Static Detection of User-specified Security Vulnerabilities in Client-side JavaScript

Jens Nicolay
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
jnicolay@vub.ac.be

Valentijn Spruyt
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
vspruyt@vub.ac.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
cderoove@vub.ac.be

ABSTRACT

Program defects tend to surface late in the development of programs, and they are hard to detect. Security vulnerabilities are particularly important defects to detect. They may cause sensitive information to be leaked or the system on which the program is executed to be compromised.

Existing approaches that use static analysis to detect security vulnerabilities in source code are often limited to a predetermined set of encoded security vulnerabilities. Although these approaches support a decent number of vulnerabilities by default, they cannot be configured for detecting vulnerabilities that are specific to the application domain of the analyzed program.

In this paper we present JS-QL, a framework for detecting user-specified security vulnerabilities in JavaScript applications statically. The framework makes use of an internal domain-specific query language hosted by JavaScript. JS-QL queries are based on regular path expressions, enabling users to express queries over a flow graph in a declarative way. The flow graph represents the run-time behavior of a program and is computed by a static analysis.

We evaluate JS-QL by expressing 9 security vulnerabilities supported by existing work and comparing the resulting specifications. We conclude that the combination of static analysis and regular path expressions lends itself well to the detection of user-specified security vulnerabilities.

1. INTRODUCTION

Security vulnerabilities are important program defects to detect. They might lead to leaks of sensitive information, or even to the system on which the program is executed being compromised. Existing approaches that use static analysis to detect generic defects and security vulnerabilities in source code often support but a fixed set that is hard-coded. In case support for user-defined checks is available, these often need to be expressed in a general-purpose language which renders doing so rather cumbersome.

These problems can be overcome by enabling users to define the characteristics of a sought-after defect in a domain-specific language. This renders the static analysis that identifies source code exhibiting these properties *application-specific* and the defect itself *user-defined*.

A diverse set of specification languages for source code characteristics have been proposed. Specification languages for structural characteristics include graph rewrite rules [31] or logic formulas [7, 10, 32, 22, 5, 2, 12, 9] that quantify or range over a program's AST nodes. Specification languages for behavioral characteristics include reachability queries [11, 8, 37, 36, 24, 26], temporal logic formulas [25], state machines [13, 3] and logic formulas [28, 14] that quantify over control flow and data flow analysis results.

1.1 JS-QL

In this paper we present JS-QL, an internal domain-specific specification language supporting *regular path expressions* [8, 26], hosted by JavaScript. JS-QL enables the formulation of pre-encoded and user-defined queries for detecting security vulnerabilities in a readable and expressive manner. Queries are executed over a flow graph computed by a static analysis. The flow graph represents the required program information in terms of control flow and value flow.

The application-specific nature of the queries written in JS-QL, together with the possibility for users to define these queries by themselves, makes it a powerful aid in checking program characteristics, and security vulnerabilities in particular. In contrast to general program checkers, a much wider range of program characteristics can be detected.

JS-QL supports multiple types of queries, such as forward and backward queries, and universal and existential queries, allowing developers to explore their programs in several ways.

JS-QL is inspired by the strengths and shortcomings of existing program analysis tools which are extensible with user-defined rules and queries. JS-QL differs from existing solutions in its target language, the way a program is represented, and how the user is given access to this representation:

- JS-QL targets JavaScript programs.
- JS-QL works on a flow graph, which is a rich program representation—richer than a program dependence graph for example.
- JS-QL is an internal JavaScript DSL for specifying regular path expressions, and aims to be a readable and expressive query language.

PLAS'16, October 24 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4574-3/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2993600.2993612

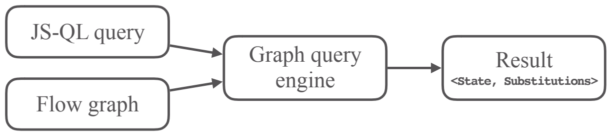

Figure 1: JS-QL architecture

1.2 Contributions

Our contributions are the following:

1. We present the JS-QL domain-specific language for expressing succinct and highly readable program queries.

2. We present the JS-QL framework to match JS-QL queries against JavaScript programs.

3. We evaluate JS-QL by expressing multiple security vulnerabilities and comparing the resulting specifications to corresponding ones of existing work (Section 5).

1.3 Overview of the approach

The architecture of the JS-QL framework is depicted in Figure 1. The query engine takes as input a flow graph of a program (Section 2) and a JS-QL query (Section 3). The output computed by the engine consists of `<State, Substitutions>` for all paths in the graph on which a match for the query was found (Section 4).

2. REPRESENTING PROGRAM BEHAVIOR AS FLOW GRAPHS

Any approach for detecting security vulnerabilities in a program must rely on information about the runtime behavior of that program. More specifically, it is necessary to determine the value flow (what are the possible values that expressions in a program may attain?) and control flow (what are the possible paths through a program?) of programs. In the JS-QL framework we rely on JIPDA [30] to provide this information. Given an JavaScript program, JIPDA computes a flow graph for this program. The flow graph is a finite representation of the runtime behavior of the input program. Nodes in a flow graph represent program states, and edges denote transitions between states.

Other types of graphs, such as a program dependence graph [17], can also be used to track information flow, but these graphs often lack more general information about program states, making them less qualified for use as the (main) program representation. On the other hand, a larger and more precise representation of program information generally is more costly to compute and query than a simpler one.

2.1 Computing the flow graph

JIPDA defines the small-step semantics of a subset of JavaScript as an abstract machine that transitions between states [15]. As the abstract machine describes the individual steps required to evaluate programs expressed in JavaScript, this type of semantics is referred to as *small-step operational semantics*. Starting from an initial state, which represents the input program to be evaluated, the machine continually transforms machine states into possible successor states, generating a graph of all potentially reachable states. Every state corresponds to a snapshot of the program state (expressions, values, the heap, and the stack) while the machine is evaluating the program. Evaluation states represent the evaluation of an expression of the program in a particular binding environment. Continuation states indicate that the machine is ready to continue evaluation with a value it has just calculated. Other types of states, such as return states and throw states, are special kinds of continuation states and indicate a special type of control flow.

