Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science
and Applied Computer Science

Expressing and checking application-specific, user-specified security policies

Graduation thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Applied Sciences and Engineering: Computer Science

Valentijn Spruyt

Promotor: Prof. Dr. Coen De Roover
Advisors: Jens Nicolay
Quentin Stievenart

JUNE 2016

Vrije Universiteit Brussel

Faculteit Wetenschappen en Bio-Ingenieurswetenschappen
Departement Computerwetenschappen
en Toegepaste Informatica

Expressing and checking application-specific, user-specified security policies

Proefschrift ingediend met het oog op het behalen van de graad van
Master of Science in Applied Sciences and Engineering: Computer Science

Valentijn Spruyt

Promotor: Prof. Dr. Coen De Roover
Begeleiders: Jens Nicolay
Quentin Stievenart

JUNI 2016

Abstract

Acknowledgements

Contents

Chapter 1

Introduction

1.1 Motivation

1.2 Objective

1.3 Overview

Chapter 2

Detecting security vulnerabilities

In order to check for security vulnerabilities we first have to find a suitable way to represent a program. This representation has to contain specific information about the program to be able to answer questions about security vulnerabilities. The information we need in this dissertation is twofold:

1. A program can contain many branches, loops and other control structures. We need to know the exact order of execution along each path in the program before we can make assumptions about security vulnerabilities. Therefore information is needed about which functions can be applied at a call site. This type of information is called *control flow*.

2. Variables in JavaScript are mutable, so their values can change at any moment in a program. *Value flow* information tells us exactly what values an expression may evaluate to. This is very important w.r.t. security, as some harmless variable may become referenced to a malicious variable somewhere in the program. From there on, that variable should be marked as pointing to the same value as the malicious variable.

Aside from the representation, some technique has to be found to efficiently express security checks in the form of user-specified, application specific security policies. A naive way to examine programs would be to run them and keep track of any relevant information along the execution. Not only would this be tiresome, we can also not guarantee that the program will ever terminate, that it terminates without errors, or that it will have the same outcome for different inputs. A better approach would be to to analyze the program without having to run it. To this extent, a technique called *static analysis* can be used.

This chapter describes how static analysis can be used to examine programs and how this analysis can be addressed to obtain information about specific parts of a program. First, section 2.1 describes more precisely what static analysis is

and how it is interesting for this dissertation. Next, We discuss some approaches using static analysis to find generic vulnerabilities in programs in section 2.2. Finally, some application-specific approaches for checking security vulnerabilities are discussed. For these approaches we take a deeper look on how they query the information specified by the analyses they perform. We end this chapter by giving a brief conclusion.

2.1 Introduction to static analysis

Rice's theorem tells us that there is no general or effective method to prove non-trivial properties about a program. This problem is similar to the halting problem, which is undecidable. *Static analysis* is a technique for analyzing computer programs without having to execute them. In this way we can avoid the possible problems we might encounter using a naive technique, as described above. The results of the analysis indicate program defects or prove certain properties of the program. As proving non-trivial properties about a program is undecidable, static analysis focusses on the instances of the problem about which it can tell whether the program satisfies a property or not, and leaves other instances unsolved. The results of the static analysis will then be a useful set of approximate solutions. Figure 2.1 shows the main difference between a regular decider, which will always provide an exact answser, and a static analyzer.

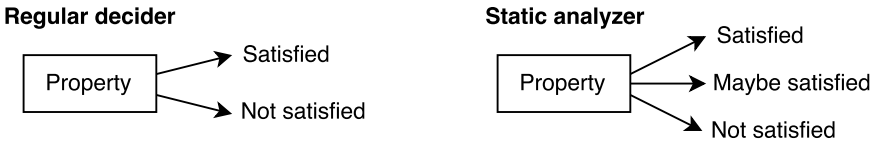

Figure 2.1: Proving program properties: Regular decider and static analyzer

Precision is very important in static analysis. Consider a static analyzer that concludes for each property that it is *maybe* satisfied. It is clear to see that there is no precision in this analysis, rendering it useless. We have to strive to attain enough precision to solve the maximum number of problem instances. *Speed* on the other hand is less important for static analysis. As static analysis is decidable, it is guaranteed that the analysis will run in finite time, but gathering precise results is much more meaningful than the performance of the analysis itself. One particularly interesting technique used for static analysis is *abstract interpretation*. This technique mimics interpretation of the program and allows to stay close to the original language semantics of those program without having to modify or instrument them to perform the analysis (in contrast to other static analysis techniques such as *symbolic execution*). This mimicing of programs fits well for this

dissertation, as we need to check for application-specific security vulnerabilities. It is thus a prerequisite that the semantics of the analyzed program lean as close to the original semantics as possible. A nice feature of abstract interpretation is that it allows to specify the precision needed by parameterizing it with e.g. a *lattice*.

2.1.1 Abstract interpretation

Abstract interpretation is a static analysis technique used to reason about a program. It does this by interpreting an approximation of a program through abstraction of its semantics. A *sound* analysis can be performed and the precision of this analysis can be adjusted to the user's needs through various mechanisms. This increase in precision comes at the cost of a greater analysis running time.

Abstract interpretation works in a similar way as normal program interpretation (so-called *concrete interpretation*). The concrete interpretation of a program can be described as follows: A program *e* can be injected into an initial state s_0, the entry point of the program. From this state other states can be reached using a *transition function*, until after several transitions a final state is reached. If no such state is ever reached, the execution will not terminate and hence will run indefinitely. The output of interpreting a program like this is a possibly infinite trace of execution states. The layout of this execution trace might depend on the input of the program or other changing values, making it useless for static analysis.

Abstract interpretation solves this by applying abstraction in order to compute a finite trace. Primitive values and addresses are *abstracted* to be made finite, resulting in something which is computable in finite time but less precise. Abstract interpretation is similar to concrete interpretation: A program is again injected, but this time into an *abstract state* $\hat{s_0}$. A transition from one state to another is done through an *abstract transition function*. The difference between this and a regular transition function is that an abstract state can make an abstract transition to multiple states. This is a consequence of the precision loss due to abstraction. Figure 2.2 shows the concrete and abstract interpretation traces for `while(x < 5){ x--;}`. We assume that for the concrete case `x` is smaller than 5 when it reaches the code. The program will then never terminate, leading to an infinite execution trace. For the abstract case, we assume that `x` is abstracted. We see that in abstract state $\hat{s_3}$ the program can go to either $\hat{s_4}$ or $\hat{s_4}'$, and that the (possibly infinite) `while` loop is represented as a loop in the abstract state graph. This finite representation of a program (which is actually an abstract state graph) proves to be useful to provide answers to non-trivial questions about the program.

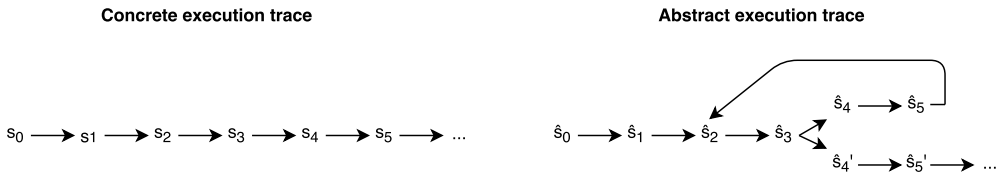

Figure 2.2: Traces of concrete and abstract interpretation

2.1.2 Mathematical background

In order to fully understand abstract interpretation, we will first look at the mathematical concepts it relies on. The concepts defined in this section will aid us in formally defining an `abstraction`. This definition is needed to understand how precision is caused by abstracting values.

Definition 1. *A relation* $\sqsubseteq$*:* $S \times S$ *is a* ***partial order*** *if it has the following characteristics:*

1. *Reflexivity:* $\forall x \in S : x \sqsubseteq x$
2. *Transitivity:* $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
3. *Anti-symmetry:* $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

Definition 2. *A* ***partially ordered set*** *(*S*,*$\sqsubseteq$*) is a set with a partial order*

Definition 3. *For a subset* $X \subseteq S$*,* u *is an* ***upper bound*** *of* X *if* $u \in S, \forall x \in X : x \sqsubseteq u$*.* u *is the* ***least upper bound*** *of* X *(*$\bigsqcup X$*) if for every upper bound* x*,* $u \sqsubseteq x$*. Similarly, the* ***lower bound*** *of* X *can be defined as:* $l \in S, \forall x \in X : l \sqsubseteq x$*.* l *is the* ***greatest lower bound*** *of* X *(*$\bigsqcap X$*) if for every lower bound* $x, x \sqsubseteq l$*. Two important operators on partial orders are* ***join*** *(*$\sqcup$*) and* ***meet*** *(*$\sqcap$*).* $x \sqcup y$ *denotes* $\bigsqcup\{x, y\}$*, the least upper bound of* x *and* y*,* $x \sqcap y$ *denotes* $\bigsqcap\{x, y\}$*, the greatest lower bound of* x *and* y*.*

Definition 4. *A* ***lattice*** $(L, \sqsubseteq)$ *is a partially ordered set in which any two elements have a least upper bound and a greatest lower bound. A* ***complete lattice*** $(C, \sqsubseteq)$ *is a partially ordered set in which all subsets have a least upper bound and a greatest upper bound. A complete lattice includes two special elements: a* ***bottom*** *element* $\bot = \bigsqcap C$ *and a* ***top*** *element* $\top = \bigsqcup C$*.*

Definition 5. *A* ***Galois connection*** *is a particular correspondence between two partially ordered sets* $(A \sqsubseteq_A)$ *and* $(B \sqsubseteq_B)$*. More precisely this correspondence is a pair of functions: the* ***abstraction function*** $\alpha : A \to B$ *and the* ***concretization function*** $\gamma : B \to A$*, such that* $\forall a \in A, b \in B : \alpha(a) \sqsubseteq_B b \Leftrightarrow a \sqsubseteq_A \gamma(b)$*.*

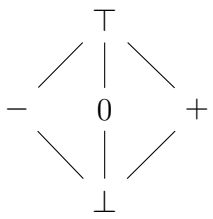

Figure 2.3: Partially ordered set of signs (complete lattice)

2.1.3 Abstraction

An *abstraction* $\hat{X}$ of a concrete set X in abstract interpretation is a Galois connection between the power set of X $(\mathcal{P}(X), \subseteq)$ and X itself $(X, \sqsubseteq)$. The abstraction function α maps a concrete value to its abstract counterpart, whereas the concretization γ function maps an abstract value to its concrete counterpart*s*. Abstract values, sets and operations are generally indicated with a hat. The following abstraction example illustrates how abstraction works, and how it causes *imprecision* to occur.

Example (Sign abstraction). A possible abstraction of integers $\mathbb{Z}$ could be to map them onto the set of signs $\widehat{Sign}$. The set of signs forms a complete lattice with $\sqsubseteq$ ordering, as depicted in figure 2.3. This abstraction could be used in an analysis to detect divisions by zero, for example. We can define the abstract and concretization functions as follows:

$$\begin{aligned}
\alpha : \mathcal{P}(\mathbb{Z}) &\to \widehat{Sign} \\
\alpha(Z) &= \bot \text{ when } Z = \varnothing \\
&= 0 \text{ when } Z = \{0\} \\
&= + \text{ when } \forall z \in Z, z > 0 \\
&= - \text{ when } \forall z \in Z, z < 0 \\
&= \top \text{ otherwise}
\end{aligned}$$

$$\begin{aligned}
\gamma : \widehat{Sign} &\to \mathcal{P}(\mathbb{Z}) \\
\gamma(P) &= \varnothing \text{ when } P = \bot \\
&= 0 \text{ when } P = \hat{0} \\
&= \mathbb{Z}^+ \text{ when } P = + \\
&= \mathbb{Z}^- \text{ when } P = - \\
&= \mathbb{Z} \text{ otherwise}
\end{aligned}$$

The addition operator $+\colon \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ can also be abstracted to $\hat{+}\colon \hat{\mathbb{Z}} \times \hat{\mathbb{Z}} \to \hat{\mathbb{Z}}$ following the rules of sign. Some examples:

$$\{0\}\hat{+}\{+\} = \{+\}$$
$$\{-\}\hat{+}\{-\} = \{-\}$$

but for more advanced examples, we can easily see a loss in precision:

$$\{+\}\hat{+}\{-\} = \{-, 0, +\}$$
$$\{0\}\hat{+}\{0, +\} = \{0, +\}$$

When applying the concretization function after the abstraction function, we observe that the result is less precise. Consider the negation function $f(N) = \{-n | n \in N\}$, which negates an integer. Applying the concretization function to an integer directly results in no loss of precision:

$$f(\{1\}) = \{-1\},$$

whereas applying it after the application of the abstraction function overapproximates the concrete value:

$$(\gamma \circ f \circ \alpha)(\{1\}) = \mathbb{Z}^+$$

$\mathbb{Z}^+$ is an overapproximation of $\{1\}$, conserving all properties that hold for $\{1\}$. The closer something is abstracted to its concrete value, the higher the precision of the analysis will be.

2.1.4 conclusion

To conclude this section, we briefly discuss why this loss of precision is important for this dissertation. When calculating value flow by performing static analysis through abstract interpretation, precision is also lost. In most programming languages, variables and objects point to values and addresses respectively. For this, JavaScript is no exception. It is obvious to see that simple calculations lose precision as in the example above. As a result of abstract interpretation, multiple addresses may point to the same object. This is exactly the kind of imprecision that is introduced in the analysis that is used in this dissertation. As each of these addresses is as valid of an address as any other, we need to consider all addresses of matched variables/objects on all matched paths in the state graph.

2.2 Support for generic vulnerabilities

Static analysis is often used by model checkers to verify if a program satisfies a set of properties (i.e. a specification of a program). These tools often require

additional information to be added to the program before being able to analyze it. The OWASP LAPSE+ plugin for Eclipse[32] for example requires the user to annotate all possible vulnerability sources and sinks in the source code. It then checks if there is information flow between a source and a sink. Although applicable to many programs, tools for finding general characteristics of programs is limited in several ways: The set of problems that these tools can detect is often restricted and a lot of tools support detection for similar problems. Tools detecting bug patterns detect only the patterns that are pre-encoded in the tool. This implies that the tool only supports bug patterns that are pre-encoded and thus will most likely miss any bug pattern that isn't already encoded. Additionally, poorly encoded patterns may miss bugs, making the analysis of the tool unsound. Adding or extending functionality to existing solutions is often cumbersome and in most cases even impossible to do manually, which makes these solutions less useful for certain domain-specific programs, as these require a flexible tool. To this extent, a more practical approach would be to develop a tool which allows users to define by themselves what they wish to detect in a program. This approach would make the analysis *application-specific* and the detection rules would be *user-defined*. More about application-specific approaches can be found in section 2.3. The remainder of this section discusses the most popular static analysis approaches for model checking and finding generic code characteristics and vulnerabilities.

JOANA

The Java Object-Sensitive Analysis project (JOANA[36]) is an eclipse plugin which checks for security leaks in Java programs. The tool supports all Java language features (except for reflection) and scales well for larger programs. The analysis they use is flow-sensitive, context-sensitive, object-sensitive and lock-sensitive, minimizing the amount of false positives drastically. The types of security flaws JOANA is able to detect are:

1. *Confidentiality*: Information about sensitive values, like passwords or personal data, should in no case be conveyed to public outputs.

2. *Integrity*: The dual of confidentiality: In no way should unsafe program inputs alter secure data or influence sensitive computations of a program.

These flaws are detected by creating a system dependence graph (SDG) of the program on which information flow between sources and sinks is checked through program slicing. The SDG is an overapproximation of the information flow through the program. A benefit of this kind of graph is that it is able to detect direct (data dependencies) as well as indirect (control dependencies) dependencies. In order for the analysis to run, the user has to specify which parts of a program should

act as sources and which should acts as sinks. This is done by adding annotations to the source code. JOANA comes with a machine-checked soundness proof. Although JOANA is good in what it does, it is limited in the amount of vulnerabilities it detects and there is no way to extend the tool to support more vulnerabilities.

Flawfinder

Flawfinder[40] is a tool for examining C/C++ source code and detecting security weaknesses. It comes with a database of well-known problems, such as buffer overflow risks and race conditions. The results of the analysis performed is a report of all found flaws with a corresponding security risk level. Although being useful to quickly check for security vulnerabilities, flawfinder is not flexible as it is not extensible, nor is it aware of the semantics of the system under test. Control flow and data flow analysis aren't supported by the tool, making it rather a rather naive approach.

FindBugs

FindBugs[19] is a static analysis tool for detecting bugs in Java programs. They detect many classes of bugs by checking structural bug patterns against a program's source code. These classes of bugs can be subdivided into three main classes: Correctness bugs, dodgy confusing code and bad practices. Recently, the Find Security Bugs plugin[1] was developed on top of Findbugs. This plugin can detect 80 different (pre-encoded) vulnerability types, among which are the top 10 OWASP security vulnerabilities. An example security violation detected by the plugin is the parsing of an untrusted XML file. The contents of this file might be malicious ant thus poses as a risk for the application. As the plugin is able to detect a wide range of bugs, it an ideal tool to checking for the most common security vulnerabilities. Nevertheless, only those vulnerabilities can be detected, and when a new class of security violations arises, there will be no way to add detection rules for them as a regular user.

CodeSonar

CodeSonar[22], developed by GrammaTech, is a proprietary source code analysis tool that performs an unified data flow and symbolic execution analysis for C, C++ and Java programs. They claim to detect more code flaws than the average static analysis tool because they don't rely on pattern matching or other similar approximations. The approach they use is to compile source code and generate

[1] http://find-sec-bugs.github.io

several intermediate representations, such as control flow graphs, call graphs and AST's. They then traverse/query these models to find particular properties or patterns that indicate defects. Next to performing general checks, CodeSonar provides an C API which gives access to its intermediate representations of the compiled program. A user can then define custom checks on these representations. We can't verify the ease of use of these custom checks as we couldn't find any examples. The hybrid approach of CodeSonar (general checks *and* application-specified checks) preludes the next section of this chapter, which discusses approaches that support detecting application-specific vulnerabilities through user-defined queries and rules.

2.3 Support for application-specific vulnerabilities

The problem with tools supporting detection of generic vunerabilities is that they often don't allow users to write their own rule or queries to find domain- and/or application-specific flaws. Even if some mechanism for specifying user-defined rules is available, as in PMD[2] for example, it often is cumbersome to write them in the tool's input language. This limitation makes that these tools are often not very flexible, and it makes it hard to extend them.

In this section we discuss how putting the detection of application-specific characteristics and vulnerabilities in the hands of the application developer can be fruitful, by presenting some approaches which allow users to define their own program queries. Two main considerations for creating such a tool are (i) the way a program is represented and (ii) how the user is given access to this representation. The following approaches each have their own techniques to do so, and we will elaborate on their advantages and disadvantages.

