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

TODO inleidende tekst

2.1 Introduction to static analysis

2.2 Support for generic vulnerabilities

2.3 Support for application-specific vulnerabilities

2.4 Conclusion

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[11][7]. However, other approaches exist that lean more closely towards the implementation of our system. As discussed in section 2.1, 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 [13]. 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 [8], which enables us to query flow graphs directly. The internals of this algorithm will be discussed in greater detail in section 4.4.

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.

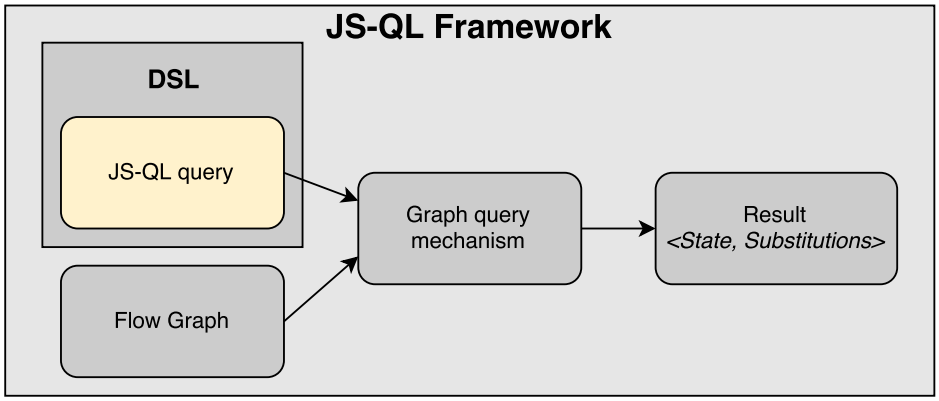

Figure 3.1: JS-QL framework architecture

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[3] 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[] abstract state graph 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

The abstract state graph is an alternation of four different types of states. JIPDA internally uses Esprima[5] 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. 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).

1. *Evaluation state*: Represents the evaluation of an expression or statement in the program.
2. *Continuation state*: ...
3. *Return state*: Indicates the return of a function application, having the returned value stored in one of its properties.
4. *Result state*: Final state of the graph, indicating the abstract final value(s) of the program. The graph can have more than one result state, depending on the program's nature.

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

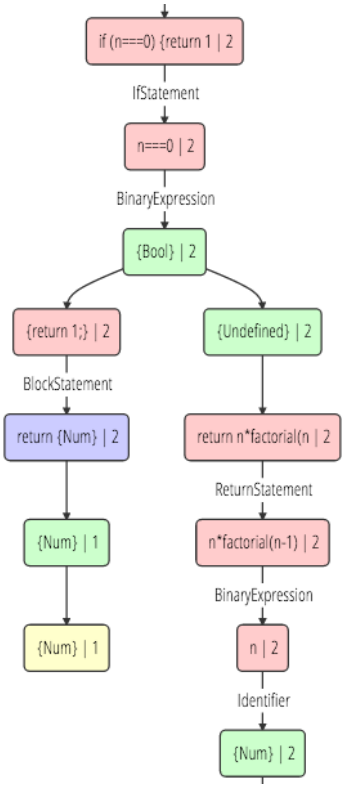

Figure 3.2: Example JIPDA abstract state graph

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.

```
1  {
2      "type": "Program",
3      "body": [
4          {
5              "type": "FunctionDeclaration",
6              "id": {
7                  "type": "Identifier",
8                  "name": "answerToTheUniverse"
```

```
 9              },
10              "params": [
11                  {
12                      "type": "Identifier",
13                      "name": "arg"
14                  }
15              ],
16              "defaults": [],
17              "body": {
18                  "type": "BlockStatement",
19                  "body": [
20                      {
21                          "type": "ReturnStatement",
22                          "argument": {
23                              "type": "Literal",
24                              "value": 42,
25                              "raw": "42"
26                          }
27                      }
28                  ]
29              },
30              "generator": false,
31              "expression": false
32          }
33      ]
34  }
```

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* $\hat{\beta}$. Each binding maps to a value through the *store* $\hat{\sigma}$. 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 $\hat{\nu}$ is to locate its binding in $\hat{\beta}$. Next, the value of the variable can be looked up in the store by composing these two functions. The value of $\hat{\nu}$ is

given by $\hat{\sigma}(\hat{\beta}(\hat{\nu}))$. 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   //β̂ contains a binding x → Âddr
3   var x = 3;
4
5   //σ̂ has an entry Âddr → V̂al
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. 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.

Application context

Some text about the application context.

Stack

Some text about the stack.

3.3 Internal and external DSLs for querying graphs

3.4 Design of an internal DSL for querying flow graphs

Chapter 4

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

4.1 Syntax and structure

4.2 Types of queries

4.3 Defining policies

4.4 The matching engine

Chapter 5

Implementation

5.1 Used technologies

5.2 Design of the query language

5.3 Design of the matching engine

5.4 Conclusion

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 [4].

6.1.1 Writes to prototype objects

Many websites use bookmarklets to store user information to automate the login process, for example [1]. 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.1 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 [10]. 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*[2]. 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 [12]). 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 [6].

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 [6] 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[3], 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 [9].

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

Compound Policies

```
1  RegularPathExpression.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.1: The `writeToBuiltinObjectPrototype` predicate

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

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

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

[7] 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.

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

[9] 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.

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

[11] 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.

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

[13] 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.