Because JIPDA is flow-sensitive, it is guaranteed that a state `A` on some path in the graph occurs before a state `B` on the same path if `A` occurs before state `B` in the program. This simplifies reasoning about patterns in a program, because no false positives will occur with regards to the order of execution of states.

2.2 Finitizing the semantics

JS-QL queries (Section 3) examine the resulting flow graph to obtain the necessary control and value flow information required for detecting non-trivial security vulnerabilities. However, the problem with computing a flow graph using an abstract machine as described above, is that the graph can become unpractically large, or even infinite. To ensure that resulting flow graphs are finite and reasonable in size, while still containing useful and consistent results about the behavior of programs, JIPDA applies the technique of *abstract interpretation*. Abstract interpretation is a theory that offers a solution to the problem of costly or uncomputable runtime properties [6]. Although the concrete semantics of a non-trivial program are not computable with finite resources, abstract interpretation nevertheless offers a framework for getting useful answers to non-trivial questions about programs. The answers we obtain from abstract interpretation will necessarily be approximations due to the undecidability of the questions asked.

In the context of this paper the results of flow analysis must conservatively approximate the control flow and value flow of programs, erring on the safe side in order to remain faithful to the original semantics. For example, if during concrete interpretation a call site invokes a particular function, the analysis must reflect this possibility, but the analysis may be imprecise in the sense that it overestimates the set of called functions at call sites.

The default abstract semantics for JS-QL equips JIPDA's abstract machine with a set lattice containing primitive types and pointers, and with monovariant object allocation (i.e., based on the syntactic allocation site).

3. THE JS-QL QUERY LANGUAGE

JS-QL, short for JavaScript Query Language, is a language for expressing different kinds of application-specific and user-defined security vulnerabilities. To this end, the language specializes in expressing queries matching sequences of program states in the flow graph by making using of *regular path expressions* [8, 26]. Regular path expressions describe paths in a graph in a syntax similar to regular expressions. It is important that exploring and accessing information in a flow graph happens in an expressive yet readable way, as such graphs might be complex to understand. We are convinced that regular path expressions enable users to write clean and understandable queries.

We first discuss the design of JS-QL (Section 3.1), and then present an overview of the language (Section 3.2).

3.1 Design

We have designed JS-QL as an internal *domain-specific language* (DSL) built on top of JavaScript. Domain-specific languages are well suited for querying graphs, because they allow the expression of ideas at the level of abstraction of the problem domain—in our case queries over program information contained in a flow graph. DSLs avoid much of the boilerplate code that would be required when writing the same queries in a general-purpose language. However, the cost of designing and implementing a DSL is non-negligible, and users have to be educated in order to use the language.

3.1.1 External DSLs

DSLs that have a compiler to translate DSL programs into applications are called *external* DSLs. The main advantage of an external DSL is that the implementation of the compiler can be completely tailored to the DSL. The external DSL in turn is not restricted in any way regarding notation and primitives because its syntax is independent of any underlying host language. There is ample related work on external DSLs for graph traversal and querying [16, 23, 1].

3.1.2 Internal DSLs

An *internal* (or *embedded*) DSL is built on top of a host language—JavaScript in case of JS-QL. This type of DSL inherits the infrastructure of its host language, and tailors it toward the domain of interest. Although internal DSLs are restricted by the syntax of their host language, they can make full use of the host language as a sublanguage, thus offering the expressive power of the host language as well as its domain-specific expressive power. This expressive power, along with not having to build a full fledged compiler for our DSL, are the main reasons we preferred an internal DSL over an external one for JS-QL.

The Gremlin language [33] and Dagoba [35] are examples of existing internal DSLs for graph traversal. Günther and Cleenewerck [21] illustrates how well the flexibility and expressiveness of Ruby accommodates the implementation of an internal DSL for surveys. These languages served as inspiration for the design of JS-QL as an internal domain-specific query language.

3.1.3 DSL design patterns

The principal design pattern used in JS-QL is *method chaining*. Method chaining is a pattern in which the output of one method flows naturally as input to another method. This approach offers a *fluent interface* [18] to the user with the aim to write readable queries while avoiding boilerplate code and coding mistakes. A fluent API is ideal for querying states in flow graphs, as it is natural to describe states we want to encounter on a path through the graph in a clearly specified order. Consider for example the query `G.skipZeroOrMore().functionCall()`. `G` indicates the start of a query, and the query then `skip`s zero or more states until a function call state is encountered.

3.2 Overview of JS-QL

Entry point.

JS-QL is a internal DSL, and therefore has to provide an entry point that can be used to start querying the flow graph. A single entry point `G` is created for this purpose, and represents the flow graph without any states or paths matched. The query `G.state()` for example matches the first state in the graph.

States and state properties.

The `state` construct is the single most basic element of the language. It matches any state in the flow graph, but does not provide other information. Nevertheless is it the most important building block of the language because it is used to construct higher-level queries and predicates, as we discuss later in this section.

Constructs `evalState`, `kontState`, `returnState` and `resultState` are equivalent to `state` but match only those specific kinds of states.

JS-QL deconstructs program states as nested key-value pairs. Keywords `node`, `value`, `benv`, `store`, etc. are *state keywords* that correspond to program state properties. The key indicates the property to be matched, whereas a value can be one of three things: a *variable* that gets bound to the key's corresponding value in the JIPDA state, a *nested map* that further deconstructs the current property, or a *literal* that only matches with its specific value.

The `this` JS-QL keyword is an implicit property that is made available for each deconstructable map in the language. Its value represents the immediately enclosing object.