PQL

The PQL language is designed to check if a program conforms certain program design rules[28]. More precisely, it can be used to check the presence of sequences of events associated with a set of related objects. The language allows programmers to query for these types of sequences in an application-specific way, rendering very useful to detect design defects on a per application basis.

Either dynamic or static analysis can be used to solve these PQL queries, but only the latter is of interest for this dissertation. The static analyser described uses a context-sensitive, flow-insensitive, inclusion-based pointer analysis. As PQL attempts to optimize results of the static analysis to use them in the dynamic

[2] https://pmd.github.io

analysis, the used analysis must be sound. The points-to information, together with the program representation, is stored as datalog rules in a deductive database called *bddbddb*. This is very similar to the approach of GateKeeper[14].

Two interesting features of the PQL language are the support for subqueries and the ability to react to a match. The latter is only useful in the case we use dynamic analysis. Subqueries however add significant power to a language. In the case of PQL, they allow users to specify recursive event sequences of recursive object relations. Figure 2.4 shows how a recursive query is written in the PQL language.

```
query derivedStream(object InputStream x)
returns object InputStream d;
uses object InputStream t;
matches {
    d := x
    | {t = new InputStream(x);
      d := derivedStream(tmp);}
}

query main()
returns method * m;
uses
    object Socket s;
    object InputStream x, y;
    object Object v;
matches {
    x = s.getInputStream();
    y := derivedStream(x);
    v = y.readObject();
    v.m();
}
executes Util.PrintStackTrace(*);
```

Figure 2.4: A PQL recursive query for tracking data from sockets

The actual matching of queries to these rules happens by first translating the PQL queries to the corresponding datalog queries. This happens automatically, to shield the user from dealing with the points-to information or the datalog program representation directly. Once translated, the queries get resolved by the *bddbddb* system, after which te results are ready to be interpreted by the user. One thing to note is that since the analysis used is flow-insensitive, sequencing is not supported in such a way that the user can distinguish whether program point `a` happens before or after program point `b`. The same goes for negation: No guarantees about ordering can be made, which means that one can not deduce that an excluded event (i.e. a negated event) happens between two points in a sequence. They solve this by ignoring all excluded events, in order to maintain soundness of their approach.

A benefit of using Datalog is that it is very efficient and has way less overhead compared to fully fledged declarative programming languages. Storing an analysis/program as datalog rules may be efficient, but it is hard to get a good overview of the program by just looking at these rules. PQL closes the gap between the lack in readability of plain datalog rules and writing clean application-specific queries

by introducing its own language. However, this language feels somewhat verbose and the syntax is something the user has to get used to.

Pidgin

Pidgin[24] is a program analysis and understanding tool which allows users to specify and enforce application-specific security guarantees. In their approach they generate a *Program Dependence Graph* (PDG) of programs by using an interprocedural data flow analysis (object-sensitive pointer analysis). This kind of graph contains all information about how data data flows through a program. More precisely, each pair of connected nodes indicates that the second node of the pair depends in some way on the first node. Figure 2.5 shows a PDG of a guessing game.

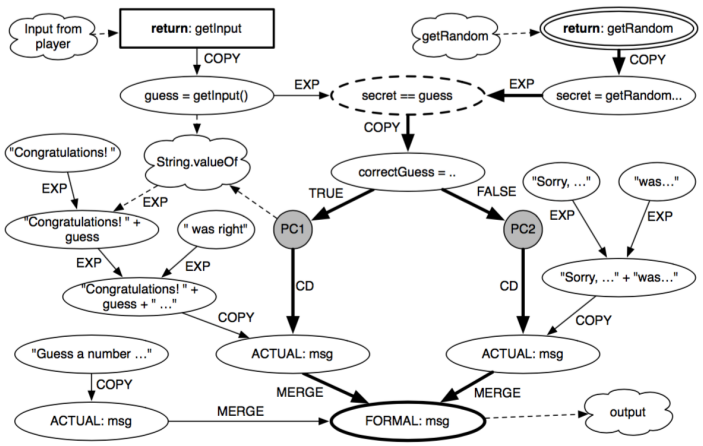

Figure 2.5: Program dependence graph of a guessing game

We can see that the value that flows to the output function indirectly depends on the value of the secret. This can be seen as a security violation and can be queried for in the Pigdin language. To this extent, the Pidgin Query Language (PidginQL) defines specialized constructs that allow the user to retrieve all relevant information from the PDG. As the program is represented as such a graph, the approach specializes in detecting if there is information flow from one node to another. An example of a PidginQL query is seen in listing 2.1. The usual approach to writing queries goes as follows: First, the nodes between which information flow needs to be detected are specified and stored in variables. This is done through the aforementioned constructs. After specifying these *source* and *sink* nodes, another type of nodes can also be stored in a variable, namely *declassifier* nodes. These nodes act as 'sanitizers' in a way that when information flows

between a source and a sink and this information also flows through a sanitizer, then the flow is allowed. When all node have been captured in variables, declassifier nodes can be removed from the graph. The resulting graph is now modified in such a way that only the user-specified declassifier nodes are are removed. On this graph the final check is done if there is any flow left between the specified sources and sinks.

```
1 //source, sink and declassifier
2 let source = pgm.returnsOf("getRandom") in
3 let sink = pgm.formalsOf("output") in
4 let declassifier = pgm.forExpression("secret == guess") in
5 //Remove declassifiers and check for flow
6 pgm.removeNodes(declassifier).between(source, sink)
7 is empty
```

Listing 2.1: A typical PidginQL query

The PidginQL language is expressive and powerful for detecting information flows. The language however lacks expressiveness when it comes to inspecting nodes. There are no constructs that allow the user to only find nodes with a certain name, for example. A second limitation is the result of queries. Most static analysis tools report all found violations, with more information about the violating code. PidginQL on the other hand just indicates whether there is flow or not (remember the `is empty` construct on line 7).

Metal

Metal[**Metal**] can best be described as a general analysis tool of programs for which a user has to write application-specific extensions (also called checkers). These extentions are then executed as a traditional dataflow analysis, but can be augmented in ways outside the scope of traditional approaches. Metal extensions are applied depth-first to the control-flow graph of a function. This flow graph is computed from the AST of the program. By applying an extendion depth-first, each program point down a single path is checked. This process is very similar to pattern matching. Checkers in the Metal language consist of two parts: The actual code that describes the checker and a corresponding state machine. They distinguish two types of state machines: *Global* state machines and *variable-specific* state machines. The former detects program-wide properties, whereas the latter detects object-specific properties.

Checkers are written by defining states between which the state machine can transition. When the current program point matches a pattern described in the current state of the checker, the state machine transitions to the next state and/or performs an action (usually printing a warning). When there is no match with

the current program point, the state machine doesn't transition and the analysis continues with the next program point. Example 2.6 clarifies the approach. The example shows an global extension which checks for the double enabling of disabling of interrupts (`sti()` and `cli()` respectively). When enabling interrupts, the state machine transitions to the 'enabled' state. when then enabling interrupts again, an error showing "Double sti" is printed. The same happens for disabling interrupts: When a user attempts to disable interrupts twice, or when the end of a path is reached when interrupts are still disabled (and thus the state machine is still in the 'disabled' state), an error will be printed.

```
.module macros.m

sm cli_sti {
  enabled:
     { cli(); } ==> disabled
   | { sti(); } ==> stop,
        { err("Double sti"); }
    ;
  disabled:
     { sti(); } ==> enabled
   | { cli(); } ==> stop,
        { err("Double cli"); }
   | $end_of_path$ ==>
        { err("Did not reverse"); }
    ;
}
```

Figure 2.6: A simple double enabling/disabling checker

The Metal language is a good example of a clearly readable query language. Pattern matching languages are often most readable, and the approach used in Metal is both very expressive and flexible. Describing the states in the order one wishes to detect them is in our opinion the sweet spot between powerful, expressive languages and flexible, readable languages.

JunGL

In contrast with other languages presented in this section, JunGL[**JunGL**] is a scripting language to perform refactorings, based on pattern matching. The interesting part of their approach is how they specify which parts of the code they wish to refactor. In order to refactor some source code, the program first has to be represented in some way which is easy to access programmatically. They do this by parsing the code into a uniform graph, initially containing only the information of the parsed AST. Lazy edges containing control flow information can also be added to the graph, but only when the refactoring needs this information. JunGL

makes use of *path queries* to express which patterns they wish to match. We believe that these queries are both very readable and very expressive. For example, figure 2.7 shows the path query to find the path from a variable occurence `var` to its declaration as a method parameter:

```
  [var]
  parent+
  [?m:Kind("MethodDecl")]
  child
  [?dec:Kind("ParamDecl")]
&
  ?dec.name == var.name
```

Figure 2.7: A typical JunGL path query

The above style of query denotation offers the exact amount of flexibility, readability and expressiveness that we need to let developers express user-specified queries over a graph. An additional advantage of this type of queries is that no boilerplate code is needed for writing queries, giving an 'out-of-the-box' feel to this type of languages.

2.4 Conclusion

In this chapter we discussed what static analysis is and why it is important for detecting characteristics and vulnerabilities in source code. We have also seen some approaches that enable users to write clean and readable queries that define the patterns to be detected. We can conclude that the state of the art in static analysis has reached a point where tools that support the detection of generic patterns often aren't expressive and flexible enough for application-specific use. A solution for this problem is a tool and a language which allow users to specify their own, custom queries for an application. In this way, no database of pre-encoded vulnerability detection queries is needed as the queries that are specified will mostly be too application-specific to generalize. We believe that expressing vulnerability queries (in the rest of this dissertation referred to as *security policies*) is most readable and expressive using path expressions, as they can nearly be read as a regular sentence consisting of consecutive pieces of code/states to be detected.

Chapter 3

Overview of the approach

3.1 Architecture

A program can be represented in several ways. There is extensive reading material on how logical programming can be used to represent and analyse programs[33][26]. However, other approaches exist that lean more closely towards the implementation of our system. As discussed in section **??**, static analysis can be a means of representing implicit and explicit information about a piece of source code. For our approach, we needed a representation containing enough information to look up non-trivial properties about how information and data flows in the program. Abstract interpretation of a program produces an abstract state graph that meets these requirements. The graph contains information about control- and data flow, providing a rich source of information that can be extracted through some query language and a querying mechanism.

Querying programs depends greatly on the way a program is represented and how queries are transformed into query-engine-friendly data structures. One way would be to resolve queries using existing techniques such as [39]. This technique matches queries expressed in Datalog against a database of rules representing the relations of an entire program. Since our approach represents programs as flow graphs, an alternative method to resolve queries needs to be applied. A suitable algorithm to solving queries is presented in [27], which enables us to query flow graphs directly. The internals of this algorithm will be discussed in greater detail in section **??**.

It is important that exploring and accessing information of a flow graph happens in an easy and user-friendly way. We believe regular path expressions to be the most legible way to write clean and understandable queries. With the JS-QL language, we offer an internal domain-specific language specialized in expressing queries corresponding to sequences of states in the flow graph.

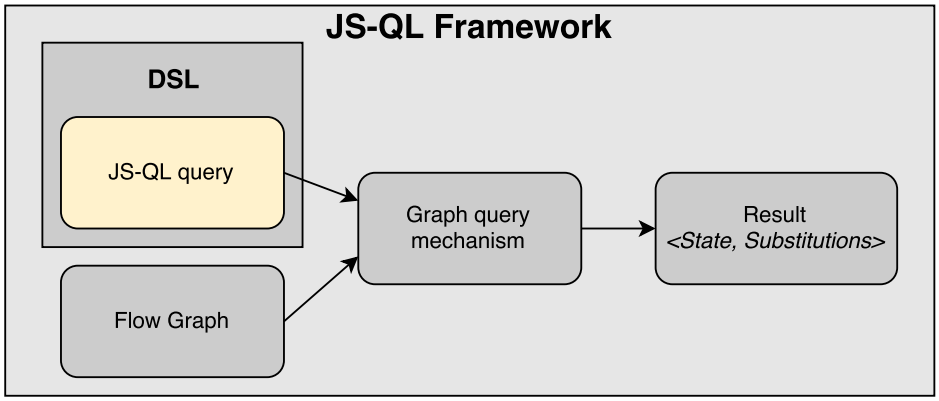

Figure 3.1: JS-QL framework architecture

The actual architecture of the JS-QL framework is depicted in figure 3.1. The query engine takes as input (i) a flow graph and (ii) a query, written in the JS-QL language. The output will consist of tuples `<State, Substitutions>` for all paths on which a match for the query was found.

3.2 Flow graphs for JavaScript programs

The need for detailed control- and data flow information in our program representation graph limits the types of graphs that can be used for our framework. Program dependence graphs[11] for example can be very useful to track the flow of information between certain points in a program but often lack more general information about program states, making them less qualified to use as our main program representation. In contrast, the JIPDA[30] abstract state graph, generated by statically analyzing source code through *abstract interpretation*, contains all the information needed to precisely express patterns to be detected in a program. This section takes an in-depth look at the JIPDA abstract state graph and the information it holds in its states. Figure 3.2 shows part of a typical graph produced by JIPDA for a program containing a check for whether a number is equal to zero or not.

As can be observed, the graph depicts all possible paths a program can traverse. Since the analysis in 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 state `a` occurs first before state `b` on the path. This makes reasoning about patterns in a program much easier, since no false positives will occur with regards to the order of execution of states. The graph produced by the JIPDA analysis is also a flow graph, and more precisely maintains information about two types of flows:

1. *data flow*: Information about what values an expression may evaluate to.

2. *Control flow*: Information about which functions can be applied at a call site.

We need these kinds of information to be able to make correct assumptions at certain states in a program. Consider the expression `f(x)` for example. Function `f` will be the function that is invoked. The value of `f` however may depend on other operations that occur before this function call, such as another function call. Therefore it is important to know which function(s) `f` may refer to, illustrating the need of control and data flow.

States of an abstract state graph

JIPDA internally uses Esprima[18] to parse JavaScript code and set up an abstract syntax tree (AST). This AST is the starting point for the analysis that JIPDA performs, hence information about the nodes from the AST is also contained in certain states in the resulting graph. The small-step semantics of a program are defined by an abstract machine that transitions between different states. The abstract machine is in eval-continuation style, indicating that a state is either an evaluation state or a continuation state. These states correspond to the states that can be seen in the abstract state graph. This graph is an alternation of four different types of states. These states are marked in red and are so-called *evaluation states*. Other states are *continuation states* (green), *return states* (blue) and *result states* (yellow). The states the machine can be in are described below:

1. *Evaluation state*: Represents the evaluation of an expression of the program in the binding environment β.

2. *Continuation state*: A state which indicates that the machine is ready to continue evaluation with the value it just calculated.

3. *Return state*: This is a special kind of continuation state, as it indicates the return of a function application. When the machine is in this state it is ready to continue evaluation with the value calculated for the return of the function application.

4. *Result state*: The final state of the graph, indicating the final computed value(s) of the program. This is also a special kind of continuation state. The machine and graph can have more than one result state, depending on the program's nature.

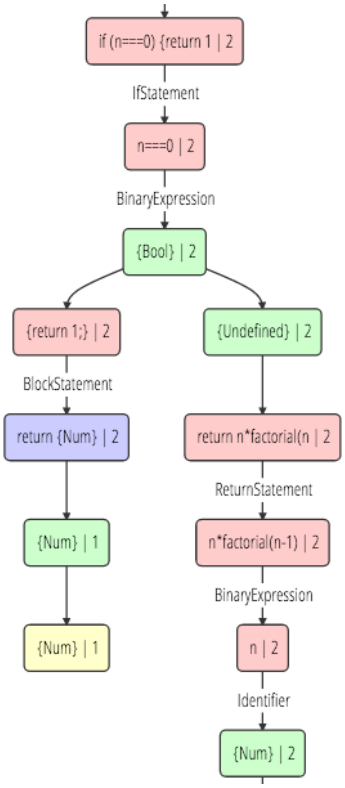

Figure 3.2: Example JIPDA abstract state graph

These states all contain valuable information about the point in the program they represent. The next part of this section discusses the different attributes that can be found in the states of the abstract state graph.

Node

As said earlier, evaluation states contain information about the expression or statement they represent in the program. This information is stored in the form of an AST node, as obtained by the Esprima parser. Detailed information about the current expression or statement can be found in the properties of these nodes. Our approach makes extensive use of this information to find a match for a specified pattern along the graph. Note that node information is exclusively available in evaluation states. If we parse the following program

```
function answerToTheUniverse(arg){
  return 42;
}
```

we obtain its corresponding JSON representation, listed in 3.1.

```
{
    "type": "Program",
    "body": [
        {
            "type": "FunctionDeclaration",
            "id": {
                "type": "Identifier",
                "name": "answerToTheUniverse"
            },
            "params": [
                {
                    "type": "Identifier",
                    "name": "arg"
                }
            ],
            "defaults": [],
            "body": {
                "type": "BlockStatement",
                "body": [
                    {
                        "type": "ReturnStatement",
                        "argument": {
                            "type": "Literal",
                            "value": 42,
                            "raw": "42"
                        }
                    }
                ]
            },
            "generator": false,
            "expression": false
        }
    ]
}
```

Listing 3.1: Parsed JavaScript program AST

The parsed source code is a list of nodes contained in the body property of the ”program” AST node. This is in fact the root node of the AST. Each node has its own *type* that distinguishes different kinds of expressions and statements. The example code in 3.1 shows that the parsed code is a ”FunctionDeclaration” with its own id, parameters, defaults and body attributes. We observe that the attributes in turn can again be (a list of) nodes.

Binding environment and store

In JIPDA, variables point to addresses. The mapping of a variable to an address is called a *binding*. These bindings reside in a *binding environment* β. Each binding maps to a value through the *store* σ. The store acts as a heap where bindings represent addresses on that heap. Being able to capture bindings, addresses and values in metavariables enables us to express and inspect data flow properties of programs. Variables are mapped to values in two stages. The first step for looking up a variable ν is to locate its binding in β. Next, the value of the variable can be looked up in the store by composing these two functions. The value of ν is given by $\sigma(\beta(\nu))$. This way of mapping variables to values allows us to reason about individual bindings, which is necessary because during interpretation multiple bindings to the same variable can exist simultaneously. Listing 3.2 gives an example of how a variable gets a binding and is later looked up.

```
1 function f(){
2   //β contains a binding x → Addr^
3   var x = 3;
4
5   //σ has an entry Addr^ → Val^
6   //and the (set of) corresponding value(s) for x is returned.
7   return x;
8 }
9 var value = f();
```

Listing 3.2: Example of the binding environment and store workings

Value

The lookup of a variable through a binding in the store results in the (set of) value(s) for that variable. This information is available in all states but evaluation states. Values can either be addresses or undefined. For continuation states, the value will represent the looked up or calculated values of an expression. A return state's value is the set of possible values that will be returned. Result states contain the final values of a program.

