cases and the benefits it affords security researchers.

In future, we intend to work on identifying more patterns of SGL queries for optimization and rewriting. We are also working on applying machine learning techniques to automatically infer SGL queries that characterize API usage in libraries.

REFERENCES

[1] 2002. OVAL - Open Vulnerability and Assessment Language. (2002). https://oval.mitre.org

[2] 2003. OASIS Application Vulnerability Description Language (AVDL) TC | OASIS. (2003). https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=avdl

[3] 2004. VuXML. (2004). https://www.vuxml.org/

[4] 2014. CPE - Common Platform Enumeration. (2014). https://cpe.mitre.org/

[5] 2014. Heartbleed. (2014). http://heartbleed.com/

[6] 2015. FindBugs. (2015). http://findbugs.sourceforge.net/

[7] 2015. FREAK SSL/TLS Vulnerability - US-CERT. (2015). https://www.us-cert.gov/ncas/current-activity/2015/03/06/FREAK-SSLTLS-Vulnerability

[8] 2015. GHOST: glibc vulnerability (CVE-2015-0235). (2015). https://access.redhat.com/articles/1332213

[9] 2016. Datomic - Home. (2016). http://www.datomic.com/

[10] 2016. Over 6,000 vulnerabilities went unassigned by MITRE's CVE project in 2015. (2016). https://www.csoonline.com/article/3122460/techology-business/over-6000-vulnerabilities-went-unassigned-by-mitres-cve-project-in-2015.html

[11] 2017. Aether | projects.eclipse.org. (2017). https://projects.eclipse.org/projects/technology.aether

[12] 2017. CWE - Common Weakness Enumeration. (2017). https://cwe.mitre.org/

[13] 2017. DataStax Enterprise Graph. (2017). https://www.datastax.com/products/datastax-enterprise-graph

[14] 2017. Equifax: Cybersecurity Incident & Important Consumer Information. (2017). https://www.equifaxsecurity2017.com/

[15] 2017. Gremlin DSLs in Java with DSE Graph. (2017). https://www.datastax.com/dev/blog/gremlin-dsls-in-java-with-dse-graph

[16] 2017. How we found exploitable zero-days in the open-source GlassFish server with the Security Graph Language. (2017). https://www.sourceclear.com/blog/How-we-found-exploitable-zero-days-in-the-open-source-GlassFish-server-with-the-Security-Graph-Language/

[17] 2017. JovalCM. (2017). http://jovalcm.com/

[18] 2017. openCypher. (2017). http://www.opencypher.org/

[19] 2017. Oracle Critical Patch Update Advisory - October 2017. (2017). http://www.oracle.com/technetwork/security-advisory/cpuoct2017-3236626.html

[20] 2017. SourceClear. (2017). https://www.sourceclear.com/

[21] 2017. TinkerPop 3 Documentation. (2017). http://tinkerpop.apache.org/docs/current/reference

[22] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2016. Foundations of Modern Graph Query Languages. *CoRR* abs/1610.06264 (2016). http://arxiv.org/abs/1610.06264

[23] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. 2016. QL: Object-oriented Queries on Relational Data. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[24] D. ChavarrÃŋa-Miranda, V. G. Castellana, A. Morari, D. Haglin, and J. Feo. 2016. GraQL: A Query Language for High-Performance Attributed Graph Databases. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1453–1462. https://doi.org/10.1109/IPDPSW.2016.216

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

[26] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *Security and Privacy (SP), 2017 IEEE Symposium on. IEEE*.

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

[28] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language. *CoRR* abs/1508.03843 (2015). http://arxiv.org/abs/1508.03843

[29] Moshe Y. Vardi. 2016. A Theory of Regular Queries. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '16)*. ACM, New York, NY, USA, 1–9. https://doi.org/10.1145/2902251.2902305

[30] Mitchell Wand. 1987. Complete Type Inference for Simple Objects. (01 1987), 37-44 pages.

[31] Fabian Yamaguchi. 2015. Pattern-Based Vulnerability Discovery. (2015).

[32] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 590–604. https://doi.org/10.1109/SP.2014.44

[33] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, Berkeley, CA, USA, 10–10. http://dl.acm.org/citation.cfm?id=1863103.1863113