JS-QL enables users to specify sequences of states through method chaining. When checking the flow graph against this type of query pattern, all states in the pattern must be matched in exactly the order specified. When a state in a query pattern is encountered that does not match with the current state in the flow graph, the matching process is aborted for the current path that is investigated in the flow graph.

Variables.

Variables are strings that start with a `?`. JS-QL has to use strings for variables because of the embedded nature of the language: JavaScript would not recognize variables specified as names (i.e., as regular JavaScript variables).

Temporary variables are variables that are not contained in the resulting substitutions. By introducing these variables in the code, no conditionals have to be written to check whether a certain property is present in a map, as we can just use a temporary variable as a replacement. Using temporary variables, a predicate (see below) can also provide information that can be queried optionally.

Regular path expressions.

JS-QL supports the use of the Kleene `star` operator to match any number of states, and the Kleene `plus` operator to match at least one state. These constructs must be placed *after* the state(s) they are applied to.

Query patterns can be surrounded by braces. Left and right braces in JS-QL are denoted by `lBrace` and `rBrace`, respectively.

JS-QL provides the `or` language construct to specify disjunction. The following example detects all uses of a variable `v` in binary arithmetic expressions. The variable can be on either side of the binary operator.

```
G.skipZeroOrMore()
.lBrace()
  //Left-hand side with name 'v'
  .state({node: { type: 'BinaryExpression',
                left: {name:'v'}}})
  .or()
  //Right-hand side with name 'v'
  .state({node: { type:  'BinaryExpression',
                right: {name:'v'}}})
.rBrace()
```

JS-QL combines techniques from regular expressions with a special built-in construct, the **wildcard**. In JS-QL, a wildcard serves the sole purpose of matching any state it gets compared with. Because wildcards in JS-QL enable users to specify that some states in the flow graph can be skipped, another name for this construct is **skip**. A **wildcard** construct acts just like regular **states** in a query, meaning that they only match a single state in the flow graph.

Negation.

When placed before a state, the **not** construct will only be matched if the negated state can not be matched with the current state in the flow graph. This can be used to ensure that a state does not occur on the matched path. A negated state immediately followed by **star** (**plus**) can be read as: "Match zero (one) or more states that are *not* the negated state".

Properties.

JS-QL has a built-in keyword **properties** that can be used to introduce additional properties. Introducing properties increases the expressivity of the language. When the construct is used to obtain more information from already bound variables, it avoids queries with deeply nested maps which quickly become confusing to read and bothersome to modify. Function **prop** can be used to query for a specific property. The first argument of **prop** is the function that is applied when the matching engine processes the query. The remaining arguments of **prop** are passed as arguments to the applied function. Another way to define properties is by simply specifying which attribute of a variable one wishes to capture, as in line 6 in the following example.

```
1   G.skipZeroOrMore()
2   .state({
3         node:{ declarations: '?decls' },
4         properties:{
5           '?dec'      : prop('memberOf', '?decls'),
6           '?decName'  : '?dec.left.name'
7           '?decNameU': prop(function(a){
8                             return a.toUpperCase();
9                           }, '?decName')
10
11        }})
```

This example matches all states in a flow graph that declare variables. For each declaration **?dec** in a list of declarations **?decls**, we introduce the declaration itself, the name of the declared variable **?decName** and its uppercase conversion **?decNameU** as properties.

Filters.

JS-QL provides the **filters** keyword for specifying filters in queries. A filter can be any function, predefined or specified by the user, that returns a boolean value. When returning a **true** value, pattern matching continues. Otherwise the matching process aborts and no match for the current path in the flow graph is found. Filters in JS-QL work very similar to properties, except that they act as guards who filter out states that do not satisfy certain conditions. As no variables have to be stored for filters, the value of the **filters** keyword is a JavaScript array, instead of a (nested) map. A filter is declared through the **cond** JavaScript function, which is similar to **prop** for properties. The following example query detects multiple variable declarations in one declaration statement.

```
G.skipZeroOrMore()
.state({
      node:{declarations: '?decls'},
      properties:{
        '?length' : prop('length', '?decls')
      },
      filters:[
          cond('>', '?length', 1)
      ]
    })
```

Dataflow.

Variables and functions from the input program can be looked up in JS-QL using the **lookup** keyword. The value-part of this keyword is a map with the names of the variables to look up as keys and the variable names that need to be bound to the addresses as values.

Predicates and policies.

JS-QL provides a number of built-in predicates to abstract over matching single states. For example, built-in predicate **functionCall** matches states that evaluate a function call. In addition, users can specify their own predicates and policies. Policies are sequences of **state**s and predicates forming a query pattern.

Recursion.

Recursive queries are queries that invoke themselves. JS-QL supports recursive queries by providing the **rec** function. This function takes two arguments: the mapping for the recursive step, and the predicate that is called recursively. A recursive query can for example be used to detect by which variables a variable is tainted. Our implementation contains a naive **taintedBy** policy that considers variables. This policy takes three optional arguments: **orig** denotes the original value which is aliased, **alias** represents the alias of the original value, and **rec** keeps track of all variables that have been used as aliases in between **orig** and **alias**.

Type of queries.

JS-QL is a flexible query tool that supports different types of queries.