Stack

The stack is a local continuation delimited by a meta-continuation. The *local continuation* is a (possibly empty) list of frames which acts as a stack (of frames), with normal push and pop functionalities. A *meta-continuation* is either empty or a stack address pointing to the underlying stacks in the stack store. These

stack addresses are generated at call sites and thus represent the application contexts. Useful information such as the call stack can be obtained by tracing out all reachable stack addresses in the stack store, starting from the context that is directly contained in the current state. The traversal of the stack terminates when we encounter an empty meta-continuation, also called the *root context*. A program starts and terminates evaluation in this root context, provided that evaluation happens without errors. The root context corresponds to the top-level part of a program, the global namespace in JavaScript.

Although our framework doesn't provide stack traversal functionalities, basic properties of the stack (local continuation and meta-continuation) can be used and inspected to detect different kinds of states. For a function application, states corresponding with the start and end of the application will have the same local and meta-continuation. With this information, we can for example check for each path if there is a function application followed by a specific state *before* the end of that function application. A concrete example of such a state is a recursive function call.

3.3 External DSLs for querying graphs

For almost any branch of science and engineering we can distinguish between two types of approaches. One type of approach is the *generic* approach, which offers solutions to a wide range of problems within a certain domain. However, these solutions are often suboptimal. When we reduce the set of problems we want to solve, an often better approach to solving these problems would be the *specific* approach. In software engineering terms these two approaches translate to two types of languages: General purpose languages (GPLs) and domain-specific languages (DSLs) respectively. *Domain-specific language* is no new concept. Many programming languages that are now considered general purpose language started out as domain-specific languages. Cobol, Fortran and Lisp for example all came into existence as dedicated languages for solving problems in a certain area[9], but gradually evolved into the full fledged languages they are today. The rest of this section is devoted to the comparison of GPLs and DSLs, in which we advocate that DSLs are the best approach for the instantiation of the JS-QL framework. We further give an overview of related work about DSLs for querying graphs.

3.3.1 Domain-specific language vs. general purpose language

Before we start this comparison, we give a formal definition of domain-specific languages and general purpose languages:

Definition 6. *A domain-specific language (DSL) is a programming language of executable specification language that offers, through approproate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.*

Definition 7. *A general purpose language (GPL) is a programming language that is broadly applicable across application domains, and lacks specialized features for a particular domain.*

The key focus for DSLs are its focussed expressive power. The expressiveness of DSLs comes from the fact that they were created to solve a small set of problems. They offer a high-level set of mechanisms for the programmer to express his ideas for a particular application domain. A DSLs aim is to have the language focus specifically on those aspects and concepts that are relevant to a particular problem domain, hiding all boilerplate code that comes along with GPLs. Designers of general purpose programming languages also try to help the programmers express their ideas concisely and clear, but even with the most elegant programming language difficulties arise when programs get bigger and more complex. To this extent, extra features were developed for GPLs to further abstract code and reduce complexity. Amongst these features are functions, subroutines, packages, objects ... Even though these features are useful for general applications, the languages that implement them often have a set of operational baggage associated with them which makes a program unneccesarily complex to develop[16].

In contrast to the generic approach, the domain-specific approach to language design makes it possible to allow low-level system requirements to guide the design of the required high-level language features one wishes to incorporate into his language, instead of being required to use existing general-purpose designs. We therefore believe that a domain-specific language is the best pick for our query language. Benefits of domain-specific languages include:

- DSLs are application-specific. This allows users to express their ideas at the level of abstraction of the problem domain.
- DSL programs are concise, self-documenting and highly reusable[3]
- Increased productivity: Once the language design and implementation have finished, work becomes much more efficient as you don't have to write the boilerplate code of the GPL manually. In this way you can replace a lot of GPL code with a few lines of DSL code.
- Domain expert involvement: DSLs whose domain, abstractions and notations are closely aligned with how domain experts reason and express themselves, allow for a fluent integration between developers and domain

experts. Domain experts can read, and possibly even write code in the language as they are not directly confronted with any implementation details.

- Programs are often expressed at a level of abstraction that is meaningful for the domain. This brings along that these programs contain domain knowledge and that they can be reused with few to no modifications.

- Improved code quality: Fewer bugs, better architectural conformance, increased maintainability. This is the result of the partially removing the programmers freedom and the avoidance of code duplication by providing DSL constructs.

Some counterarguments for using a DSL are:

- The cost of designing, implementing and maintaining a DSL

- The cost of educating DSL users

- A DSL has limited applicability

- The difficulty of finding the correct scope for a DSL

We argue that the costs for setting up a DSL do not weigh up against the benefits of a DSL. The high reusability alone makes up for the one-time investment of designing and implementing the language. When developing a language for a certain domain, naturally its applicability will be limited to that domain only, as this is the purpose of a domain-specific language. Finding the correct scope for a DSL might be cumbersome, but there is a great amount of literature about specifying the domain of a problem[35] and the domain for DSLs[25][15].

3.3.2 External DSLs

Many DSLs come along with a compiler which translates DSL programs into applications. These kinds of DSLs are called *external* DSLs. the compiler is also called an application generator[5], whereas the DSL is the application-specific language. The main advantage of external DSLs is that the implementation of the compiler can completely be tailored to the DSL. The DSL in turn is restricted in no way with regards to notation, primitives and the like because its syntax is independent of the underlying host language (since there is none). The remainder of this section discusses existing work about external DSLs used for graph traversal and graph querying.

StruQL

StruQL is the query language behind the Strudel system[10]. The language is built to support the retrieval and construction of data for web sites. This data is represented as *data graphs* and originates from external sources, the integrated view and the web site itself. These data graphs depict web sites as nodes, representing web pages or atomic values, interconnected with directed, labelled edges. These edges then represent the links or attribute values that connect two nodes. The language enables users to create and query data graphs, but the real power of StruQL lies in their ability to express regular path expressions. This allows for very flexible queries describing the paths about which information needs to be accessed in great detail. It also allows to compute the transitive closure of an *arbitrary 2n-ary* relation, meaning that it can compute all reachable nodes from a certain node for any input graph. Buneman et al[4] have formally proven that this is not a trivial computation.

GraphQL

GraphQL[17] is a query language which allows to query graph databases. The language uses a graph pattern as a basic operational unit. These graph patterns consist of a graph structure and a predicate on attributes of the graph. They introduced the notion of formal languages for graphs. This is useful for composing and manipulating graph structures and is used as a basis of the graph query language. The core of the language is a graph algebra in which the selection operator is generalized to graph pattern matching and a composition operator is introduced for rewriting matched graphs. In terms of expressive power, the language is contained in Datalog. This means that every query in GraphQL can be converted to a Datalog query. The language allows users to express concatenation, disjunction and recursion, allowing users to write dynamic queries. They address the NP-completeness of subgraph isomorphism by using neighborhood subgraphs and profiles, joint reduction of the search space, and optimization of the search order.

ASTLOG

ASTLOG[7] is a query language for syntax-level C/C++ program analysis and is well suited to construct anchored patterns to match tree-like structures. The language is built as a Prolog variant syntax-wise, but instead of transforming an entire program into a database of Prolog rules, it is able to match *objects* to queries directly. These objects are being made available through a C/C++ compiler frontend which provides an interface to the syntactic/semantic data structures build during the parse of a program. Among the available objects are the AST nodes of a program. These nodes can then be examined and queried by user-defined predicates

in a similar fashion as one would do in Prolog. This allows for application-specific composable predicates.

Lorel

The Lorel language[1] was designed to query semistructured data. This kind of data can be seen as a graph with complex values at internal nodes, labeled edges and atomic leaves. The language's syntax resembles that of OQL (*Object Query Language*), but has two two additional features: (i) A coercion mechanism for value/object comparisons and (ii) powerful path expressions. Coercion is needed for semistructured data, as two objects may represent the same data in different ways. Lorel introduces *general path expressions*, a way to define label completion and regular expressions in paths. Regular expressions are supported through `.`,`+`,`?`,`*`,`()` and `|`, label completion is done as in SQL, namely with the `%` symbol.

3.3.3 Internal DSLs

In contrast to external DSLs, *internal* (or *embedded*[20]) DSLs don't require a custom compiler. These languages inherit the infrastructure of some other (general purpose) language, and tailor it towards the domain of interest. In this way the language can be interpreted by its host language, saving the developer a lot of work. 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 in addition to domain-specific expressive power of the DSL. This expressive power along with not having to build a fully fledged compiler for our DSL are the main reasons we prefer the internal DSL approach above the external DSL one. The rest of this section describes three internal DSLs, two for graph traversal and one that illustrates the flexibility and expressiveness of embedded DSLs. The terms internal DSL and embedded DSL both have the same meaning in the rest of this dissertation and both refer to the type of DSL that is embedded in a host language.

Gremlin

[34] presents the Gremlin graph traversal machine. The machine traverses graphs according to a user-specified traversal, making use of so-called traversers. These traversers can be seen as 'workers' who walk through the graph, keeping a bag of information on their back about the path they have already taken and the current graph node they are in. The machine is developed in such a way that it can be implemented as an embedded DSL in any host language, provided that the host

language supports *function composition* and *functions functions as first-class entities*. The Gremlin language has an *instruction set* of about 30 steps and each query is a sequence of these steps (i.e. a path). Querying graphs through paths is a well-known approach, but the Gremlin machine also supports nested paths for which each nested path is a graph traversal on it's own. Queries are transformed into traversals, so each traversal can be made application-specific. They present 9 different traversals, including a recursive and a domain-specific one.

Dagoba

Dagoba is an in-memory graph database system written in JavaScript. The chapter about Dagoba in the *500 Lines or Less*[1] book provides an elaborate explanation on how to create a flexible, easily extensible internal DSL. The language is built as a fluent API, and explains which mechanics (such as lazy evaluation) go hand in hand with this kind of language representation. They also describe how they interpret the language and define some optimizations of the system, mainly through query transformators.

A little language for surveys

A Little Language for Surveys [8] explores the use of the Ruby programming language to implement an internal domain-specific language. It checks how well the flexible and dynamic nature of the language accomodates for the implementation of a DSL for specifying and executing surveys. Two key features of the Ruby programming languages are exploited because they especially support defining internal DSLs: The flexibility of the syntax and the support for blocks. Function calls for example are easily readable, since the braces surrounding the arguments can be omitted and the arguments list can consist of a variable number of arguments (The latter is also supported in JavaScript[21]). They make extensive use of the fact that entire blocks can be attached to method calls. These blocks are passed unevaluated to the called method, enabling *deferred evaluation*. Next to these features, the meta-programming facilities of the Ruby makes it possible for them to read a DSL program and execute it in the contexts specified by that program. A two-pass architecture is used, splitting up the parsing and interpretation of the program. This is common practice for internal DSLs.

[1] http://aosabook.org/en/500L

3.4 Design of an internal DSL for querying flow graphs

Crafting a compiler for our languages falls outside the scope of this dissertation and we believe that the overhead for building an external DSL does not weigh up against the benefits of an internal DSL (as discussed in 3.3.1). In this section we discuss the design process of our internal domain-specific language named JS-QL.

3.4.1 Internal DSL design constraints

This section describe some factors that influenced the design of our query language. A first constraint for the language is that it was designed as an embedded DSL. This has as a consequence that we have to use the constructs and syntax the host language offers, JavaScript in our case. The JS-QL languages makes extensive use of JavaScript `Objects` that function as dictionaries. A limitation for us was that these dictionaries only accept strings as keys. Function calls for example can't serve as keys in JavaScript objects as their value can't always be calculated at compile time. This was a serious drawback for us and it even lead to an inconsistency in our language, as will be discussed in chapter 4. Our language has to be easily extensible as users need to be able to specify their own predicates and policies. To this extent, JavaScript surfaced as an ideal language. The dynamic typing and optional function arguments made creating flexible predicates and policies a lot easier.

Flow graphs need to be queried, so the language has to fit these needs in the form of appropriate predicates. The design of our language depended on the information that is contained in each state of the graph. The JIPDA graph states contain several fields that in turn can recursively contain other fields. The nature of the algorithm we use together with the structure of these states asked for a close mapping of states to query predicates. As JavaScript objects are great for storing nested information, we chose to use them as a mapping for states. From now on these objects will be referred to as *dictionaries*. Fields in a dictionary now have a one-to-one relationship with fields in a state, making the matching process for the algorithm less of a burden. Another constraint imposed by the flow graph was that sequences of states had to be expressed in a precise yet legible fashion. This had as a result that the language was set up as a fluent interface, enabling the user to specify a number of states separated with a simple dot. We can thus say that the type of graph helped shape the JS-QL language.

A final constraint was the need of an environment where queries can be expressed and evaluated against the flow graph. It would be tiresome to specify queries in one place and the input program elsewhere. This would also imply that every time a change to the program or the query has to be made, at least one separate file has to be modified. This is clearly not an optimal solution. As there is no

read **e**val **p**rint **l**oop (REPL) available for JavaScript in the browser, we opted to extend the existing environment of the JIPDA analysis with support for (i) writing queries and (ii) checking these queries against the flow graph.

3.4.2 DSL implementation techniques and patterns

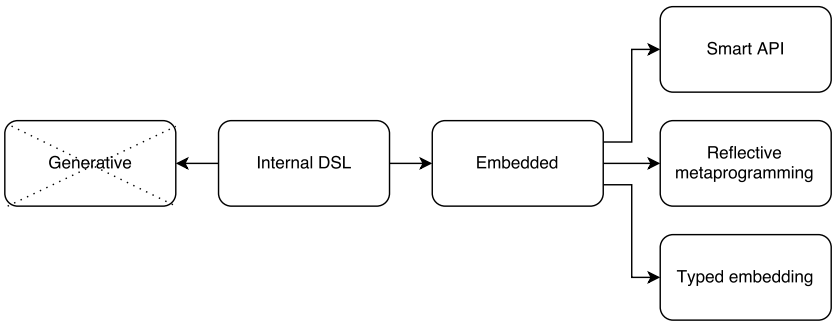

Figure 3.3: Internal DSL implementation techniques

Figure 3.3 shows the different kinds of implementation techniques for internal DSLs, as specified by [13]. Our DSL doesn't generate any code, so we won't discuss generative internal DSLs. Embedded internal DSLs on the other hand can be implemented in three ways:

1. *Smart API*: Readability is key for DSLs. Implementing a fluent interface is a way to improve readability and make Smart APIs. To this extend method chaining is a popular technique: It can be implemented by making the output of one method flow naturally as the input of another. Benefits are that a series of invocations in the DSL feel more natural and that it expresses the series of actions you want to perform or detect in the problem domain. Boilerplate code is not needed for this type of internal DSL, hence the name *Smart* API.

2. *Reflective metaprogramming*: The decorator pattern[2] is a well known pattern for extending individual objects. The ability to extend individual objects in a DSL greatly adds flexibility to the language. For some dynamically typed programming languages it is possible to define dynamic builders. These builders have a similar objective as the decorator pattern: construct an object incrementally within a DSL. This technique allows the language designer to smooth out the implementation aspect of the DSL by letting the user construct methods dynamically through the meta-object protocom of the language runtime.

[2]https://en.wikipedia.org/wiki/Decorator_pattern

3. *Type embedding*: Types make for more concise and robust code. This is the philosophy that is behind the type embedding technique. Internal DSLs with a statically typed host language often use this pattern to make the language more expressive using the power of a type system. Each abstraction in the domain model should be made typed (and generic). This allows for fewer code duplication and a large part of the work will be done by the compiler of the host language. By making abstractions typed, correctness of programs written in the DSL is guaranteed: If the code compiles, it will most likely be correct.

For our internal language we can already rule out the type embedding technique as JavaScript is a dynamically typed language. The problem domain of our approach doesn't need extra functionality to be added to individual objects in our language. These individual objects in our case are the states of the flow graph. States already are self-contained (they contain all necessary information) and don't require additional individual functionality.

This leaves us with the smart API approach, which is ideal for our domain. By chaining methods in our DSL we can specify which states we want to encounter along the graph in a clearly specified order. Combining a smart API with several carefully chosen DSL design patterns[12] result in the concise and easily readable language that JS-QL is today. The remainder of this section elaborates on the chosen design patterns.

Method chaining

Method chaining is the bread and butter of our DSL and at the same time also the *only* way of expressing queries. This approach offers a fluent interface to the user in which it is hard to make any coding mistakes. The ability to express a state one wishes to encounter as a chained method to states he discovered earlier in the graph allows to build very readable queries. We illustrate this with an example: `G.skipZeroOrMore().functionCall()`. It is immediately obvious that this code is very intuitive: `G` is the entry point of our language, which indicates the start of a query. We then search for a path in the graph that contains a function call somewhere down that path. This is expressed by `skip`ping some, possibly none, states until a function call state is encountered.

Literal map

Specifying that one wishes to find states is often quite general. To this extent we need some sort of mechanism to express which types of states we want to match and what information we want to capture in (meta)variables. A literal map

provides just this functionality by letting the user specify detailed key-value pairs. An example for could be: `.functionCall(name: 'f', arguments: '?args')`. The literal map, enclosed in curly braces, indicates that only fuction calls with name `f` need to be matched and that the arguments of the matched state need to be captured in metavariable `?args`.

Object scoping

The example for the method chaining design pattern is also applicable for the object scoping pattern. A single entry point `G` is created for queries, limiting the impact of the query that object. This pattern remedies two JavaScript flaws: Global namespace pollution and malicious code injection. This malicious code will only harm the `G` object, which is contained in some sort of sandbox.

Deferred evaluation

Some queries contain definitions for extra properties and filters. The information for these filters and properties is often not available at compile-time of our language. A mechanism is needed to delay the evaluation of those filters and properties until the matching process in the backend has collected enough information. Our framework handles this by creating thunks for filters and properties and unwrapping these thunks when the matching engine needs to evaluate them. By the time the evaluation happens, all variables should be bound in these thunks in previous matching steps. Consider a metavariable `?val` which captures the value of an assignment. If we only want to match assignment states with a value greater than 2, we have to create a thunk for the filter function > with arguments `?val` and `2`. We can't disregard states with `?val` greater than 2 immediately, as we don't know which value will be bound to `?val` at compile time.

Delimiter directed translation

Inherent to our DSL is delimiter directed translation. As method chaining is used as a means to set up our fluent interface, all methods are separated by a dot (the delimiter). Each method separated by this dot gets separately translated internally into a representation that is easier for our backend to process.

Newline separators

Finally, the newline separators design pattern is incorporated in our DSL. This design pattern allows users to enter newlines between parts of their code. This greatly improves the readability and can split a program up in logical parts. Our DSL supports newlines in queries. This can be very useful to separate different

states or sequences of states. Consider an example in which we only want to find assignments to variables `a` and `b`. Code can then be divided in two distinct parts, as in example 3.3.

```
1 .assign({leftName: 'a'}) //first logical part
2 .or()
3 .assign({leftName: 'b'}) //second logical part
```

Listing 3.3: Newline separators

Chapter 4

JS-QL: An internal DSL approach for querying flow graphs

In this chapter we present the JS-QL framework. The framework offers the possibility for developers to write application-specific queries to check for certain program properties. More specific, it tries to offer a solution to developers who want to test their applications for vulnerabilities by writing and enforcing security policies for them. The framework consists of three main parts:

1. *The JS-QL query language*: Short for **J**ava**S**cript **Q**uery **L**anguage. This is the domain-specific language in which users can express all kinds of security policies. An overview of the language is given in section 4.1.

2. *The matching engine*: This is the core of the framework. It matches the user-defined query against states of the JIPDA abstract state graph, capturing and unifying all the relevant information. This engine can be configured to behave differently for certain queries, as will be discussed in section 4.1.2.

3. *The graphical user interface*: The user interface provides the infrastructure for the developer to interact with the framework. It contains a section where users can specify the input program and security policy, a graphical component representing the abstract state graph corresponding to the input program and a visual and textual representation of the query results. The textual representation allows developers to inspect the captured variables, a handy feature when these variables are compound data structures.

4.1 The JS-QL query language

The abstract state graph obtained from the JIPDA analysis is a perfect starting point to inspect a program for certain characteristics and security vulnerabilities.

In chapter 3 we motivated our choise to design an internal DSL to query for specific (sequences of) states in this graph, with the aim to discover program patterns that might lead to violations of user-defined security policies. The language constructs are built to make it easy for the user to specify which kind of pattern he wishes to detect. The rest of this section presents all facets of the JS-QL language: Section 4.1.1 discusses all constructs of the language and gives an in-depth explanation on how to use them in a correct way. As different security policies require different traversals of the state graph, more than one type of query is needed. We discuss the difference between several query types in section 4.1.2. In order to have an effective query language, we must allow the user to create compound queries out of the available language constructs. Section 4.1.3 shows how this can be done within the framework.

4.1.1 JS-QL syntax

In this section we will discuss the available constructs and syntax of the JS-QL language. The examples in this section will be simplistic and will demonstrate how each construct works. They will therefore not always represent an actual security policy, but will rather serve as a guideline for using these constructs. We chose to use *path expressions* to express queries in JS-QL, and more precisely *regular path expressions*. This adds the flexibility of regular expressions to the language, as the most relevant features of regular expressions are incorporated in JS-QL.

The entry point

As our language is an internal DSL, meaning that it is embedded in a host language, the host language has to provide an entry point from where we can start using the JS-QL language. We chose to map this point to the `G` object, which is short for **G**raph. This implies that all query patterns in JS-QL will start from this object. A simple example is seen in 4.1, where the first state of the graph is matched.

```
1 //Match the first state of the graph
2 G.state()
```

Listing 4.1: Matching the first state starting from entry point `G`

State

the `state` construct is the single most basic element of the language. It matches any state in the graph, but doesn't provide much information on its own. Never-

theless is it the most important building block of the language, as it can be used to construct higher-level queries and predicates. States can be made more precise and expressive by parametrizing them with *state constraints*, but in order to know what we can query for, we will first give a short overview of what information is available in which states. To get a more detailed explanation on what each piece of information represents, we refer to the section about flow graphs (3.2). Table 4.1 indicates what information is available in which type of state. The table also shows which keyword is used to represent the information is that is in embodied in the states.

Legend	
Evaluation state	E
Continuation state	K
Return state	R_t
Result state	R_s
All states	A

State property	Available in states	Keyword
Node	E	node
Meta continuation	A	kont
Local continuation	A	lkont
Binding environment	E	benv
Store	A	store
Value	$K\ R_t\ R_s$	value
Identifier	A	_id
Successors	A	_successors

Table 4.1: Information in the states of the abstract state graph

As readability is key in a query language, JS-QL provides four extra constructs that are semantically almost equivalent to the regular `state` construct. `evalState`, `kontState`, `returnState` and `resultState` are included in the language, with the purpose of enhancing readability, but also to match only those specific kinds of states. Note that the framework also supports the usage of the identifier and successors of a state, but it is very uncommon to use them, as they are semantically irrelevant to queries. The identifier of a state would only be relevant when a query is matched in that exact state. When this is the case, the state gets marked in the visual graph representation and all query information is then contained in that state. To retrieve the information, one simply has to click the state to inspect all variable values. Successors also contain few additional information, as all direct successors of a state are already made explicit in the state

graph. A use case for the use of the successors keyword could be to find all states after which some branching occurs. This could then be specified as a JS-QL query which captures the successors array in a variable `?succArr`, and then filters the results to only contain the states with the length of `?succArr` strictly greater than 1.

To understand how we can parametrize states, we first have to know how to define variables in JS-QL. Variables in JS-QL are strings, starting with a `?`. The fact that we use strings comes from the embedded nature of our language: If we were to specify variables as literals, the host language would complaint that it doesn't recognize the literal. Listing 4.2 illustrates this.

```
1 //Capture the 'type' property of the node in variable '?nType'
2 G.state({ node : { type: '?nType' }})
3 //Exception: ?nType is not recognized by the host language
4 G.state({ node : { type: ?nType }})
```

Listing 4.2: Defining variables in JS-QL

As the example might already indicate, JS-QL deconstructs state properties as nested key-value pairs. In this way, each part of information can be captured in a variable. The key indicates the property the user wishes to match whereas the value can be one of three things:

1. A *variable*. When placing a variable as the value in a key-value pair in JS-QL, that variable gets bound to the key's corresponding value in the JIPDA state. The '?nType' variable in the example above gets bound to the value of `type`, which in this case corresponds with the type of the AST `node` for the currently matched state.

2. A *nested map* which further deconstructs the current property. The example above does this by further deconstructing the `node` property of a state (which represents the corresponding AST node) in order to reach the `type` of that node and store it in a variable. It is obvious that this is most used to match specific AST nodes.

3. A *literal*. Literals are mostly used to filter the states to be matched. When applying this to the example above, the '?nType' variable could be replaced by the literal 'ExpressionStatement' for example. Note that the question mark (`?`) is omitted. The resulting query would then only match a state having the `type` of its corresponding AST `node` equal to 'ExpressionStatement'.

States can thus be parametrized by matching the keywords displayed in table 4.1 as keys with values that can be variables, literals or nested maps. It is obvious

that queries matching single-state patterns aren't quite qualified as being security policies. JS-QL therefore allows users to specify sequences of states as a query. When checking the state graph against this query, all states in the query pattern need to be matched one after another. When a state in the query pattern is encountered that doesn't match the current state in the state graph, the matching process is aborted for the current path that is investigated in the state graph. Consider the following query:

```
1 G.state({ node : { type: '?tpe' }})
2  .state({ node : { type: '?tpe' }})
```

Listing 4.3: Unification in JS-QL

We immediately see that the variable `?tpe` occurs twice in the query. This can be done on purpose to achieve *unification*. Unification simply means that two variables with the same name have to contain the same value. After executiong the first line, the first state in the graph is matched (if it has the `node` property) and the variable `?tpe` is bound to the type of the node. The matching engine then proceeds to the next state in both the query and the state graph. If the next state again has the `node` property with the same type as already bound to `tpe`, the unification process has succeeded and the whole query will match. If the node type of the next state isn't equal to the value already bound to `?tpe`, or if that state doesn't have a `node` property, there is no match. The results of a successfully matched query will be the set of all possible *substitutions*, together with the identifier of the state where the last element of the query matched. Figure 4.1 gives a simplistic visual representation of this process for the query listed in listing 4.3. For the graph on the left-hand side, the query is fully matched by successfully unifying the the type of the first and second state. In contrast, the graph on the right-hand side will not produce a match as {`?tpe` : 'ExpressionStatement'} can't'be unified with {`?tpe` : 'AssignExpression'}. More complex examples of unification can be found later in this chapter.

All queries presented until now match the state graph from the beginning of the graph only. This behaviour is often undesirable as a developer usually wants to detect a pattern *somewhere* in his code, not necessarily at the beginning. To resolve this, JS-QL combines techniques from regular expressions with a special built-in construct, the *wildcard*.

Wildcard

Wildcards are usually known as 'things of which the value can be anything', and this is no exception in JS-QL. A `wildcard` serves the sole purpose of matching any state it gets compared with. In other words, an equally correct name for this construct could have been `skip`, as it skips a state in both the query (the

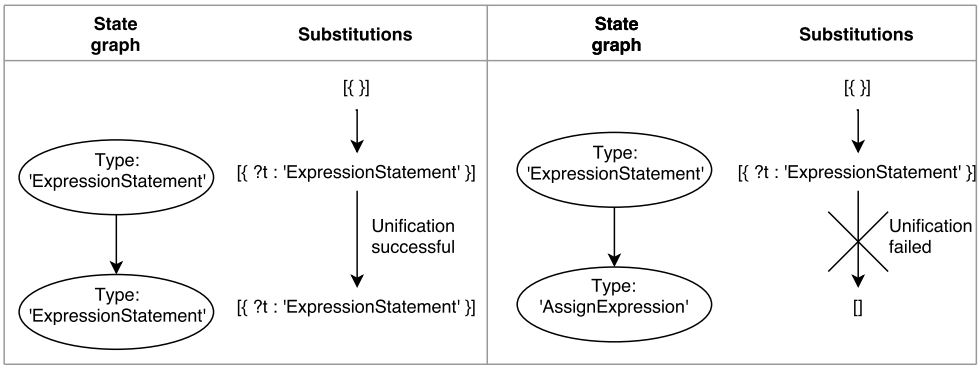

Figure 4.1: Visual representation of the unification process

`wildcard` state itself) and the abstract state graph (the state the wildcard gets matched with). When talking about states in a query pattern, both `state` as `wildcard` match this definition. `wildcard`s act just like regular `state`s in a query, meaning that they only match *1* state in the state graph. It is very unlikely that a developer knows exactly after how many states a violation would occur, so simply enumerating wildcards followed by the state to be matched would typically be a very tiresome effort. We therefore need to be able to specify that we wish to skip *zero or more* states before matching the following state in the query pattern. This is where the power of regular expressions comes in handy. Just like regular expressions, JS-QL supports the use of both the Kleene star (`star`) and the Kleene plus (`plus`) operators. In our language, the `star` and `plus` constructs are both placed *after* the state(s) they are applied to. Placing `star` behind a sequence of states indicates that those states can occur *zero or more* times at the current position in the state graph. The semantics of `plus` are very similar, except for the fact that the states have to occur at least once in the order they are specified. Just like with regular expressions, pieces of a pattern can be surrounded by braces. Left and right braces in JS-QL are denoted by `lBrace` and `rBrace` respectively. The default behavior of both the kleene star and kleene plus operators is to apply them to the state that occurs right before it. If any kleene operator has to be applied to multiple states, these states have to be wrapped in braces in JS-QL. Listing 4.4 shows the differences. Lines 1 and 2 are semantically equivalent, as the braces on line 2 only contain 1 state, a `wildcard` in this case. Lines 3 and 4 on the other hand are semantically very different. The query on line 3 matches all but the first states in the state graph, whereas the query on line 4 matches every other state in the graph, starting from the second state. The combinations of `wildcard().star()` and `wildcard().plus()` are so commonly used in JS-QL, that a special construct is created for each of them:

skipZeroOrMore() and skipOneOrMore() respectively.

```
1 G.wildcard().star() // Equal to G.skipZeroOrMore()
2 G.lBrace().wildcard().rBrace().star()
3 G.wildcard().state().plus()
4 G.lBrace().wildcard().state().rBrace().plus()
```

Listing 4.4: Kleene operations differences

Disjunction

Sometimes when writing a query, more than one state on the path is allowed for the query to match. Consider a simple language in which we want to detect all uses of a variable v. Using a variable in this language can only be done by using it in an binary arithmetic expression. To keep the example simple, we disregard all other possible uses of a variable. We also assume in this example that no other variable was assigned the value of v, so that no aliases of v exist in the code. When using a variable in a binary arithmetic expression, the variable can be on either side of the operator. A naive solution to query for all uses of v would be to first launch a query that finds all occurences of v on the left-hand side, followed by a query that detects all occurences on the right-hand side. To alleviate the work of the user, JS-QL offers the disjunction construct or, which allows to specify that 1 state in the state graph can be matched by multiple states in the query pattern. If we then were to match all occurrences of v on the left- and right-hand side of arithmetic expressions, we could write a query as in listing 4.5.

```
1  G.skipZeroOrMore()
2  .lBrace()
3    //Left-hand side with name 'v'
4    .state({node: { type: 'BinaryExpression',
5                    left: {name:'v'}}})
6    .or()
7    //Right-hand side with name 'v'
8    .state({node: { type:  'BinaryExpression',
9                    right: {name:'v'}}})
10 .rBrace()
```

Listing 4.5: The JS-QL disjunction operator

This query first skips zero or more states, starting from the beginning of the graph. It then matches a state (evalState would be equally correct) with a node property of type 'BinaryExpression'. Remember that the node property of evaluation states contains the AST information for the current expression. Because of this, the left and right properties of the BinaryExpression are again nodes that can be further deconstructed. The or construct splits the query in two

different query paths. One path will try to match the pattern specified before the construct, whereas the other path searches for matches for the pattern specified after the `or`. The same rules apply w.r.t. braces as for the `star` and `plus` operators. For the first path, we deconstruct the left property of the node and match its name with the literal `v`. This automatically filters out all states for which this condition doesn't hold. What remains is a match for each state for which the condition holds. The same is done for the second path, with the only difference that the name of the right node now has to be equal to `v`. As the query doesn't store any variables along the path, the result of the query can only be observed in the visual representation of the abstract state graph. In this graph, all matching nodes will be marked with color, indicating a match.

When examining the example above, we notice that very few relevant information is available as a result of the query. A more detailed result should contain the actual node or even the entire state that was matched, so we could inspect it further. To this extent, an additional implicit property was made available for each deconstructable map in the language. This property represents the entire object by which it is encapsulated. We chose to give this property an appropriate name, indicating that it points to the object currently being inspected, namely `this`. Listing `lst:updatedDisjunction` gives an updated version of the relevant code from the previous example. In this version, the `thisNode` variable will be bound to the node of the matched state.

```
1 //...
2 .state({node: { this: '?thisNode',
3                  type: 'BinaryExpression',
4                  left: {name:'v'}}})
5 //...
```

Listing 4.6: Using the `this` keyword

Specifying additional properties

Sometimes it can be useful to capture extra information about already matched variables. Doing so in a separate section, exclusively designed for this purpose, has two advantages. First of all, it enhances the readability of queries. Queries with deeply nested maps can quickly become confusing to read and bothersome to modify. Secondly, it opens up for opportunities to make the language even more expressive. JS-QL has a built-in keyword `properties`, which can be used to obtain more information from already bound variables.

Expressing properties can be done in two ways, as indicated in listing 4.7. As the language we are using is an embedded language, we can take advantage of the host language. JavaScript allows the value of key-value pairs to be a function

call with arguments. We can we can benefit from this by defining a JavaScript function `prop`, to which we can pass which kind of information we wish to obtain and from which variable we want to obtain these properties. The first argument of `prop` is the function that needs to be applied when the matching engine processes the query. The arguments of this function are all other arguments that were passed to `prop`. Lines 5 and 7 of the example below shows how to properly use the `prop` function. We have to defer the evaluation of the function passed as a first argument to `prop` because at compile-time the values of the variables aren't calculated yet (as no matching has happened). This function, in the example below 'memberOf', can be a user-specified function or a built-in function. As for now, JS-QL only has three build-in functions that work on variables, and all three require that variable to be of type *Array*:

1. *length*: A query could contain `prop('length', '?arr')`, where `?arr` is a variable with a value of type `Array`. The function then returns the length of the array bound to the variable.

2. *at*: This function takes 2 additional arguments: A variable containing an array, and an index `i`. The resulting value is the `i`th element of the array.

3. *memberOf*: This function is the most important and also most used one. It takes a variable containing an array `arr` as an argument and expands the current substitution set, so that for each element in `arr` a new substitution set is created with that element appended to it. In the example below, the substitution set before the execution of line 5 would look like
   `[{?decls : [dec1, dec2]}]`,
   and would afterwards look like:
   ```
   [{?decls : [dec1, dec2], ?dec : dec1 },
    {?decls : [dec1, dec2], ?dec : dec2 }]
   ```

Another way to define properties is by simply specifying which attribute of a variable one wishes to capture. Line 6 of the example below shows how the 'name' of the 'left' attribute of `?dec` is bound to `?decName`. Declaring a new property variable is done similarly for both ways of defining properties: The key of the map should contain the variable name to be declared, whereas the value should be the property specification. One might notice that this order of key-value pairs is different from all other notations in JS-QL. This is because the host language doesn't allow function calls to be keys in maps, and thus restricts the syntax of our language in that way.

The example below matches all states of the state graph that declare variables. These declarations are captured in `?decls`. Next, for each declaration, a new substitution set is generated by the 'memberOf' function. Each substitution set

now contains a variable `?dec` bound to an element of the `?decls` array. Finally, the name of the declaration in each substitution set is captured in `?decName`.

```
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       }})
```

Listing 4.7: Specifying additional properties in JS-QL

It is very important to keep in mind that only the variables that are already bound can be used in properties. This implies that the order of the keywords in a query is important for the semantics of the query. If we were to switch the `node` and `properties` keywords of the above query, an error would occur because the `?decls` variable wouldn't be available to use in the properties section.