Existential queries match a pattern if at least one path is found. *Universal queries* require that the query matches for the same substitutions along all possible paths in the flow graph between two states. The following example is a universal query that checks where a variable has a constant value. It assumes the presence of a predicate **def(name:'?x')** that matches all definitions and redefinitions of a JavaScript variable bound to **?x**.

```
 G.skipZeroOrMore()
 .def({name: '?x'})              // Define the
     variable
 .not().def({name: '?x'}).star() // As long as it is
     not redefined
```

Queries can explore the flow graph in two directions. *Forward queries* match states in the flow graph starting at the initial state and in the direction of the final states of the graph. *Backward queries* match states in the flow graph starting at the final states and in the direction of the initial state of the graph. Although backward queries are less common, they are needed for performing certain types of program analyses such as live variable analysis. The example below depicts a backward query for performing a live variable analysis. The entry point for backward queries is `F`, and it assumes the presence of a predicate `use(name:'?x')` that matches all uses of a JavaScript variable bound to `?x`.

```
F.skipZeroOrMore()
.use({name: '?x'})              // Read the variable
.not().def({name: '?x'}).star() // As long as it isn'
    t written
```

4. GRAPH QUERY ENGINE

The graph query engine is the core of our tool. It matches user-defined queries against states in a JIPDA flow graph, capturing and unifying relevant program properties. The query engine takes a flow graph and a JS-QL query as input (Figure 1).

We based our algorithm for solving queries on the work presented in Liu et al. [26]. A JS-QL query is parsed like a regular expression. Each state in the pattern represents one character in the regular expression, defined by objects of type `RegexPart`. These parts of the pattern have five fields to ease the translation from regular expression to automaton:

1. *Name*: The name of a regular expression part. In the current implementation, the name denotes the type of the state/predicate that the `RegexPart` represents (e.g. `state`, `wildcard`, `not`, `lbrace`).

2. *Symbol*: The actual symbol that will be parsed by the parser to set up the automaton corresponding to the query.

3. *Object*: The argument of the state/predicate in which all variables are bound and properties, filters, and lookups are specified.

4. *ExpandFunction*: A higher-order function representing a recursive predicate or policy that is called for recursive queries. This argument does not need to be specified when a query is not recursive.

5. *ExpandContext*: A unique identifier to avoid overlapping recursive variable names, only used for recursive queries.

States and predicates are function calls returning `this` to enable method chaining. Each function call represents one state in the pattern, and thus for each of these calls the corresponding `RegexPart` gets pushed into a map containing the pattern information.

Recursive query patterns can contain an arbitrary number of states, so they cannot be modeled directly as a sequence of `RegexParts` as the length of the actually matched pattern is not known. We therefore store a recursive query in a single `RegexPart` object, and mark it with the name "subgraph". Additionally, we specify the *ExpandFunction* and *ExpandContext* to be able to process the subgraph in the matching algorithm. We adopted the idea to treat recursive queries in this manner from the PQL language [28].

The entire query pattern (i.e. the map) is processed by applying *Thompson's Construction Algorithm* and the *Subset Construction Algorithm* consecutively to obtain an NFA and DFA respectively [34].

The algorithms for existential and universal queries that are used as the basis of the tool were first presented in Liu et al. [26]. In our work we require algorithms for matching regular path expressions with graphs containing simple information. These algorithms match the edge labels of the graph with the edge labels of the regular path expression (in the remainder of this paper, we will call these expressions *patterns*).

The algorithms presented by Liu et al. do not support recursion, and therefore do not provide constructs for recursive queries. We augmented these algorithms to consider subgraphs as a regular data structure and implemented a way to process them. This required transforming the JIPDA flow graph such that all state information is available in the edges instead of the nodes. This presented no difficulties as no explicit edge information is present in regular JIPDA flow graphs.

The information on the edge labels in the approach described by Liu et al. consists of simple information. In their work the arguments of a pattern can only be symbols, such as a string or a literal, limiting the type of graphs that can be analyzed. We extended these algorithms to support nested maps as arguments, as these are the main data structure in JS-QL for representing a state.

A JS-QL query output consists of tuples `<State, Substitutions>` for all paths on which a match for the query was found. JS-QL supports three types of variables that each play a particular role.

1. *Regular variables* contain the information that the user wants to match in a query. When a match succeeds, they are contained in the resulting substitutions.

2. *Recursive variables* are used as intermediary variables. They function as variables that were bound in the previous step of a recursive query, enabling a recursive step to work with the value of one or more variables of the previous step.

3. *Temporary variables* are state-local variables used when not specifying a certain argument for a predicate or policy.

By allowing the user to work with these three types of variables, writing queries becomes flexible because the bindings in a substitution set can be limited to only the information needed by the user. When imagining JS-QL without the support of temporary variables for example, the size of the substitution set for more complex queries would quickly grow large. This decreases the readability of the results and makes interpretation of these results more difficult.

5. EVALUATION

We evaluated the expressiveness of JS-QL by expressing 9 security vulnerabilities distilled from 3 papers that present alternative approaches for expressing security-related program queries: GateKeeper [20], PidginQL [24], and ConScript [29]. Table 1 gives an overview of the 9 different vulnerabilities and their origin. For each vulnerability, we evaluate how well its formulation in JS-QL matches the original formulation. Because we do not have space to discuss

Policy	Origin	Name
V1	GateKeeper	Prototype poisoning
V2	GateKeeper	Global namespace pollution
V3	GateKeeper	Script inclusions
V4	PidginQL	CMS non-administrator sends message to all CMS users
V5	PidginQL	Public output depends on user's non-cryptographically hashed password
V6	PidginQL	Existing database is opened before master password is checked
V7	ConScript	String arguments to `setInterval`, `setTimeout`
V8	ConScript	Non-HTTP-cookies
V9	ConScript	Resource abuse

Table 1: Evaluated policies and their origin.

all 9 vulnerabilities, we choose to report on one vulnerability per approach we compare against: V1, V4, and V7. We conclude this section by evaluating our framework based on the results from the experiments, and specify advantages and limitations of our approach (Section 5.4).

5.1 GateKeeper

GateKeeper is a mostly static approach for soundly enforcing security and reliability policies for JavaScript programs [20]. Programs are represented as a database of Datalog rules against which GateKeeper policies, also written in Datalog, are checked.

Vulnerability 1: Prototype poisoning

Prototype poisoning compromises trusted code by manipulating global prototypes from which that code inherits [27]. An example of this attack is changing the `toString` function on the global object to spoof a URL.

```
String.prototype.toString = function() {
    return "www.trustedsite.com";
}

var login = function() {
  if (document.location.toString() === "www.
      trustedsite.com") {
    // proceed
  }
}
```

GateKeeper.

Guarnieri and Livshits [20] defines the **FrozenViolation(v)** predicate to detect writes to prototypes of built-in objects (Listing 1). This predicate first looks for all stores of field **v**. This field points to location **h2**, which represents the points-to address for variables. Only writes to built-in objects are infringements of the policy, which implies that **h2** has to point to a field of of one of these objects. This is expressed as follows: in **BuiltInObjects(h)**, **h** points to the heap location of a built-in object. The **Reaches(h1,h2)** predicate makes sure that the field that was stored reaches the built-in object directly or indirectly, by recursively checking if one of the properties of the built-in object has a field pointing to the stored field.

JS-QL.

Listing 2 depicts the JS-QL query to express the same vulnerability. The filter on lines 10-12 indicates that we only want to detect writes to the **prototype** property of the **String** object. When omitted, the query detects all writes to this object.

```
Reaches(h1,h2) :- HeapPtsTo(h1,_,h2).
Reaches(h1,h2) :- HeapPtsTo(h1,_,h3),
                  Reaches(h3,h2).

FrozenViolation(v) :- Store(v,_,_),
                      PtsTo(v,h2),
                      BuiltInObject(h1),
                      Reaches(h1,h2).

% Specify all built-in objects
BuiltInObject(h) :- GlobalSym("String", h).
BuiltInObject(h) :- GlobalSym("Array", h).
% ...

GlobalSym(m,h) :- PtsTo("global", g),
                  HeapPtsTo(g,m,h).
```

Listing 1: Vulnerability 1 in GateKeeper

```
1  G.skipZeroOrMore()
2  .state({
3    node:{
4      expression:{
5        left:{
6          properties: '?props',
7          mainObjectName: 'String'
8        }
9      }
10   },
11   filters:[
12     cond('contains', '?props', 'prototype')
13   ]
14 })
```

Listing 2: Vulnerability 1 in JS-QL

The JS-QL policy only detects writes to the **String** object. The implementation of JS-QL also contains a compound policy to detect writes to *all* built-in objects' **prototype** property.

Discussion.

JS-QL proves to be able to specify the prototype poisoning vulnerability and similar vulnerabilities. The GateKeeper query is more verbose compared to its JS-QL equivalent, indicating that JS-QL queries can be concise while remaining expressive.

Other vulnerabilities and conclusion

Vulnerabilities V2 and V3 were also expressible in JS-QL. GateKeeper excels in writing concise queries to detect certain individual properties of a program. It is however difficult to specify a query in GateKeeper which finds a sequence of properties in a program. JS-QL does not have this limitation, as it is designed to match states along an abstract flow graph. While JS-QL can also express individual properties of a program such as calls of a certain method, it is also capable of finding complex patterns.

Two other features that JS-QL offers and GateKeeper lacks is filtering and defining extra properties. It would be very cumbersome to write a query in GateKeeper to find all function calls to methods that take more than four arguments, which can be considered as a code smell. JS-QL provides the **properties** and **filters** constructs to express this.

Although the query for detecting V1 is actually a bit shorter when expressed in JS-QL, GateKeeper is less verbose in most situations. This is because data flow analysis happens behind the scenes in GateKeeper, whereas JS-QL has to do the checks for aliasing in the language itself.

5.2 PidginQL

Johnson et al. [24] presents a query language, PidginQL, for querying a program dependence graph (PDG). This type of graph is different from a flow graph generated by JIPDA because it only depicts dependencies between program statements, rather than modeling the whole execution of a program.

Vulnerability 4: CMS non-administrator sends message to all CMS users

In a scenario where not only administrators can broadcast messages, a regular user with bad intentions could easily take advantage of this situation to cause harm to the system. For instance, a CMS application with a large number of users could be exploited by sending a message to all users asking them to provide sensitive information such as their password. The attacker can then compromise the contents of the victim's account. To avoid this undesirable behavior, we need a policy which prevents regular users from sending messages to a large number of other users.

Pidgin.

Listing 3 is the Pidgin query that detects this vulnerability. First, the query searches for all nodes that are entries of the `addNotice` method and stores them in a variable. The `addNotice` method is the method that sends messages to all users. Next, all points in the PDG are found that match a return node of the `isCMSAdmin` method with a return value which is truthy. In order to know if there exists some path in the graph where `addNotice` is called when the return value of `isCMSAdmin` is false, all paths between the nodes in `addNotice` and `isAdmin` are removed from the graph for all paths where `isAdmin` is true. Finally, the intersection of the nodes in this "unsanitized" graph and the nodes in the `sensitiveOps` argument is taken, the latter representing a broadcast in this case. When this intersection is not empty, we assume that vulnerability exists in the remainder of the graph. This last part is checked by the `accessControlled` method.

JS-QL.