Filtering states

Filters in JS-QL work very similar to properties, except that they act as guards who filter out states that don't satisfy certain conditions. A filter can be any function, predefined or specified by the user, that returns a boolean value. When returning true the pattern can be matched further, otherwise the matching process aborts and no match for the current path in the state graph is found. A filter is declared through the `cond` JavaScript function (similar to `prop` for filters), and takes a filter function as a first argument. All other arguments are passed as the arguments to the filter function. As no variables have to be stored for filters, the notation is not traditional in such a way that it doesn't use map. Instead, a JavaScript array lists all filters to be satisfied. Consider the following example: JavaScript allows the declaration of multiple variables in one declaration statement, e.g.:
`var a = 1, b = 2;`
Some companies see this as a bad practice, as it is harder to maintain and less error-prone. A query can be written to to detect all violations of this company policy, by storing the length of the declarations in a variable and checking if the length of the variable is larger than 1. Only the states for which this filter is satisfied will be contained in the results of the query. Listing 4.8 illustrates this example.

```
1  G.skipZeroOrMore()
```

```
2  .state({
3        node:{declarations: '?decls'},
4        properties:{
5          '?length' : prop('length', '?decls')
6        },
7        filters:[
8            cond('>', '?length', 1)
9        ]
10     })
```

Listing 4.8: Filtering for multiple declarations

Data flow in JS-QL

Support for data flow opens up to write a whole new class of queries. As JavaScript is a dynamic language, any variable can be assigned to any other variable. This phenomenon is known as *aliasing*, and is very common in the language. As already discussed in chapter 3, JIPDA stores the addresses and values of variables in the `store`. Values stored for primitive types are indistinguishable in the store, because of abstract interpretation:
`var x = 4; var y = 9999;`
Both x and y are integers and will point in the store to a value of {`Num`}. We thus can't track aliasing for primitives. Reference types in JavaScript however are distinguishable in the store as they each point to a set of unique addresses for that reference. Remember that reference types can point to multiple addresses as a consequence of precision loss of abstract interpretation. A feature of the abstract state graph that affects the expressiveness of our language is that it displays states as they are evaluated. This means that for assignments for example, the address of the left-hand side of the assignment often isn't available in the assignment state, as the value (and address) of the right-hand side needs to be evaluated first. This happens *after* the assignment state in the state graph. An example is the simple program: `var x,y; x = {}; y = x;` In this program, `x` gets assigned a fresh object, after which it gets aliased to `y`. `x` and `y` point to the same set of addresses after execution of this part of the program. The relevant graph part is depicted in figure 4.2

The problem with this kind of representation in the graph is that when we try to match the assignment in a query, no address information for `x` is available. After evaluating the fresh object, `x` gets stored in the store. Later on in the program, `y` gets assigned the value of `x`. Now, there is no problem because the address of `x` is already available in the store, so it can be looked up in the query. Note that the data flow detection in JS-QL isn't nearly as powerful as for example a taint analysis. Queryies and policies can be written to mimic a taint-analysis, but this

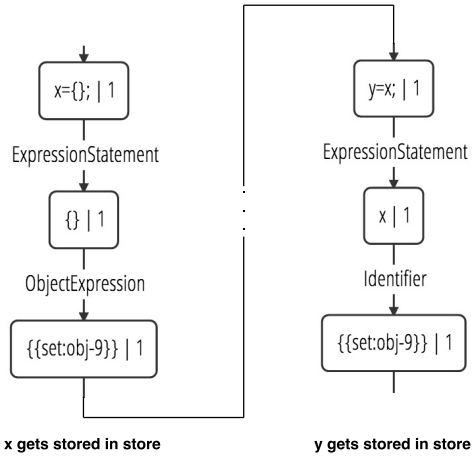

Figure 4.2: Assignment representation in the state graph

requires some work. Later in this chapter we will discuss how custom queries and policies can be defined.

Variables can be looked up in JS-QL using the `lookup` keyword. The value-part of this keyword is again a map with the names of the variables to lookup as keys, and the variable names that need to be bound to the addresses as values. Performing a lookup in JS-QL happens by first looking into the lexical scope. When variables with the same name are defined in both a function and the global scope, the lookup value of that variable will depend on the state that is currently matched. Therefore, we need to provide some sort of mechanism to 'overwrite' this default behavior, in case the address of the global variable is needed when the currently matched state is inside a function application. To this extent, JS-QL recognizes the `_global` string as an indicator to perform a lookup in the global scope. Listing 4.9 shows how variables can be looked up. First, the name of the right-hand side node of the assignment expression gets bound to `?rn`. When the assigment expression gets matched with a state in the global scope, all three address variables `?rnAddr`, `?globrnAddr` and `?xAddr` will have the same value. However, when the assignment on line 7 gets matched, `?globrnAddr` will point to the globally defined `x`, whereas the other two lookups contain the address of the locally defined `x`.

```
1 //Javascript program
2 var x, y;
3 x = {};
4 y = x;
5 var f = function(){
6   var x = 4; //local x gets declared
7   y = x;
8 }
```

```
9  f();
10 //JS-QL query
11 G.skipZeroOrMore()
12 .state({node:{
13           expression:{
14             type:'AssignmentExpression', right: {name:'?rn'}
15         }},
16         lookup:{
17           '?rn'         : '?rnAddr',      //lookup by variablename
18           '_global.?rn': '?globrnAddr', //lookup in global scope
19           'x'           : '?xAddr'        //lookup based on name
20         }})
```

Listing 4.9: Looking up addresses in JS-QL

Negation

Expressing which statements and expressions we want to detect on a path in the graph is made easy by JS-QL. Sometimes however, this doesn't suffice for some queries. In some languages, accessing a file after it was closed results in an error in the application. This might compromise the integrity of the system. A naive approach to writing a policy for this would be to detect all calls to the `access` method that follow, after some wildcard states, the call of a `close` method for a file `f`. While this query would match and return all violations against the policy, false positives would occur. This is because of the wildcard states between the two function calls. One of these calls could be a call to the `open` method, meaning that accessing the file afterwards is permitted. A better approach would be to specify that we *don't* want to encounter a call to `open` for `f` between closing and accessing it. This is exactly what the `not` construct does. When placing this construct right before a state, the query will only be matched if the negated state can't be matched with the current state in the state graph. When placing `not` before a state, and `star` or `plus` right after that state, it can be read as: "Match zero/one or more states that are *not* the negated state". A better query for the example above would then be written as in listing A.1, which can be found in the appendix for brevity reasons. The query first skips several states until it reaches a call to `close` on file `f`. The address of the file to be closed is also stored in a variable to correctly detect only the accessing and opening of that specific file. Next, all states are matched that are not opening `f`. Finally, the actual violation is detected, namely accessing the file after it has been closed and not re-opened. Negation is in the current version of JS-QL subject to some limitations:

1. Variables that are bound in a negated state, are only visible to that state. They will thus not be included in the resulting substitutions.

2. Currently, only one state can be negated. Negating sequences of states wrapped in braces is not yet supported.

4.1.2 Types of queries

The most straightforward way to query for properties is by just specifying a pattern and an input program. The pattern is then checked and each violating path is reported in the results. Sometimes however other types of queries are needed for the detection of specific policy violations. In this section we discuss the different types of queries and how they can be used.

Existential queries

All queries already presented in this chapter are existential queries. They report a violation of a policy for each path on which they encounter the violation. Existential queries can be defined as follows:

Definition 8. ***Existential queries****: Given an edge-labeled directed graph G where labels may have parameters, a vertex $v0$ in G, and a parametric regular-expression pattern P, compute all pairs of vertex v in G and a substitution θ for parameters in P such that there exists a path from $v0$ to v in G that matches some sentence accepted by P under θ.*

What this means is that *any* path in the state graph matching user-defined query will produce a resulting substitution. For that path, the policy has been violated, but chances are that that path never gets executed when actually running the program, as a consequence of the overapproximations of the static analysis in JIPDA. A branch of a conditional might never be executed in a program for example. If the value of the test in the conditional gets overapproximated (i.e. with an abstract value of $\{Bool\}$), the state graph will depict both branches, as it can't decide which branch will be taken. Existential queries thus match a pattern *if there exists a path* in the state graph matching the query.

Universal queries

Sometimes detecting if a pattern occurs along a path in the state graph doesn't suffice. Universal queries provide stronger guarantees for queries, as they require that the query matches for the same substitutions along all possible paths in the state graph between two states. We define universal queries as follows:

Definition 9. ***Universal queries****: Given an edge-labeled directed graph G where labels may have parameters, a vertex $v0$ in G, and a parametric regular-expression*

pattern P, compute all pairs of vertex v in G and substitution θ for parameters in P such that every path from $v0$ to v in G matches some sentence accepted by P under θ.

The intrinsic difference between universal and existential queries is how they match a pattern. Where it suffices for existential queries that just one path exists in the state graph, universal queries make sure that *all* paths between two points in the graph match the specified query. For the sake of a representative example, imagine a JS-QL predicate `def({name:'?x'})`, which checks all definitions and redefinitions of a variable bound to `?x`. This variable has a constant value in the state graph as long as no redefinitions of `?x` happens along any path between two states in the graph. We can then query for each state in the graph where `?x` has a constant value. The query in listing 4.10 shows how this can be expressed in JS-QL. Creating such predicates will be discussed in the next section. The definition of a variable `v` is matched, and any state following that matched state that isn't a redefinition of `v` will be contained in the result. All states in the state graph up until a redefinition of `v` (or the end of the graph) will then be marked in color in the user interface.

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

Listing 4.10: Checking for constant folding using a universal query

Query direction

Queries in most traditional systems are viewed as straightforward, in the sense that they match a part of a graph or other program representation from point a to b. This way of reasoning implies that queries are always matched from the beginning to the end of a program (if control-flow information is available, that is). Our framework supports this manner of querying in the traditional way, but also allows to explore the state graph bottom-up. This can be meaningful to search for certain program properties. *Forward* queries are queries as we've defined them until now. They match the state graph from the beginning state to the resultstates and are pretty easy to understand and read. *Backward* queries on the other hand traverse the graph in a bottom-up manner, meaning that the query starts at the end of the state graph and matches states until the starting state of the graph is reached. Although backward queries are less common they can be useful to perform some program analysis, such as live variables analysis. This analysis calculates for each program point the variables that may be potentially read before their next write. A variable is thus live if it holds a value that may be needed in the future. The backward query for this analysis goes as follows:

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

Listing 4.11: Live variables anlysis in JS-QL

The `use` and `def` predicates are again user-defined predicates. Starting from the resultstate of the program, some states are skipped until the first use of variable `?x` is found. Note that this is in fact the *last* use of that variable in the state graph for that liveness set. The query then marks all states that aren't a write to `?x`. In this way, one or more states will be marked, representing the path on which variable `?x` was live.

4.1.3 Defining predicates and policies

Expressing queries and policies with only the `state` and `wildcard` constructs quickly becomes tiresome as every attribute of the state has to be explicitly specified. The JS-QL framework remedies this by letting users specify their own predicates and policies as well as by providing some basic customizable predicates for single expressions and statements. We can distinguish predicates and policies by what they match. Predicates are like the `state` construct, in the sense that they match only one specific state. For example, JS-QL has a built-in predicate `functionCall` which matches function calls. The nice thing about these predicates is that a user can specify what he wants to match in a state. Policies on the other hand are sequences of predicates and/or `state`s, forming a pattern. We will demonstrate how to write predicates by dissecting a relatively simple built-in predicate called `assign`. All predicates and policies can be written in a separate file, as long as they are extend in the *JSQL prototype*:
`JSQL.prototype.assign = function(obj){...}`
This is the basic notation for named predicates, in this case the `assign` predicate. The dots in the code above will be filled in by the actual predicate code. We immidiately notice the `obj` argument. This argument represents the map of all properties of the state that need to be matched. By abstracting the map to a single `obj` variable, the user is free in which properties and attributes he wishes to match or omit for a specific query. The usefulness of predicates would drastically be reduced if the user has to again pass a nested map of properties to the predicate. We therefore let the developer of the predicate decide which properties he wishes to match, and how he names these properties in the predicate. For the example of the `assign` predicate, we chose to provide 3 basic attributes to the user: `this`, `left` and `right`, representing the whole assignment node, its left- and right-hand side resp.

```
1 var s = {}; // variable representing the state
2 var objThis  = this.getTmpIfUndefined(obj.this);
3 var objLeft  = this.getTmpIfUndefined(obj.left);
4 var objRight = this.getTmpIfUndefined(obj.right);
```

Listing 4.12: State properties of the `assign` predicate

A particularly handy feature of JavaScript is the named keys of a map. When accessing such a key in the map (like `obj.left` for example), the corresponding value is returned when found. When no such key exists, JavaScript returns `undefined`. To ease the use of predicates, some attributes in the `obj` map can be made optional. We do this by using the `getTmpIfUndefined` method, which returns a *temporary variable* when the value for its argument is `undefined`, and the regular value (a literal or a variable) when it is contained in the map. Temporary variables are variables that won't be contained in the resulting substitutions. By introducing these variables in the code, no conditionals have to be written that check whether an attribute has been specified in the map of a predicate, as we can then just use the temporary variable as a replacement. We instantiate a variable for each attribute as seen in the example. Remember that the attributes can have any name as long as it is clear to the user what the attribute stands for: `obj.right` could have also been `obj.abc`, but that would harm the readability of the predicate and confuse the user.

Mapping the attributes of the `obj` map to a state happens by setting up the state chain. To make things not too complicated for query and predicate developers, we provide a `setupStateChain` method which does just that. As can be seen in the example above, a map that mimics a state of the state graph is defined though variable `s`. This variable will be used to build the mimicked state. For each attribute that we provide through the predicate, an entry has to be made in the state map `s`. We indicate which piece of information we want to match by passing an array of keys we want to traverse in a state as a second argument to `setupStateChain`. The last key of this array will have the third argument as its value. The first argument is the map in which we want to store this information. When executing line 1 and 2 of listing 4.13, `s` will now contain 1 direct attribute `node`, which will in turn have 1 attribute `expression`, which will finally have 2 attributes, `left` and `right`. As we only match assignments, we filter the operator of the expression to be '=' on line 4. Note that this is not limited to querying the `node` property of a state. All information in the state graph can be queried through a predicate.

```
1 this.setupStateChain(s,['node','this'], objThis);
2 this.setupStateChain(s,['node','expression','left'], objLeft);
3 this.setupStateChain(s,['node','expression','right'], objRight);
4 this.setupStateChain(s,['node','expression','operator'], '=');
```

Listing 4.13: State chain setup of the `assign` predicate

We want to be able to specify properties, filters and lookups as in regular states. JS-QL allows this in the same way as in states. Putting everything together happens by finalizing each state map (only `s` in this case) and specifying the state(s) that match the predicate/policy. Listing 4.14 shows how finalization is done and how the pattern is specified. Finalizing handles things like lookups, filters and properties through the `finalize` method. This method extracts all relevant information from the `obj` map and adds it to the state map it gets as a first argument. Finally, the pattern is specified and each state in the pattern gets initialized its own designated state map. For predicates, there will always be at most one state map. Policies however can have multiple state maps, as they represent a sequence of states, each with their own map. This is the only characteristic in which predicates and policies differ. An example of a policy can be seen in listing A.4 in the appendix.

```
1 this.finalize(s, obj); //Finalize a specific state
2 return this.state(s);  //Fill in the state map
```

Listing 4.14: Finalizing the `assign` predicate

The full code for the predicate is listed in A.2 in the appendix. We can now use this predicate in any query, with the arguments that we wish to match in a state, as seen below:

```
1  G.skipZeroOrMore().assign({left: '?l'})
2  G.skipZeroOrMore().assign({right: '?r'})
3  G.skipZeroOrMore().assign({left: '?l', right:'?r',
4                             properties:{
5                                '?rName' : '?r.name'
6                             },
7                             lookup:{
8                                '?rName' : '?rAddr'
9                             }})
10 //And so on ...
```

Listing 4.15: Using the `assign` predicate

recursion

Some types of queries or analyses require that a variable is reintroduced in several consecutive states, or that a trace of information is kept along each state. This behavior can be achieved by defining *recursive queries*. Recursive queries are queries that can invoke themselves again, until a base case is matched. This type

of query can for example be used to detect by which variables a variable is tainted (i.e. influenced/marked). JS-QL supports recursive queries by providing the `rec` function. This function takes two arguments: (i) The mapping for the next recursive step and (ii) the predicate/policy that needs to be called recursively. The `taintedBy` policy is included in the appendix as listing A.3. It describes a naive taint analysis which only considers simple assignments. It takes three arguments, which can all be omitted: `orig` denotes the original value which will be aliased, `alias` represents the alias of the original value and `rec` keeps track of all variables that have been used as aliases inbetween `orig` and `alias`. The relevant code of the policy is found below:

```
1  this.setupStateChain(s1, //State map s1
2                       ['node','expression','right','name'], orig);
3  this.setupStateChain(s1, //State map s1
4                       ['node','expression','left','name'], alias);
5  this.setupStateChain(s2, //State map s2
6                       ['node','expression','right','name'], orig);
7  this.setupStateChain(s2, //State map s2
8                       ['node','expression','left','name'], flow);
9
10 return this .lBrace()
11             .state(s1)                    //1. From orig to alias
12             .or()
13             .state(s2)                    //2. From orig to flow
14             .skipZeroOrMore()             //   Skip some states
15             .rec(newObj,this.taintedBy) //   From flow to alias
16             .rBrace();
17
18 //JS-QL query:
19 G.skipZeroOrMore({orig: '?o', alias:'?alias', rec:'?r'})
```

Listing 4.16: Recursive call of the `taintedBy` policy

The policy matches all simple assignments by name. Lines 1-8 set up the state maps of 2 separate states. The first state matches a direct assignment from `orig` to `alias`. The second state does the same, but the alias in this case is an intermediate assigned variable `flow`. So when we assign a to b and b to c, the resulting state can look like: {orig: a, flow: b, alias: c}, indicating that c is an alias of a, and that it obtained the value of a through b. Lines 10-16 show the pattern that needs to be matched. Or we match an assignment directly (the base case), or we match a state from `orig` to `flow`, and later in the graph from `flow` to `alias`. The recursive call will keep occuring until no more match is found in the state graph, or when the base case is matched. This policy clearly shows that recursive queries can also be used to discover data flow properties of a program. When writing a policy similar to `taintedBy` which also keeps track of the address of the values, even more accurate data flow properties can be detected.

4.1.4 conclusion

In this chapter we described the syntax and usage of the JS-QL language. We discussed the basic constructs used to build queries in section 4.1.1, as well as more advanced predicates that are either built-in or that can be user-specified. JS-QL supports several types of policies, and each of them is described in section 4.1.2. Writing predicates and policies might seem tiresome at first glance, but actually is quite repetitive and easy to do. The benefit of predicates and policies is that they are 'write once, use often'. A method for defining custom predicates and policies is described in section 4.1.3. We can conclude that the JS-QL can be learned with little effort, after which expressive queries can be written to check all kinds of code characteristics.

Chapter 5

Implementation

This chapter describes the implementation details of the JS-QL framework and language, as presented in chapter 4. The implementation of JS-QL is publicly available[1] and can freely be used to test source code for characteristics and vulnerabilities.

5.1 Architecture

The architecture of the implementation separates each component in a different module, providing the possibility to replace these modules by alternative implementations. In this section we discuss what each component represents.

Datastructures

The `DataStructures` module defines all datastructures used in JS-QL. These include all kinds of tuples used for our matching algorithm, but also the alternative representations of edges and nodes to transform the abstract state graph to a compatible graph for the matching engine.

AbstractQuery

The `AbstractQuery` module defines the core of the matching engine. It contains all operations on substitution sets: Matching, merging, defining extra properties, filters and lookups. The module provides an interface to the actual query algorithms `ExistentialQuery` and `UniversalQuery`, which perform the actual query matching processes.

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

Automaton

The `Automaton` module defines the uniform representation of finite state machines (also known as finite automatons). It abstracts away whether the automaton is a deterministic or non-deterministic automaton, and provides information about the accepting states and starting state of these automatons.

ThompsonConstruction

The `ThompsonConstruction` module defines an algorithm to convert a regular expression to a NFA (non-deterministic finite automaton). It parses the regular path expression and adds zero or more states to the newly created automaton for each step in the parsing process.

SubsetConstruction

The `ThompsonConstruction` module defines an algorithm to convert a NFA to a DFA (deterministic finite automaton). What this means is that it eliminates all ϵ-transitions, which are transitions that can occur without reading an input symbol. The resulting automaton is used for both query algorithms.

JipdaInfo

The `JipdaInfo` module transforms state information to a more readable format for users. This transformation is necessary to have enforce consistency in states representing an AST node. The transformed states are the actual states that are queried instead of the original JIPDA states.

JSQL

The `JSQL` module defines the internals of the JS-QL query language. It is implemented as an embedded DSL in JavaScript and allows users to define application-specific predicates and policies. The module is made available for the user through a fluent interface, increasing the readability of the language.

SecurityAnalysis

The `SecurityAnalysis` module glues every component together in the framework. The initialization and transformation of the abstract state graph, execution of queries and processing query results are all invoked by this module.

5.2 The user interface

The JIPDA analysis comes packed with an interface which enables the user to inspect the abstract state graph. This state graph is generated when the user provides an input program and a lattice to perform the abstract interpretation. For our framework we augmented this user interface in several ways described in this section. Figure 5.1 shows the user interface, illustrating what we will discuss in the next sections.

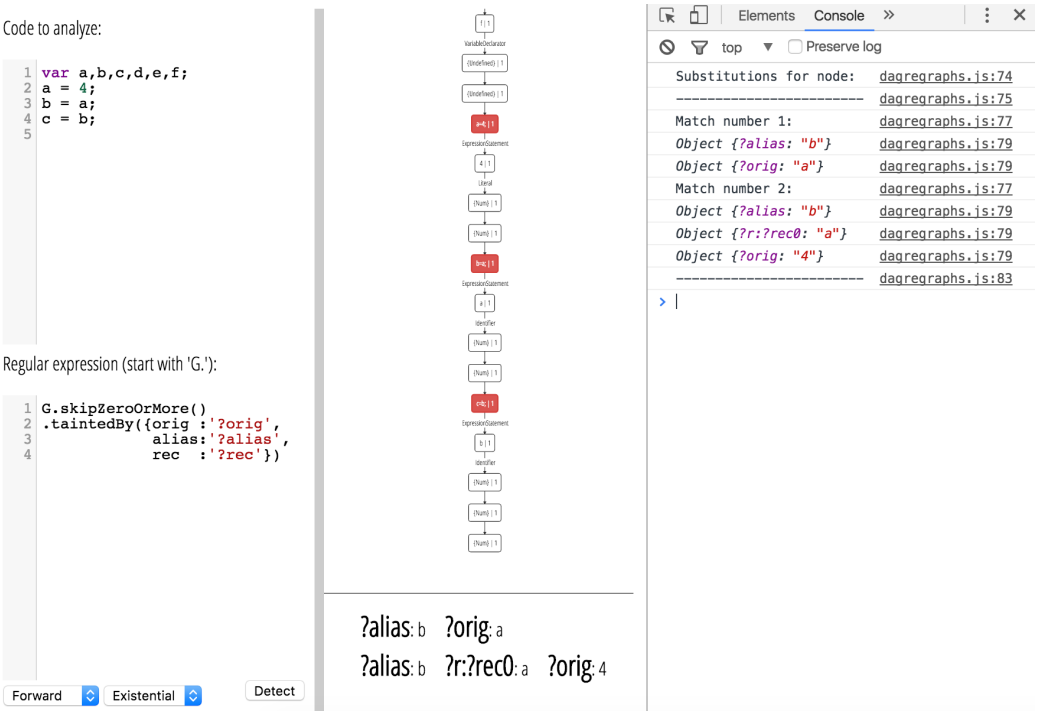

Figure 5.1: The JS-QL framework user interface

Query interpretation

We decided to allow the user to specify queries in the user interface itself. In this way we avoid to switch between different screens and/or files to run just one query. The user interface contains a third-party JavaScript plugin which enables syntax highlighting. We use this plugin as input textboxes for a query and the input program. Along with entering the needed information, a user can decide the kind of query that needs to be executed. This can be done by selecting the direction together with type of query (existential or universal). When hitting the 'detect' button, internally a new `Query` object is made with the provided query string.

Query objects contain three fields: The query direction, its type and a `JSQL` object, representing the instantiated query. Although being unsafe, the query string is converted to a JSQL object by `eval`uating it. For the sake of this dissertation, the safety of the application isn't relevant for its functionality.

The abstract state graph

JIPDA already provided the functionality to display the abstract state graph in the user interface. In order to show results, this representation of the graph needed to be modified so that it became optimal for the user to reason about query results. We changed the following:

1. *Edge labels*: The JIPDA state graph already contained edge labels, but the information they contained was irrelevant for our approach. As our approach uses edge labels to match states, we had to shift information from nodes to edges. All evaluation states' outgoing edges are also augmented with a visible edge label representing the type of AST node it contains, to make it easier for the user to see what he has to specify in his queries.

2. *State colours*: The default colours of all JIPDA states have been stripped from the graph. This was done to increase the contrast with marked states (i.e. states indicating a match of the query). When the matching engine produced all results, these results have to be transferred to the corresponding states of the state graph. A 'marker' property was added to each matched state, containing the match information as well as a CSS class to highlight them in the otherwise colourless graph. This CSS class can be customised by the user.

Results inspection

Each query result is a set of substitutions, mapping variables (denoted by a starting `?`) to their corresponding values in the state graph. Every matching state is marked with its substitution set(s). These results can then be further explored through the results section under the state graph or in the browser's built-in console, which allows to inspect results in even greater detail.

5.3 The query language

The JS-QL query language is implemented as an embedded DSL with JavaScript as its host language. We motivated the use of a DSL in chapter 3 and explored the

JS-QL syntax and semantics in chapter 4. This section describes how the DSL is implemented and how we incorporated several DSL implementation techniques.

A JS-QL query gets 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 5 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 just 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 doesn't need to be specified when no recursion happens in a query.

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

States and predicates are actually just function calls returning `this` to enable method chaining (which is a commonly used technique to implement a fluent interface). Each function call actually 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. An exception to this are recursive queries. Recursive query patterns can have an arbitrary amount of states, so we can't model them directly as a sequence of `RegexParts` as we don't know the length of the actually matched pattern. We therefore store a whole recursive query in just one `RegexPart` object, and mark it with 'subgraph' as its name. Further we specify the `ExpandFunction` and `-Context` to be able to process the subgraph in the matching algorithm. The idea for treating recursive queries like was adopted from the PQL language[28]. The information in this map is unmodified in the sense that the variables aren't resolved yet. The entire query pattern (i.e. the map) is then processed by applying *Thompson's Construction Algorithm* and the *Subset Construction Algorithm* consecutively to obtain a NFA and DFA respectively. These algorithms won't be discussed in too much detail, as they are well described in many online resources and in the literature[37]. An overview of how queries are processed is depicted in figure 5.2.

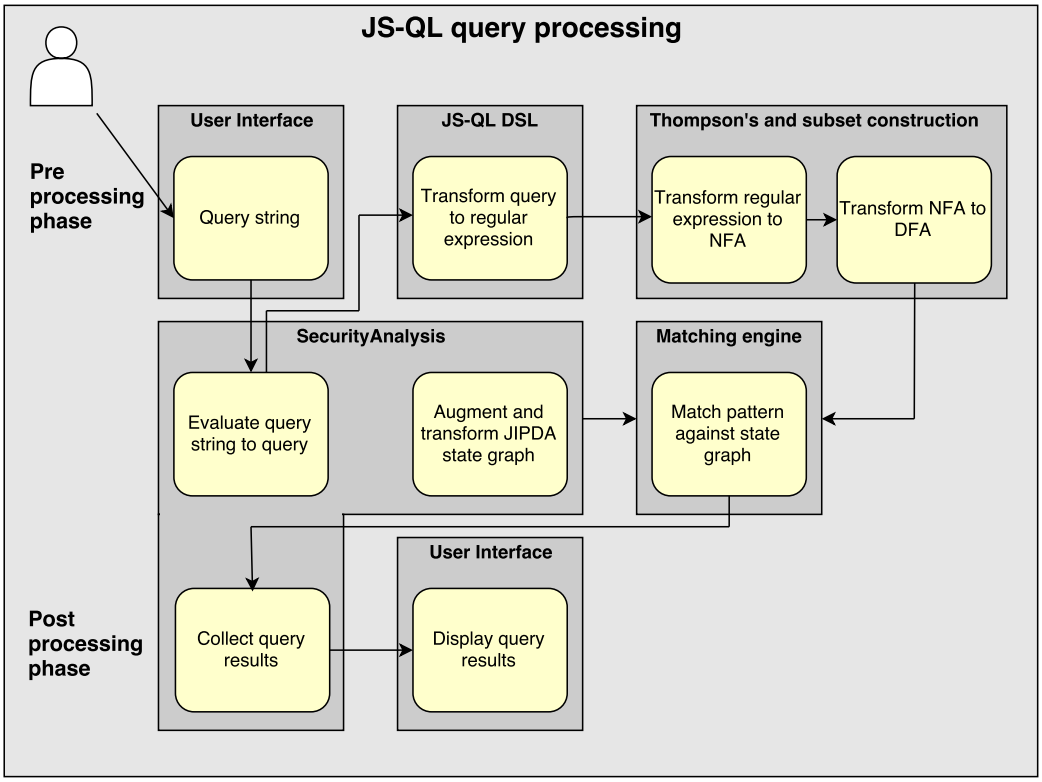

Figure 5.2: Stages of processing a query

Regular, temporary and recursive variables

JS-QL supports three types of variables: Regular, temporary and recursive variables. Each of these types of variables has its own function in the framework:

1. *Regular variables*: These variables contain the information that the user wants to match in a query. When a match succeeds, they will always be contained in the resulting substitutions.

2. *Recursive variables*: Variables that are used as intermediary variables. They function as a variable that was 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. The `taintedBy` example used in chapter 4 uses a recursive variable to store all intermediary assignments.

3. *Temporary variables*: These variables are state-local. They are used when a user doesn't specify a certain argument for a predicate or policy. When only specifying the `left` argument for the `assign` predicate shown in chapter

4, the `right` and `this` will be bound to temporary variables. These variables get dropped from the resulting substitutions.

By allowing the user to choose from three types of variables, writing queries becomes very flexible. In this way 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 grow very large very quickly. This greatly decreases the readability of the results and makes interpretation of these results much harder.

Deferring evaluation of properties and filters

One way to define properties is to use the `prop` function. Filters on the other hand can be expressed through the `cond` function. Both of these functions have one thing in common: The value they return depends on the value of the variables specified by the user. The problem now is that the value of these variables is not yet calculated at compile-time. Consider the following example: `cond('===', '?var', 3)`. Here, the state declaring `?var` is not yet matched in the state graph, meaning that `?var` is not yet bound and that we can't check if it is equal to 3. We remedy this by deferring the evaluation of the specified function. Instead of passing the result of evaluating the function directly with its arguments, we wrap them into a *thunk* and pass that thunk to the matching engine. When the matching engine finally needs the value of the thunk, it unwraps it and resolves all variables to the values to which they are bound. If not all variables are bound, the query fails.

5.4 The matching engine

Chapter 6

Evaluation

In this chapter we validate and evaluate the expressiveness of the JS-QL query language by expressing some existing security policies, described in other related work. We start this chapter by expressing 9 security policies distilled from 3 papers in sections 6.2, 6.1 and 6.3 respectively. Every JS-QL policy will be evaluated by comparing how well it matches the policy expressed in the original paper. Finally, in section 6.4, we evaluate the query framework by specifying its advantages and limitations. We will also briefly compare the query languages presented in this chapter in terms of expressiveness, verbosity and conciseness.

6.1 The GateKeeper language

In this section we attempt to express 3 policies originally presented in [14].

6.1.1 Writes to prototype objects

Many websites use bookmarklets to store user information to automate the login process, for example [2]. This is a common strategy used to reduce the amount of information the user has enter every time he visits the website. An attacker website however can alter the JavaScript environment in such a way that he can steal all of this information from the user. Imagine a simple login function which checks the current location of the webpage to verify that it is on the correct webpage. The current location can be compromised by overwriting the `toString` function of the `String` object, as depicted in 6.1. This function can be configured to always return a ”good” location. In this way, the login function can be called in the environment of a malicious website, possibly leaking sensitive information.

```
1 String.prototype.toString = function(){
2     //Always return "spoofed" url
```

```
3       return "www.goodwebsite.com";
4   }
5
6   var login = function(){
7     if(document.location.toString() === "www.goodwebsite.com"){
8       //leak information on untrusted website
9     }
10  }
```

Listing 6.1: Prototype poisoning example

Gatekeeper expresses policies by defining a set of rules in datalog. In order to detect writes to prototypes of builtin objects, they define the `FrozenViolation(v)` predicade, as shown in listing 6.2. 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 builtin 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 builtin object. The `Reaches(h1,h2)` predicate makes sure that the field that was stored reaches the builtin object directly or indirectly, by recursively checking if one of the properties of the builtin object has a field pointing to the stored field.

```
1   Reaches(h1,h2) :- HeapPtsTo(h1,_,h2).
2   Reaches(h1,h2) :- HeapPtsTo(h1,_,h3),
3                     Reaches(h3,h2).
4
5   FrozenViolation(v) :- Store(v,_,_),
6                         PtsTo(v,h2),
7                         BuiltInObject(h1),
8                         Reaches(h1,h2).
9
10  % Specify all built in objects
11  BuiltInObject(h) :- GlobalSym("String", h).
12  BuiltInObject(h) :- GlobalSym("Array", h).
13  % ...
14
15  GlobalSym(m,h) :- PtsTo("global", g),
16                    HeapPtsTo(g,m,h).
```

Listing 6.2: Policy 1 in GateKeeper

Writing this policy in JS-QL is easy. To ease the work for the programmer, we augmented the Jipda-nodes corresponding with `MemberExpressions` two extra fields: `mainObjectName` and `properties`, representing the root object and the property-chain array that was accessed respectively. An example: for `o.x.y.z`, `o` would be the `mainObjectName`, and `[x,y,z]` would be the array `properties` which represents the properties that were chained. Listing

6.3 depicts the JS-QL query to efficiently express this policy. Note that the filter on lines 10-12 can be omitted. This filter simply indicates that we only want to detect writes to the `prototype` property of the `String` object. When this is omitted, we will detect all writes to this object.

```
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 6.3: Policy 1 in JS-QL

This example JS-QL policy only detects writes to the `String` object. We wrote a compound policy `writeToBuiltinObjectPrototype` to detect writes to all builtin objects' prototype property. The code for this policy can be found in listing A.4 in the appendix. This policy is just the disjunction of states similar to the state in listing 6.3, with the only difference in the `mainObjectName` property, which corresponds to a different builtin object name.

6.1.2 Global namespace pollution

Working in a JavaScript environment often involves the inclusion of multiple (third-party) scripts. These scripts offer instant access to functionality which would be tiresome to implement for every project yourself. Some of these scripts are written by other parties, so one can't be sure that they follow the same coding guidelines as he does. Inexperienced programmers might not be aware of the JavaScript namespacing patterns [31]. This leaves an open window for a phenomenon called "global namespace pollution". Defining variables in the global scope in JavaScript can lead to unanticipated behaviour of the program when another script defines a global variable with the same name.

Preventing stores to the global object (i.e. in the global scope) can be enforced through a simple two-lined GateKeeper policy. GateKeeper handles the global object explicitly by defining a variable `global`. Global variables can then be simulated as fields of this object. Note that JIPDA does this in a similar way. A policy to detect global stores can then be defined as in 6.4: The global object

variable is located on address `g`. Every field store `h` that points to a field of `g` will then be detected by the `GlobalStore` policy.

```
1 GlobalStore(h) :- PtsTo("global",g),
2                   HeapPtsTo(g,_,h).
```

Listing 6.4: Policy 2 in GateKeeper

We could write a similar policy in JS-QL that would also look if the address of the variable points to the global object. However, this is more difficult in our system. Not because of any language restrictions, but because of the nature of JIPDA. When a variable or function gets declared or when a variable is assigned to, the right-hand side first has to be evaluated. This is also reflected in the JIPDA graph. Only when the expression is evaluated, the store and environment are modified to contain the recently evaluated information. What this means is that the allocation address for newly created variables isn't yet available in the states we query on lines 3,5 and 7 in listing 6.5. We remedy this by looking a bit further down the graph, more specifically in the states where this information *IS* available. The policy goes as follows: After skipping to an assignment or a declaration of a function or variable, we bind the name the variable's or function's name to metavariable `?name`. We then again skip some nodes until we find a state where the address of `?name` is available and bind it to `?nameAddress`. Finally, we search for the variable or function with the same name in the global object and also bind it to `?nameAddr`, which filters the resulting substitutions to only contain information about globally declared objects.

```
1  G.skipZeroOrMore()
2  .lBrace()
3    .assign({leftName:'?name'})
4    .or()
5    .variableDeclaration({leftName:'?name'})
6    .or()
7    .functionDeclaration({name:'?name'})
8  .rBrace()
9  .skipZeroOrMore()
10 .state({lookup:{
11       '?name': '?nameAddr',
12       '_global.?name' : '?nameAddr'
13     }})
```

Listing 6.5: Policy 2 in JS-QL

6.1.3 Script inclusions

A well known exploit in JavaScript environments is *heap spraying*[6]. This is an attacking technique that can eventually even compromise a user's system. In short,

it arranges the layout of the heap by allocating a vast amount of carefully-chosen strings, installing a certain sequence of bytes at a predetermined location in the memory of a target. When this is achieved, the exploit is triggered. This trigger depends on the user's operating system and browser. Such an agressive attack can be instantiated on the victim's computer by simply including a malicious script. This could be a reason to write a policy which detects all script inclusions. Regular script inclusions through `<script></script>` tags can be detected by hand. Javascript however also allows programmers to write arbitrary HTML code by using the `document.write` and `document.writeln` functions. Listing 6.6 gives an example of malicious script inclusions.

```
1  var evilScript;
2  var scripts = ["<script>bad1</script>","<script>bad2</script>"];
3
4  for(var i = 0; i < scripts.length; i++){
5    evilScript = scripts[i];
6    document.write(evilScript); //violation
7  }
8
9  var o = {};
10 o.f = document.writeln;
11 o.f("<script>bad3</script>"); //Violation
```

Listing 6.6: Script inclusion example

This policy can be written with only a few lines of datalog in GateKeeper. What needs to be detected are the calls to `document.write/document.writeln`, even when they are aliased. This is important to note because scripts used for attacks are often obfuscated. The policy in listing 6.7 does just that. `DocumentWrite(i)` first looks for the address `d` on the heap which points to the global `document` object. Next, the location of the property `write/writeln` of that object is reified in variable `m`. This is also an address on the heap. The last step is to find all call sites `i` that point to that same address on the heap.