When attempting to write a similar query in JS-QL, we need to define the problem in terms of control flow: "There must be no path between the returns of `isCMSAdmin` when the return value is false, and a call of the `addNotice` method." With abstract interpretation, a value can be both true and false at the same time, which is why we have to inspect the nodes in the flow graph. When looking at a conditional (like an `IfStatement`), we can determine whether the true of false branch has been taken by comparing the first node of the branches with the alternate/consequent of the conditional. This can be seen on lines 2 and 6 in listing 4, where the `?alt` variable of the `IfStatement` gets matched with one of the successive states, ensuring that that state is the beginning of the false branch. We bind the context of the branch state to *?kont* and the stack to a *?lkont*. The next time we encounter a state with the same context and stack,

```
let accessControlled(G, checks, sensitiveOps) =
    G.removeControlDeps(checks) ∩ sensitiveOps is
        empty

let addNotice = pgm.entriesOf("addNotice") in
let isAdmin   = pgm.returnsOf("isCMSAdmin") in
let isAdminTrue = pgm.findPCNodes(isAdmin,TRUE) in
                  pgm.accessControlled(isAdminTrue,
                      addNotice)
```

Listing 3: Vulnerability 4 in PidginQL

```
1 G.skipZeroOrMore()
2 .ifStatement({alt:'?alt'})
3 .skipZeroOrMore()
4 .functionCall({name:'isCMSAdmin'})
5 .skipZeroOrMore()
6 .state({node:'?alt', kont:'?k',lkont:'?lk'})
7 .not().endIf({kont: '?k', lkont:'?lk'}).star()
8 .functionCall({name:'addNotice'})
```

Listing 4: Vulnerability 4 in JS-QL

we know that the end of the branch has been reached. Lines 8-9 indicate that we only wish to find the calls to `addNotice` before the end of the branch.

Discussion.

While this query finds all cases where `isCMSAdmin` is false, it will not detect calls to `addNotice` outside this test. We can solve this by finding all calls to `addNotice`, but this leads to false positives. The situation would be improved if a means to express the *XOR* relation between results of the JS-QL policies existed. If we had this kind of mechanism at our disposal, we could search for all calls to `addNotice` and all calls to `addNotice` that happen in the true branch of `isCMSAdmin`, and remove all states that occur in both results. The result after removal would then only contain occurrences of the vulnerability. Currently, operations for combining queries are not supported in JS-QL, as this would require an other layer of abstraction over query results. Although feasible, combining of queries is out of the scope of this paper.

Other vulnerabilities and conclusion

We were able to express vulnerabilities V4, V5, and V6 in JS-QL, but not without difficulty. Vulnerabilities V4 and V6, when expressed in our specification language, introduce results containing false positives. Detecting these two vulnerabilities each required two separate queries. If we wish to attain a result set without false positives, we could take the exclusive disjunction of the result sets of these separate queries.

The PidginQL language is best at expressing queries that deal with the dependencies between nodes in their program dependence graph. This type of graph is very powerful to check the control and data flow between two parts of code [17], but it is more difficult to use it to detect more general properties about a program.

For JS-QL, it is the other way around. Our approach allows us to detect a wide range of general and complex properties about a program, but it sometimes has troubles detecting dependencies between states with only one query. While PidginQL may be powerful in finding dependencies as described above, it does not return much meaningful information about the found vulnerabilities. Where JS-QL returns all nodes representing occurrences of a vulnerability, PidginQL only indicates the presence or absence of vulnerabilities without specifying actual nodes that represent occurrences.

Another restriction in PidginQL is that there is no way to reason about the internals of a state in the graph. This expressiveness and flexibility comes at the cost of JS-QL queries often being more verbose.

5.3 ConScript

ConScript [29] is a client-side advice implementation for security. The language allows the hosting page to express fine-grained application-specific security policies which are enforced at runtime.

Vulnerability 7: String arguments to setInterval, setTimeout

In JavaScript, `setInterval` and `setTimeout` take a callback function as a first argument that is fired after a certain interval of time. However, a string argument can also be passed as the first argument, as Listing 5 illustrates. This potentially allows attackers to pass malicious code as a string argument to `setInterval/setTimeout`, which can lead to security threats.

```
1  var f = function(){}
2  var i = 1;
3  var s = "stringgy"
4  var o = {};
5  setTimeout(i, interval);
6  setTimeout(s, interval);  //Violation
7  setTimeout(o, interval);
8  setTimeout(f, interval);
```

Listing 5: String arguments to setTimeout

ConScript.

ConScript is an aspect-oriented advice language that is able to detect security violations such as the one depicted in listing 5. The aspects are written in JavaScript, which enables the programmer to make full use of the language. The ConScript language also provides a type system to assure that the policies are written correctly, as can be seen on line 1 in Listing 6. Lines 10-11 depict the actual registration of the advice on the `setInterval` and `setTimeout` functions. When called, the `onlyFnc` function is triggered instead, which checks if the type of the argument is indeed of type "function". `curse()` has to be called within the advice function, disabling the advice in order to prevent an infinite loop, but has no additional semantic value for the policy itself.

JS-QL.

Because JS-QL queries range over a flow graph constructed as the result of an abstract interpretation, it must be the case that the used value abstraction does not abstract away the type of a value. This can be accomplished by using a type lattice that abstracts concrete values to the set of types they represent, and this is also the default abstract value lattice in JS-QL. In this lattice, abstract value `{Str,Num}` for example represents a concrete value that is either a string or a number. We can define a `isString` helper function that checks whether a variable *may* be of type string by using a membership test. The JS-QL query in listing 7 uses this function to determine whether the value of

```
1  let onlyFnc : K x U x U -> K =
2  function (setWhen : K, fn : U, time : U) {
3      if ((typeof fn) != "function") {
4          curse();
5          throw "The time API requires functions as
               inputs.";
6      } else {
7          return setWhen(fn, time);
8      }
9  };
10 around(setInterval, onlyFnc);
11 around(setTimeout, onlyFnc);
```

Listing 6: Vulnerability 7 in ConScript

```
G.skipZeroOrMore()
.functionCall({
  name:'setTimeout',
  arguments:'?args',
  properties:{
    '?arg' : prop('memberOf', '?args'),
    '?name' : '?arg.name',
  },
  lookup:{'?name': '?lookedUp'},
  filters:[
    cond('isString', '?lookedUp')
  ]
})
```

Listing 7: Vulnerability 7 in JS-QL

the `?name` variable may be of type String or not. The query looks for a call of the `setTimeout` function and binds its arguments to `?args`. Function `memberOf` creates a new substitution set for each of the elements in the list that it takes as an argument. This allows the inspection of each individual argument `?arg` of the `setTimeout` function. We take the name of the argument and look up its value in the `lookup` clause. The query then filters out the string arguments, as already discussed above. This query will only detect the actual violation on line 6 in listing 5.