```
1  DocumentWrite(i) :- GlobalSym("document",d),
2                      HeapPtsTo(d,"write",m),
3                      Calls(i,m).
4
5  DocumentWrite(i) :- GlobalSym("document",d),
6                      HeapPtsTo(d,"writeln",m),
7                      Calls(i,m).
```

Listing 6.7: Policy 3 in GateKeeper

JS-QL also proves to be suitable to express such a policy in listing 6.8. The approach we take first skips zero or more states in the JIPDA graph. We specify that we then want to find a function call with the name of the function bound

to metavariable `?name`. In order to know to which address the called function points in the store, we look it up and bind the address to `?addr` in the lookup-clause of the `fCall` predicate. Finally we also match the address of `document.write/document.writeln` to the same `?addr` metavariable, filtering out all function calls that do not point to this address.

The analysis that we use is context-sensitive and Javascript is lexically scoped. This implies that we need to explicitly specify that we are looking for the address of the *global* `document.write/document.writeln` object. If we didn't do this and the user has defined an object with the name "document" and a property "write" or "writeln" inside the scope of the current state in the graph, we would get the address of that object instead of the global object. That is why JS-QL provides a `_global` keyword which indicates that we need to search for the address in the global namespace.

```
1  G.skipZeroOrMore()
2  .lBrace()
3  .fCall({
4    name: '?name',
5    lookup:{
6      '?name'    : '?addr',
7      '_global.document.write': '?addr',
8      }
9  })
10 .or()
11 .fCall({
12   name: '?name',
13   lookup:{
14     '?name'    : '?addr',
15     '_global.document.writeln': '?addr',
16     }
17 })
18 .rBrace()
```

Listing 6.8: Policy 3 in JS-QL

6.1.4 Conclusion

In this section we expressed 3 policies in the GateKeeper language and JS-QL. As table 6.1 indicates, all policies expressed in GateKeeper were also expressible in JS-QL. Gatekeeper excels in writing concise policies to detect certain individual properties of a program. It is however difficult, if not impossible, to express a policy which finds a sequence of properties in a program. JS-QL does not have this problem. The language is designed to match states along an abstract state graph. While it 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 policy in GateKeeper to find all function calls to methods that take more than four arguments (This is a bad code smell according to [38]). JS-QL provides the `properties` and `filters` constructs to express this.

Language	Policy 1	Policy 2	Policy 3
Gatekeeper	✓	✓	✓
JS-QL	✓	✓	✓

Legend: ✓: Fully expressible

Table 6.1: Expressiveness in JS-QL and GateKeeper

We conclude that our language is more expressive since we are able to express sequences and extra properties/filters for example, increasing the flexibility of policies. GateKeeper on the other hand is less verbose in most situations. This is because we have to express everything we want to detect inside the constructs of JS-QL (like `state({...})`). Data flow analysis for example happens behind the scenes in GateKeeper, whereas JS-QL has to do the checks for aliasing in the language itself. An example can be seen in 6.1.3, where we have to explicitly match the address of the called function to the address of `document.write/writeln`. This matching happens internally in Gatekeeper.

6.2 The PidginQL language

In this section we attempt to express 3 policies originally presented in [23].

6.2.1 Only CMS administrators can send a message to all CMS users

Imagine a situation where not only administrators can send broadcast messages. A regular user with bad intentions could easily take advantage of this situation to cause harm to the system. A CMS application for instance with a decent size of users could be exploited by sending a message to all users, asking them to reply with their password. When the attacker provides a reason to the victims convincing them to send their password, he could possibly compromise the contents of the victim's account. An example of such a reason could be that the 'administrator'

needs to have the password of a user account in order to update the software of that user to the latest version. This behaviour is undesirable, thus we need a policy which prevents regular users from sending such messages.

The policy described in [23] that addresses this issue can be found in listing 6.9. First, all nodes that are entries of the `addNotice` method are searched for and stored in a variable. `addNotice` is the method that sends messages to all users, and has the same behaviour as the broadcast method in the explanation above. 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. When this intersection is not empty, we can assume that there is a violation of the policy in the remainder of the graph. This last part is exactly what the `accessControlled` method does.

```
1 let accessControlled(G, checks, sensitiveOps) =
2     G.removeControlDeps(checks) ∩ sensitiveOps is empty
3
4 let addNotice = pgm.entriesOf("addNotice") in
5 let isAdmin   = pgm.returnsOf("isCMSAdmin") in
6 let isAdminTrue = pgm.findPCNodes(isAdmin,TRUE) in
7                  pgm.accessControlled(isAdminTrue, addNotice)
```

Listing 6.9: Policy 4 in PidginQL

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." We must note that with abstract interpretation, it is not trivial to specify whether a value is truthy or falsy. 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 of listing 6.10, 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 *?lkont*. The next time we find a state with the same context and stack, 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.

While this policy 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. It would be ideal to have a means

to express the *XOR* relation between results of the JS-QL policies. If we had this kind of mechanism at hands, we could search for all calls to `addNotice` and the calls to `addNotice` that happen in the true branch of `isCMSAdmin` and remove all states that occur in both results. The result of this removal would then contain only the violations of the policy.

```
G.skipZeroOrMore()
.ifStatement({alt:'?alt'})
.skipZeroOrMore()
.fCall({name:'isCMSAdmin'})
.skipZeroOrMore()
.state({node:'?alt', kont:'?k',lkont:'?lk'})
.not().endIf({kont: '?k', lkont:'?lk'}).star()
.fCall({name:'addNotice'})
```

Listing 6.10: Policy 4 in JS-QL

6.2.2 Public outputs do not depend on a user's password, unless it has been cryptographically hashed

Password information is something most people want to keep to themselves. It is therefore not desirable that sensitive information about this password is leaked in any way to public outputs. This leak of information doesn't have to be explicit however. Imagine a situation where a malicious piece of code checks if the length of the password is larger than 5. If the condition is true the output will display 1, otherwise the output is 0. This also reveals information about the password, and thus should be treated as a violation. The name for this kind of information flow is *implicit flow*.

```
var password = getPassword();
//computeHash(password);
var message;
if(password.length() > 5){
  message = 1;
  print(message);
}
else{
  message = 0;
  print(message);
}
```

Listing 6.11: The output depends on the password example

Since the PidginQL paper represents the program as a program dependence graph, the 'depends' relation is easily checked. In the graph there must be no path between the retrieval of the password and an output, unless `computeHash`

was called. *Declassification* happens when calling this method, which means that from then on the password is sanitized and ready to flow to an output. The policy in listing 6.12 displays how this can be expressed in the PidginQL language.

```
1 let passwords = pgm.returnsOf("getPassword") in
2 let outputs   = pgm.formalsOf("writeToStorage") ∪
3                 pgm.formalsOf("print") in
4 let hashFormals = pgm.formalsOf("computeHash") in
5 pgm.declassifies(hashFormals, passwords, outputs)
```

Listing 6.12: Policy 5 in PidginQL

The scenario for which we write a policy in JS-QL is as follows: An output depends on the password when the password is used in a conditional expression. In one or more of the branches of this conditional expression an output function is then called. The example code on which we test our policy is listed in listing 6.11. We look for a state in the graph where the password is returned, and we store the address in `?addr`. The program then continues for some states in which the `computeHash` method is *not* called with the password as an argument (lines 3-16). We then match a state representing a conditional node, in this case an `IfStatement` for which we bind the true branch to `?cons` and the false branch to `?alt`. Note that in the JIPDA abstract state graph, all evaluation steps are visible in the graph. This gives us an opportunity to check if somewhere in the condition of the conditional the password is used, before the actual branching happens. The `variableUse` predicate on line 19 performs this check. It matches any state in which a variable is used. The declarative nature of the predicates allows us to pass the address of the variable as a metavariable, so that we can specify that we only want to match the uses of the variable whose address is alread captured in `?addr`. When this results in a match, we know that the variable has been used in the evaluation of the condition of the conditional. Finally, we proceed by checking if an output function (`print` in this case) is called *inside* one of the branches of the conditional. We do this by matching the nodes of states to the already bound `?cons` and `?alt`. A match indicates that that state is the beginning of the true branch or false branch respectively. For these branches, we capture the context and current stack in two additional metavariables `?k` and `?lk`. These will be needed on line 26 to indicate that we want to find the call to `print` *before* the branch ends. This policy, found in listing 6.13, can be made more general by writing a predicate which captures all conditionals instead of just `IfStatements`.

```
1 G.skipZeroOrMore()
2 .procedureExit({functionName:'getPassword', returnAddr : '?addr'
    })
3 .not()
4   .state({
```

```
5       node:{
6         expression: {
7             callee: { name:'computeHash' },
8             arguments: '?args'
9         }
10      },
11      properties: {
12        '?arg' : prop('memberOf', '?args'),
13        '?firstName': '?arg.name'
14      },
15      lookup:{ '?firstName' : '?addr' }
16    }).star()
17  .ifStatement({cons:'?cons', alt:'?alt'})
18  .skipZeroOrMore()
19  .variableUse({addr:'?addr'})
20  .skipZeroOrMore()
21  .lBrace()
22    .state({node:{this:'?cons'}, kont:'?k', lkont:'?lk'})
23    .or()
24    .state({node:{this:'?alt'}, kont:'?k', lkont:'?lk'})
25  .rBrace()
26  .not().state({kont:'?k', lkont:'?lk'}).star()
27  .fCall({name: 'print'})
```

Listing 6.13: Policy 5 in JSQL

6.2.3 A database is opened only after the master password is checked or when creating a new database

A database can contain a lot of sensitive information, so it is important that only authorized people can access this information. It might thus be a good idea to restrict access to the database entirely, unless upon creation or when the correct credentials can be presented.

The PidginQL query in listing 6.14 describes the query pattern in pseudocode, since they had no clean way of expressing this policy. All nodes corresponding to checks of the master password are stored in the `check` variable. Lines 2 and 3 remove these nodes from the graph when the condition is true (i.e. when the master password is correct). Lastly, the nodes where the creation of a new database occurs are also deleted from the graph, resulting in a graph which consists of only nodes that represent the opening of the database. If the graph is empty, then there are no violations found.

```
1 let check = (all checks of the password)
2 let checkTrue = pdg.findPCNodes(check, TRUE) in
3 let notChecked = pdg.removeControlDeps(checkTrue) in
```

```
4 let newDB = (method to create database)
5 let openDB = (method called to open the database)
6 notChecked.removeNodes(newDB) and openDB is empty
```

Listing 6.14: Policy 6 in PidginQL

Although PidginQL doesn't offer a concrete implementation of the policy, JS-QL does. We created 2 policies that provide full coverage for the problem that is presented in this section, listed in listing 6.15. The problem can be worded otherwise: We want to find all calls to `openDatabase` that are not inside the true branch of a conditional that checks if the master password is correct. When described like this, the policy gets much more intuitive to express in JS-QL. The policy can be split up in two parts: The first part will skip to an `IfStatement` of which we bind the true branch to `?cons`, as in the previous example. We then again check if the condition of that statement uses the `isMasterPassword` to verify the correctness of the password. We want to look into all states for which this condition doesn't hold, which is described on line 7. In this case all calls to `openDatabase` are prohibited, except inside the `newDatabase` function. This policy catches all violations *after* the first matching `IfStatement`. That is why there is the need for a second part in the policy. The detection of all calls to the `openDataBase` function completes this policy, but adds as a side effect that it will add false positives. These false positives will be the calls to `openDatabase` that occur when the master password is correct. This confirms the need for the *XOR* relation, as described in the previous section.

```
1  G.skipZeroOrMore()
2  .lBrace()
3    .lBrace()
4      .ifStatement({cons:'?cons'})
5      .skipZeroOrMore()
6      .fCall({name:'isMasterPassword'})
7      .not().state({node:'?cons'}).star()
8      .beginApply({name:'?name', lkont:'?lk', kont:'?k',
9                   filters:[
10                   cond('!==', '?name', 'newDatabase')
11                  ]})
12     .not().endApply({lkont:'?lk', kont:'?k'}).star()
13     .fCall({name:'openDatabase'})
14   .rBrace()
15   .or()
16   .fCall({name:'openDatabase'})
17 .rBrace()
```

Listing 6.15: Policy 6 in JS-QL

6.2.4 Conclusion

In this section we expressed 3 policies in the PidginQL language and JS-QL. Table 6.2 indicates that not all three policies were easily expressible. We are able to express all 3 policies in JS-QL, but 2 of them will have results containing false positives. These two policies each consisted of two separate queries. If we wish to attain a resultset only containing violations and no false positives, we could take the exclusive disjunction of the resultsets of these separate queries. The PidginQL language is best at expressing policies 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[11], 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 technique allows us to detect a wide range of general and complex properties about a program, but sometimes has troubles detect dependencies between states with only one policy. PidginQl may be powerful in finding dependencies as described above, it does however not return much meaningful information about the found violations. Where JS-QL returns all violating nodes marked in a GUI, PidginQl just indicates whether there are violations or not. It doesn't specify which nodes are violating the policy.

Language	Policy 4	Policy 5	Policy 6
PidginQL	✓	✓	✓
JS-QL	❍	✓	❍

Legend: ✓: Fully expressible, ❍: Expressible with false positives

Table 6.2: Expressiveness in JS-QL and GateKeeper

Another restriction in PidginQL is that there is no way to reason about the internals of a state in the graph. Our language allows the programmer to query information in the graph on the level of each state. We can dig inside a state at any time and specify the information we wish to obtain in some user-declared metavariables. This is not possible in PidginQL. This expressiveness and flexibility brings along that JS-QL queries and policies will often be more verbose.

We can conclude that both languages are equally expressive in their own way. While JS-QL can be used for many different domains, PidginQL is especially strong in it's own domain, namely in querying for dependencies between nodes.

6.3 The ConScript language

In this section we attempt to express 3 policies originally presented in [29].

6.3.1 No string arguments to setInterval, setTimeout

`setInterval` and `setTimeout` take a callback function as a first argument. This function is fired after a certain interal or timeout. Surprisingly, a string argument can also be passed as the first argument. This is good news for possible attackers, because the string gets evaluated as if it were a regular, good-behaving piece of JavaScript code. Malicious code can then be passed as a string argument to `setInterval/setTimeout`, which can lead to a security threat.

```
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 6.16: No string arguments to setTimeout

ConScript is an aspect-oriented advice language that deals with security violations just like this. The aspects are written in JavaScript, which enables the programmer to make full use of the language. They also provide a typesystem which assures that the policies are written correctly, as can be seen in listing 6.17 on line 1. Lines 10-11 depict the actual registration of the advice on the `setInterval` and `setTimeout` functions. When called, the `onlyFnc` function will be 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. We consider this as a small hack, since it has no semantic additional value for the policy itself.

```
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 6.17: Policy 7 in ConScript

Since we can’t reason about concrete values in abstract interpretation, writing a policy that only allows strings might seem a little more tricky. This is not the case because the lattice we use gives us information about the type of the value of

variables. A string for example is indicated by the lattice value {Str}. We can then define a isString helper function which checks whether a variable is of type String or not. The JS-QL policy in listing 6.18 uses this function to determine whether the looked up value of the ?name variable is of type String or not. The policy looks for a call of the setTimeout function and binds its arguments to ?args. memberOf is a powerful construct which creates a new substitution set for each of the elements in the list that it takes as an argument. This allows us to inspect and check each individual argument ?arg of the setTimeout function. We take the name of the argument and look up its value in the lookup clause. What remains is to filter out the string arguments, as already discussed above. This policy will only detect the actual violation on line x in listing 6.16.

```
1  G.skipZeroOrMore()
2  .fCall({
3    name:'setTimeout',
4    arguments:'?args',
5    properties:{
6      '?arg' : prop('memberOf', '?args'),
7      '?name': '?arg.name',
8    },
9    lookup:{'?name': '?lookedUp'},
10   filters:[
11     cond('isString', '?lookedUp')
12   ]
13 })
```

Listing 6.18: Policy 7 in JS-QL

6.3.2 HTTP-cookies only

Servers often store state information on the client in the form of cookies. They do this to avoid the cost of maintaining session state between calls to the server. Cookies may therefore contain sensitive information that may only be accessed by the server, so it might be a good idea to prohibit reads and writes to the client's cookies. These are stored in the global document.cookie object. Listing 6.19 gives an example of possible violations.

```
1  var doc, cookie1, cookie2, cookie3, badFunc;
2  badFunc = function(){
3    var bad;
4    bad = document.cookie;          //Violation (read)
5    return bad;
6  }
7
8  cookie1 = document.cookie;        //Violation (read)
9  doc = document;
```

```
10 cookie2 = doc.cookie;            //Violation (read)
11 cookie3 = badFunc();             //Violation (read)
12 document.cookie = {value:"bad"} //Violation (write)
```

Listing 6.19: HTTP-cookies only example

Registering advices around functions is easy. In conscript, the above policy can be enforced with only a few lines of code. Listing 6.20 wraps reads and writes of the ”cookie” field of `document` in the `httpOnly` advice. An error is thrown when a violation against this policy is encountered.

```
1 let httpOnly:K->K=function(_:K){
2     curse();
3     throw "HTTP-only cookies";
4 };
5 around(getField(document, "cookie"), httpOnly);
6 around(setField(document, "cookie"), httpOnly);
```

Listing 6.20: Policy 8 in ConScript

Writing an equivalent JS-QL policy proves to be a little more verbose. The reason for this is that we only work with our own embedded DSL to query the information in the JIPDA graph. While the `getField` and `setField` in 6.20 handle the lookup of the address of `document.cookie`, we have to manually specify that we want to store the address in metavariable `?cookieAddr` and try to match it with the address of the `?name` metavariable, which we assign to the same metavariable `?cookieAddr` to filter out variables with a different address. The JS-QL policy in 6.21 specifies that it will only detect writes (the first `assign` predicate) and reads (the `procedureExit` and second `assign` predicate) of the `?name` variable which points to the address of the global `document.cookie` object. It is easy to see what the `assign` predicate does: In this case, it matches the left or right name of the assignment and looks it up. The `procedureExit` is an extra predicate which marks all returns of functions that return a value that again points to the address of the global `document.cookie` address.