Discussion.

Both approaches are able to express the security vulnerability in a concise way. The JS-QL query however is more readable than its ConScript counterpart, as it requires less boilerplate code and does not use type system annotations.

Other vulnerabilities and conclusion

It is not straightforward to compare JS-QL to ConScript, as ConScript checks for policy violations using dynamic analysis. We were able however to express vulnerabilities V7, V8, and V9 in JS-QL, and we can therefore compare the expressiveness of the queries written in each language.

The ConScript language applies advices around function calls, changing the behavior of the program if the function call was prohibited. The aspect-oriented approach allows ConScript to specify what actions that need to be taken when a violation is detected. We can not express this in JS-QL, but this is also not necessary because we detect vulnerabilities at compile-time rather than at runtime. Field accesses can also be expressed as function calls, so ConScript can reason about getting and setting values as well. JS-QL can also reason about getting and setting values, but has access to more information thanks to the underlying flow graph.

The advice functions written in ConScript have full access

	–GateKeeper–			–PidginQL–			–ConScript–		
	V1	V2	V3	V4	V5	V6	V7	V8	V9
JS-QL	✓	✓	✓	❍	✓	❍	✓	✓	✓

✓: Fully expressible, ❍: Expressible with false positives
Table 2: Expressiveness of JS-QL

to the JavaScript language, making them very flexible in behavior. By using JavaScript instead of a DSL, the policies themselves are also quite verbose, because for each policy a JavaScript function has to be created. This does allow users of ConScript to define properties and filters, as in JS-QL. However, the advice approach limits ConScript to detect only function calls, which considerably reduces expressiveness.

Querying for multiple sequential lines of code is difficult in ConScript. Where a JS-QL query could easily be written to detect a function call to method `X` after reading variable `Y`, Conscript has to define variables that function as a "boolean". The variable is set to true when `Y` is read. The advice around `X` then has to check the value of `Y` before deciding what action to perform.

We argue that JS-QL queries are more expressive than their ConScript counterparts when it comes down to the detection of different kinds of program states. JS-QL also proves flexible in terms of specifying properties and filters, but is not as flexible as ConScript because the latter has full access to the JavaScript language once an advice is triggered. Both languages are quite verbose because of the expressiveness they provide.

5.4 Overall evaluation

We evaluated the JS-QL language by expressing 9 different security vulnerabilities originating from 3 different papers. Each of the 9 vulnerabilities could be checked in under 3 seconds. The results of this evaluation are summarized in Table 2. From these results we can conclude the following:

- JS-QL can be used to express a wide variety of security vulnerabilities in a readable and flexible way.
- JS-QL is limited in expressiveness for detecting dependencies between states.

Generally speaking, JS-QL is capable of expressing any security vulnerability that can be detected in a flow graph computed by the underlying abstract interpretation. Security properties that for example require relations between flow graphs (e.g., equivalence properties [4]) fall outside the scope of JS-QL.

5.4.1 Advantages of JS-QL

Granularity.

A key advantage of the JS-QL tool is the ability for programmers to define queries and vulnerabilities that are as general or specific as needed. Starting from the `state` predicate, complex patterns can be expressed and wrapped in a self-named predicate. Flexibility is key in these predicates because the user can specify which properties he exposes through the predicate. These properties can then be queried by passing metavariables as arguments, which are bound when a match is found. Literals and metavariables that are already bound act as filters for the predicates, as in any declarative language.

Flexibility.

The JIPDA graph contains states with information of arbitrary depth. Therefore, JS-QL has to provide access to all these levels of information. The flexibility required for doing so means that JS-QL is not bound to one particular graph type, but that all types of graphs with labeled nodes and edges can be used with little to no modification of the tool itself. Only a reification layer of the new graph needs to be provided, mapping the states of the graph to the format our tool uses.

Negation.

JS-QL, in contrast to many other query languages, offers negation as a feature to increase expressiveness, albeit in a limited way (see below).

Recursion.

Another non-trivial feature of the JS-QL tool is the possibility to recursively define queries. This type of queries are especially useful for following a trace of information starting at a certain point, such as all aliases of a certain variable.

5.4.2 Limitations of JS-QL

Negation.

Negation is subject to some limitations in the current implementation of JS-QL. It is limited to only one state, which can be insufficient in some cases. Variables bound in a negated state are only visible to that state and will thus not be included in the resulting substitutions. Negating sequences of states wrapped in braces is also not supported. Unrestricted negation would make JS-QL more expressive and is a topic of future research.

Performance and Scalability.

The performance and scalability of our implementation is largely determined by the performance and scalability of the underlying abstract interpretation [19] and the graph query engine [26]. Currently, all test programs and queries are small and run within reasonable time (in under 3 seconds), but we expect that for larger programs and queries the runtime of the tool would increase significantly.

Our current implementation has computational overhead when patterns can be matched multiple times in the algorithm. Each match is a computationally heavy operation, which means that we should try to avoid matching more than once. This could be done by memoizing the substitutions between all already matched pairs of state and pattern, decreasing the computational overhead drastically and making the tool scale in the size of the input program and speed of query results.

6. CONCLUSION

This paper proposes a query-based technique and tool for detecting security vulnerabilities in JavaScript programs. Unlike most existing tools, our tool can be configured to detect vulnerabilities that are specific to a single application instead of being limited to a fixed set of pre-encoded rules. For this purpose we introduced JS-QL, an expressive specification language that enables users to write succinct and application-specific queries to test their applications against vulnerabilities. JS-QL matches queries against an abstract flow graph of a statically analyzed program. We use abstract interpretation as a static analysis technique to generate this graph containing program information. JS-QL is a domain-specific language embedded in JavaScript and is based on the concept of regular path expressions. These expressions are similar to traditional regular expressions, except that they can be applied to find certain paths in a graph instead of finding patterns in a string.