```
1  G.skipZeroOrMore()
2  .lBrace()
3    .assign({leftName:'?name',
4             lookup:
5             {
6                 '_global.document.cookie' : '?cookieAddr',
7                 '?name'                   : '?cookieAddr'
8             }
9            })
10   .or()
11   .assign({rightName:'?name',
12            lookup:
```

```
13              {
14                  '_global.document.cookie' : '?cookieAddr',
15                  '?name'                   : '?cookieAddr'
16              }
17             })
18    .or()
19    .procedureExit({returnName:'?name',
20                    lookup:
21                    {
22                        '_global.document.cookie' : '?cookieAddr',
23                        '?name'                   : '?cookieAddr'
24                    }
25                   })
26 .rBrace()
```

Listing 6.21: Policy 8 in JS-QL

6.3.3 Prevent resource abuse

Malicious scripts can prevent parts of a program to be accessible by users. Think of a website you want to access, but every time you scroll or click a mouse button, a popup appears. This is a form of resource abuse, namely the abuse of modal dialogs. This can be prevented by prohibiting calls to functions that create these resources. The ConScript policy is similar to the policy discussed in section 6.3.2. Calls to `prompt` and `alert` are wrapped in an advice which throws an error. Listing 6.22 shows the source code of the policy.

```
1 let err : K -> K = function () {
2     curse();
3     throw 'err';
4 });
5 around(prompt, err);
6 around(alert, err);
```

Listing 6.22: Policy 9 in ConScript

Wrapping an advice around a function to detect calls to that function is a way to prohibit the invocation of that function. To find function invocations in JS-QL, one just has to write a policy consisting of a `fCall` predicate. This predicate has to be configured to return all relevant information we need about the function call. In listing 6.23 we can see that a function call (AST) node contains fields for its `procedure` and its `arguments`. We bind these to `?proc` and `?args` respectively. We then further define an extra metavariable `?name` in the `properties` clause of the predicate, which maps to the name of the earlier defined `?proc`. Once we have the information about the function that is invoked, we can look up

its address and compare it to the address of the global `alert` (or `prompt`) function. When these are equal, the substitutions for the detected function call will be added to the results.

```
1  G.skipZeroOrMore()
2  .fCall({
3    procedure:'?proc',
4    arguments:'?args',
5    properties:{
6      '?name' : '?proc.name'
7    },
8    lookup:{'?name': '?alertAddress',
9            '_global.alert': '?alertAddress'}
10 })
```

Listing 6.23: Policy 9 in JS-QL

6.3.4 Conclusion

In this section we expressed 3 policies in the ConScript language and JS-QL. As ConScript is the only approach that checks for policy violations using dynamic analysis, we can't really compare approaches. We can however compare the expressiveness of the policies written in each language. Table 6.3 shows that we were able to express all 3 ConScript policies in the JS-QL language as well. 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 since we detect violations at compile-time, rather than at runtime. Field accesses can also be expressed as function calls (`getField` and `setField` in listing 6.20), so they can reason about getting and setting values as well. JS-QL can also reason about these things, but it has access to a lot more information thanks to the abstract state graph.

Language	Policy 7	Policy 8	Policy 9
Gatekeeper	✓	✓	✓
JS-QL	✓	✓	✓

Legend: ✓: Fully expressible

Table 6.3: Expressiveness in JS-QL and ConScript

The advice functions written in ConScript have full access to the JavaScript

language, making them very flexible in behaviour. By using JavaScript instead of a DSL, the policies themselves are also quite verbose, since for each policy a JavaScript function has to be created. This does allow them to define properties and filters, as in JS-QL. However, their approach limits them to detect only function calls, which certainly is a limitation and thus reduces expressiveness. Querying for multiple one sequential lines of code is also tricky in ConScript. Where a JS-QL policy 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 "bit". The variable will be 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 conclude that JS-QL queries are more expressive when it comes down to the detection of different kinds of program states. The language also proves flexible in terms of specifying properties and filters, but isn't 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.

6.4 Evaluation

We evaluated the JS-QL language by expressing 9 different policies originating from 3 different papers. This section evaluates the framework presented in the dissertation by discussing the advantages and limitations of the query language.

6.4.1 Advantages

A key advantage of the framework is the ability for programmers to define queries and policies as general or specific as they want. Starting from the `state` predicate, one can express complex patterns that fit their needs and wrap them in a self-named predicate. Flexibility is key in these predicates since the user himself can specify which properties he exposes through the predicate. These properties can then be queried by passing metavariables as arguments, which will later be bound when a match is found. Literals and metavariables that are already bound act as filters for the predicates, as in any declarative language. Negation can be useful when expressing actions that should not happen at a certain moment in a query. This was illustrated in section 6.2.1, where a function call needed to be detected before the end of a conditional branch. JS-QL, in contrast to many other query languages, offers this expressiveness, albeit in a limited way. The JIPDA graph contains states with information of arbitrary depth. Therefore, the framework had to provide access to these levels of information. This flexibility again opens up opportunities because we aren't bound to one particular graph type. Hy-

pothetically, all types of graphs that contain information in its edges and nodes can be used in the framework with only little to no modification of the framework itself. Only a reification layer of the new graph should be provided, mapping the states of the graph to the format our framework uses. Another non-trivial feature of the framework is the possibility to recursively define queries. This type of queries can be of special use when one wants to follow a trace of information starting at a certain point for example. A particularly interesting use for recursive queries is to trace all aliases of a certain variable. The result then shows all states in the graph where the original variable is aliased. Along with the marked nodes in the graph, a table containing all substituted metavariables is also displayed. We believe this representation of the results makes them well legible.

6.4.2 Limitations

Other approaches might modify the graph by deleting states and edges to obtain a new graph. This new graph then only consists of information they want to reason about. Our framework currently does not provide this functionality. Another feature that would amplify the expressive power of the framework would be some means to combine results of multiple queries, such as the use of logical arithmetics. Expressing the disjunction or conjunction of two queries would greatly improve the expressiveness of JS-QL. An example of this was given in subsections 6.2.1 and 6.2.3. Although negation is already supported, it only works for single `states`. We would also add to the expressiveness of the language if we were unrestricted while expressing negation. This is a topic of intended future research.

6.4.3 Conclusion

The combination of abstract state graphs and regular path expressions prove to be an effective means to obtain program information and define security policies. We validated our framework by expressing a range of different security policies in the JS-QL language and discussing the advantages and limitations of our approach.

Chapter 7

Conclusion and future work

7.1 Summary

7.2 Future work

Appendices

Appendix A

JS-QL policies and predicates

```
G.skipZeroOrMore()
.state({node:{
          expression:{
            callee : {name : 'close'},
            arguments: '?args'
        }},
        properties:{
          '?arg' : prop('memberOf','?args'),
          '?argName': '?arg.name'
        },
        lookup:{
          '?argName': '?argAddr' //Match address of the file
        }
       })
.not()
.state({node:{
          expression:{
            callee : {name : 'open'},
            arguments: '?args2'
        }},
        properties:{
          '?arg2' : prop('memberOf','?args2'),
          '?argName2': '?arg2.name'
        },
        lookup:{
          '?argName2': '?argAddr'
        }
       }).star() //Zero or more states
.state({node:{
          expression:{
            callee : {name : 'access'},
            arguments: '?args3'
        }},
```

```
34          properties:{
35            '?arg3' : prop('memberOf','?args3'),
36            '?argName3': '?arg3.name'
37          },
38          lookup:{
39            '?argName3': '?argAddr' //Match address of the file
40          }
41        })
```

Listing A.1: A query for detecting accesses to closed files

```
1  JSQL.prototype.writeToBuiltinObjectPrototype = function(obj){
2    var obj = obj || {};
3    var states = [];
4    var frozenObjects = ['Array', 'Boolean', 'Date', 'Function', '
       Document', 'Math', 'Window','String'];
5    var ret = this.lBrace();
6    var objProps = this.getTmpIfUndefined();
7    for(var i = 0; i < frozenObjects.length; i++){
8      var s = {};
9      this.setupStateChain(s, ['node', 'expression', 'left','
         properties'], objProps);
10     this.setupStateChain(s, ['node', 'expression', 'left','
         mainObjectName'], frozenObjects[i]);
11     this.setupFilter(s, 'contains', objProps, 'prototype');
12     this.finalize(s, obj);
13     states.push(s);
14   }
15   for(var j = 0; j < states.length; j++){
16     if(j !== states.length - 1){
17       ret = ret.state(states[j]).or()
18     }
19     else{
20       ret = ret.state(states[j]).rBrace();
21     }
22   }
23   return ret;
24 }
```

Listing A.2: The `assign` predicate

```
1    obj = obj || {};
2    var s1 = {};
3    var s2 = {};
4    var newObj = {};
5    var x = this.getTmpIfUndefined(obj.x);
6    var y = this.getTmpIfUndefined(obj.y);
7    var flow;
8
```

```
9    if(obj.rec){
10     flow = this.getRecVar(obj.rec);
11   }
12   else{
13     flow = this.getTmpIfUndefined();
14   }
15
16   newObj.x = flow;
17   newObj.y = y;
18   newObj.rec = obj.rec;
19
20   this.setupStateChain(s1, ['node','expression','right','name'],
         x); //alias
21   this.setupStateChain(s1, ['node','expression','left','name'],
        y); //leaked
22   this.setupStateChain(s2, ['node','expression','right','name'],
         x); //alias
23   this.setupStateChain(s2, ['node','expression','left','name'],
        flow); //leaked
24
25   return this .lBrace()
26         .state(s1) //assign from x to y
27         .or()
28         .state(s2) //assign from x to tmp
29         .skipZeroOrMore()
30         .rec(newObj,this.taintedBy)
31         .rBrace();
32
33 }
```

Listing A.3: The `taintedBy` recursive policy

```
1  JSQL.prototype.writeToBuiltinObjectPrototype = function(obj){
2    var obj = obj || {};
3    var states = [];
4    var frozenObjects = ['Array', 'Boolean', 'Date', 'Function', '
       Document', 'Math', 'Window','String'];
5    var ret = this.lBrace();
6    var objProps = this.getTmpIfUndefined();
7    for(var i = 0; i < frozenObjects.length; i++){
8      var s = {};
9      this.setupStateChain(s, ['node', 'expression', 'left','
         properties'], objProps);
10     this.setupStateChain(s, ['node', 'expression', 'left','
         mainObjectName'], frozenObjects[i]);
11     this.setupFilter(s, 'contains', objProps, 'prototype');
12     this.finalize(s, obj);
13     states.push(s);
14   }
```

```
15    for(var j = 0; j < states.length; j++){
16      if(j !== states.length - 1){
17        ret = ret.state(states[j]).or()
18      }
19      else{
20        ret = ret.state(states[j]).rBrace();
21      }
22    }
23    return ret;
24  }
```

Listing A.4: The `writeToBuiltinObjectPrototype` policy

Bibliography

[1] Serge Abiteboul et al. “The Lorel query language for semistructured data”. In: *International journal on digital libraries* 1.1 (1997), pp. 68–88.

[2] Ben Adida, Adam Barth, and Collin Jackson. “Rootkits for JavaScript Environments”. In: *Proceedings of the 3rd USENIX Conference on Offensive Technologies*. WOOT’09. Montreal, Canada: USENIX Association, 2009, pp. 4–4.

[3] Jon Bentley. “Programming Pearls: Little Languages”. In: *Commun. ACM* 29.8 (Aug. 1986), pp. 711–721. ISSN: 0001-0782.

[4] Peter Buneman et al. “A Query Language and Optimization Techniques for Unstructured Data”. In: *SIGMOD Rec.* 25.2 (June 1996), pp. 505–516. ISSN: 0163-5808. DOI: 10.1145/235968.233368. URL: http://doi.acm.org/10.1145/235968.233368.

[5] J. Craig Cleaveland. “Building Application Generators”. In: *IEEE Softw.* 5.4 (July 1988), pp. 25–33. ISSN: 0740-7459.

[6] Marco Cova, Christopher Kruegel, and Giovanni Vigna. “Detection and Analysis of Drive-by-download Attacks and Malicious JavaScript Code”. In: *Proceedings of the 19th International Conference on World Wide Web*. WWW ’10. Raleigh, North Carolina, USA: ACM, 2010, pp. 281–290. ISBN: 978-1-60558-799-8.

[7] Roger F. Crew. “ASTLOG: A Language for Examining Abstract Syntax Trees”. In: *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*. DSL’97. Santa Barbara, California: USENIX Association, 1997, pp. 18–18.

[8] H. Conrad Cunningham. “A little language for surveys: constructing an internal DSL in Ruby”. In: *Proceedings of the 46th Annual Southeast Regional Conference on XX*. ACM-SE 46. Auburn, Alabama: ACM, 2008, pp. 282–287. ISBN: 978-1-60558-105-7.

[9] Arie van Deursen, Paul Klint, and Joost Visser. “Domain-specific Languages: An Annotated Bibliography”. In: *SIGPLAN Not.* 35.6 (June 2000), pp. 26–36. ISSN: 0362-1340.

[10] Mary Fernandez et al. “A Query Language for a Web-Site Management System”. In: *SIGMOD Record* 26 (1997), pp. 4–11.

[11] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925.

[12] Martin Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321712943, 9780321712943.

[13] Debasish Ghosh. *DSLs in Action*. 1st. Greenwich, CT, USA: Manning Publications Co., 2010. ISBN: 9781935182450.

[14] Salvatore Guarnieri and Benjamin Livshits. “GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for Javascript Code”. In: *Proceedings of the 18th Conference on USENIX Security Symposium*. SSYM’09. Montreal, Canada: USENIX Association, 2009, pp. 151–168.

[15] Sebastian Günther, Maximilian Haupt, and Matthias Splieth. “Agile engineering of internal domain-specific languages with dynamic programming languages”. In: *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*. IEEE. 2010, pp. 162–168.

[16] K. A. Hawick. “Fluent Interfaces and Domain-Specific Languages for Graph Generation and Network Analysis Calculations”. In: *Proceedings of the International Conference on Software Engineering (SE’13)*. Innsbruck, Austria: IASTED, Nov. 2013, pp. 752–759.

[17] Huahai He and Ambuj K. Singh. “Graphs-at-a-time: Query Language and Access Methods for Graph Databases”. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’08. Vancouver, Canada: ACM, 2008, pp. 405–418. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376660. URL: http://doi.acm.org/10.1145/1376616.1376660.

[18] Ariya Hidayat. *ECMAScript parsing infrastructure for multipurpose analysis*. 2016. URL: http://esprima.org (visited on 04/26/2016).

[19] David Hovemeyer and William Pugh. “Finding bugs is easy”. In: *ACM Sigplan Notices* 39.12 (2004), pp. 92–106.

[20] Paul Hudak. “Building Domain-specific Embedded Languages”. In: *ACM Comput. Surv.* 28.4es (Dec. 1996). ISSN: 0360-0300.

[21] Ecma International. *ECMAScript 2015 Language Specification*. 2015. URL: http://www.ecma-international.org/ecma-262/6.0/index.html (visited on 04/16/2016).

[22] Raoul Praful Jetley, Paul L. Jones, and Paul Anderson. "Static Analysis of Medical Device Software Using CodeSonar". In: *Proceedings of the 2008 Workshop on Static Analysis*. SAW '08. Tucson, Arizona: ACM, 2008, pp. 22–29. ISBN: 978-1-59593-924-1.

[23] Andrew Johnson et al. *Exploring and Enforcing Application Security Guarantees via Program Dependence Graphs*. Tech. rep. TR-04-14. Harvard University, 2014.

[24] Andrew Johnson et al. "Exploring and Enforcing Security Guarantees via Program Dependence Graphs". In: *SIGPLAN Not.* 50.6 (June 2015), pp. 291–302. ISSN: 0362-1340. DOI: 10.1145/2813885.2737957. URL: http://doi.acm.org/10.1145/2813885.2737957.

[25] Gabor Karsai et al. "Design guidelines for domain specific languages". In: *arXiv preprint arXiv:1409.2378* (2014).

[26] Monica S. Lam et al. "Context-sensitive Program Analysis As Database Queries". In: *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '05. Baltimore, Maryland: ACM, 2005, pp. 1–12. ISBN: 1-59593-062-0.

[27] Yanhong A. Liu et al. "Parametric Regular Path Queries". In: *SIGPLAN Not.* 39.6 (June 2004), pp. 219–230. ISSN: 0362-1340.

[28] Michael Martin, Benjamin Livshits, and Monica S. Lam. "Finding Application Errors and Security Flaws Using PQL: A Program Query Language". In: *SIGPLAN Not.* 40.10 (Oct. 2005), pp. 365–383. ISSN: 0362-1340.

[29] Leo A. Meyerovich and Benjamin 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 '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 481–496. ISBN: 978-0-7695-4035-1.

[30] Jens Nicolay et al. "Detecting function purity in javascript". In: *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*. IEEE. 2015, pp. 101–110.

[31] Addy Osmani. *Essential JavaScript Namespacing Patterns*. 2011. URL: https://addyosmani.com/blog/essential-js-namespacing (visited on 04/20/2016).

[32] Pablo Martín Pérez, Joanna Filipiak, and José María Sierra. "LAPSE+ static analysis security software: Vulnerabilities detection in java EE applications". In: *Future Information Technology*. Springer, 2011, pp. 148–156.

[33] Thomas W. Reps. "Applications of Logic Databases". In: ed. by Raghu Ramakrishnan. Boston, MA: Springer US, 1995. Chap. Demand Interprocedural Program Analysis Using Logic Databases, pp. 163–196. ISBN: 978-1-4615-2207-2.

[34] Marko A. Rodriguez. "The Gremlin Graph Traversal Machine and Language (Invited Talk)". In: *Proceedings of the 15th Symposium on Database Programming Languages*. DBPL 2015. Pittsburgh, PA, USA: ACM, 2015, pp. 1–10. ISBN: 978-1-4503-3902-5. DOI: 10.1145/2815072.2815073. URL: http://doi.acm.org/10.1145/2815072.2815073.

[35] Mark A. Simos. "Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle". In: *Proceedings of the 1995 Symposium on Software Reusability*. SSR '95. Seattle, Washington, USA: ACM, 1995, pp. 196–205. ISBN: 0-89791-739-1. DOI: 10.1145/211782.211845. URL: http://doi.acm.org/10.1145/211782.211845.

[36] Gregor Snelting et al. "Checking Probabilistic Noninterference Using JOANA". In: *it - Information Technology* 56 (Nov. 2014), pp. 280–287. DOI: 10.1515/itit-2014-1051.

[37] Ken Thompson. "Programming Techniques: Regular Expression Search Algorithm". In: *Commun. ACM* 11.6 (June 1968), pp. 419–422. ISSN: 0001-0782. DOI: 10.1145/363347.363387. URL: http://doi.acm.org/10.1145/363347.363387.

[38] Joost Visser et al. *Building Maintainable Software, Java Edition*. 1st. O'Reilly Media, 2016. ISBN: 1491953527, 9781491953525.

[39] John Whaley et al. "Using Datalog with Binary Decision Diagrams for Program Analysis". In: *Proceedings of the Third Asian Conference on Programming Languages and Systems*. APLAS'05. Tsukuba, Japan: Springer-Verlag, 2005, pp. 97–118. ISBN: 3-540-29735-9, 978-3-540-29735-2.

[40] David A Wheeler. *Flawfinder*. 2011.