We evaluated our specification language by expressing multiple security vulnerabilities and comparing the resulting specifications to corresponding ones of existing work. The results of the evaluation indicate that the static analysis of our technique is sufficiently powerful and its specification language sufficiently expressive for expressing application-specific vulnerabilities commonly found in related work. We conclude from our experiments that our language is apt for specifying several types of security vulnerabilities. The combination of abstract flow graphs and regular path expressions proves to be an effective means to obtain program information and specify security vulnerabilities. Specifications in JS-QL are often more readable, concise, and equally expressive compared to their formulation in other languages.

The implementation of JS-QL is publicly available[1] and can be used to test source code for general characteristics and security vulnerabilities. Despite its current limitations, we believe that our tool represents a step in the right direction for allowing users to detect security vulnerabilities in their JavaScript applications.

Acknowledgments

Jens Nicolay is funded by the SeCloud project sponsored by Innoviris, the Brussels Institute for Research and Innovation.

7. REFERENCES

[1] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International journal on digital libraries*, 1(1):68–88, 1997.

[2] M. Appeltauer and G. Kniesel. Towards concrete syntax patterns for logic-based transformation rules. In *Proceedings of the Eighth International Workshop on Rule-Based Programming (RULE 2007)*, 2007.

[3] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[4] B. Blanchet. Security protocol verification: Symbolic and computational models. In *Proceedings of the 1st International Conference on Principles of Security and Trust (POST 2012)*, pages 3–29, 2012.

[5] T. Cohen, J. Y. Gil, and I. Maman. JTL: the Java Tools Language. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 89–108, 2006.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, 1977.

[7] R. F. Crew. ASTLOG: A language for examining abstract syntax trees. In *Proceedings of the 1997 USENIX Conference on Domain-Specific Languages (DSL 1997)*, pages 229–242, 1997.

[8] O. de Moor, D. Lacey, and E. V. Wyk. Universal regular path queries. *Higher-order and Symbolic Computation*, 16(1-2):15–35, 2003.

[9] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. The SOUL tool suite for querying programs in symbiosis with eclipse. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ 2011)*, 2011.

[10] K. De Volder. JQuery: A generic code browser with a declarative configuration language. In *Proc. of the 8th Int. Symp. on Practical Aspects of Declarative Languages (PADL 2006)*, pages 88–102, 2006.

[11] S. Drape, O. de Moor, and G. Sittampalam. Transforming the .NET intermediate language using path logic programming. In *Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2002)*, 2002.

[12] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 391–400, 2008.

[13] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, Oct. 2000.

[14] H. Falconer, P. H. J. Kelly, D. M. Ingram, M. R. Mellor, T. Field, and O. Beckmann. A declarative framework for analysis and optimization. In *Proceedings of the 16th International Conference on Compiler Construction (CC 2007)*, 2007.

[15] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of Programming languages (POPL 1987)*, 1987.

[16] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for a web-site management system. *ACM SIGMOD Record*, 26:4–11, 1997.

[17] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[18] M. Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.

[19] T. Gilray, S. Lyde, M. D. Adams, M. Might, and D. V. Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43th Annual ACM Symposium*

[1]https://github.com/voluminat0/Jipda-Security

on the Principles of Programming Languages (POPL 2016), January 2016.

[20] S. Guarnieri and B. Livshits. GATEKEEPER: mostly static enforcement of security and reliability policies for JavaScript code. In *Proceedings of the 18th conference on USENIX security symposium (SSYM 2009)*, pages 151–168, 2009.

[21] S. Günther and T. Cleenewerck. Design principles for internal domain-specific languages: A pattern catalog illustrated by ruby. In *Proceedings of the 17th Conference on Pattern Languages of Programs (PLOP 2010)*, 2010.

[22] E. Hajiyev, M. Verbaere, and O. de Moor. CodeQuest: Scalable source code queries with Datalog. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP 2006)*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27, 2006.

[23] H. He and A. K. Singh. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the 2008 International Conference on Management of Data (SIGMOD 2008)*, 2008.

[24] A. Johnson, L. Waye, S. Moore, and S. Chong. Exploring and enforcing security guarantees via program dependence graphs. *SIGPLAN Not.*, 50(6):291–302, June 2015.

[25] D. Lacey. *Program Transformation using Temporal Logic Specifications*. PhD thesis, University of Oxford, August 2003.

[26] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. *SIGPLAN Not.*, 39(6):219–230, June 2004.

[27] J. Magazinius, P. H. Phung, and D. Sands. Safe wrappers and sane policies for self protecting JavaScript, year = 2010. In *Proceedings of the 15th Nordic Conference in Secure IT Systems (NordSec 2010)*, pages 239–255.

[28] M. Martin, B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: A program query language. *SIGPLAN Not.*, 40(10):365–383, Oct. 2005.

[29] L. A. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP 2010)*, pages 481–496, 2010.

[30] J. Nicolay, C. Noguera, C. De Roover, and W. De Meuter. Detecting function purity in JavaScript. In *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM 2015)*, pages 101–110, 2015.

[31] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh. Towards pattern-based design recovery. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2002)*, pages 338–348, 2002.

[32] T. Rho, G. Kniesel, and M. Appeltauer. Fine-grained generic aspects. In *Proceedings of the AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL 2006)*, 2006.

[33] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages (DBPL 2015)*, pages 1–10. ACM, 2015.

[34] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[35] D. Toliver. *500 Lines or Less*, chapter Dagoba: an in-memory graph database. Lulu, 2016.

[36] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: a scripting language for refactoring. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, 2006.

[37] N. Volanschi. Condate: a proto-language at the confluence between checking and compiling. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP 2006)*, pages 225–236, 2006.