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

1 Introduction **5**
- 1.1 Context . . . 5
- 1.2 Motivation . . . 5
- 1.3 Objective . . . 5
- 1.4 Overview . . . 5

2 Background **6**
- 2.1 Program representations and querying mechanisms . . . 6
  - 2.1.1 Exploring and Enforcing Security Guarantees via Program Dependence Graphs . . . 6
  - 2.1.2 GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code . . . 6
  - 2.1.3 Parametric regular path queries . . . 6
- 2.2 Expressing policies using a domain-specific language . . . 6
  - 2.2.1 Fluent Interfaces to a Java-Based Internal Domain-Specific Languages for Graph Generation and Analysis . . . 7
  - 2.2.2 A Little Language for Surveys: Constructing an Internal DSL in Ruby . . . 8
  - 2.2.3 Dagoba: an in-memory graph database . . . 10
- 2.3 Conclusion . . . 10

3 Context **11**
- 3.1 Static analysis . . . 11
- 3.2 Conclusion . . . 11

4 The JS-QL query language **12**
- 4.1 The query language . . . 12
  - 4.1.1 Motivation . . . 12
  - 4.1.2 Syntax and structure . . . 12
  - 4.1.3 Defining policies . . . 12
- 4.2 Conclusion . . . 12

5 The query engine **13**
5.1 Architecture 13
5.2 Types of queries 13
5.3 Recursion 13
5.4 Conclusion 13

6 Implementation **14**
6.1 Used technologies 14
6.2 Design of the query system 14
6.3 Conclusion 14

7 Evaluation **15**
7.1 the GateKeeper language 15
7.1.1 Writes to prototype objects 15
7.1.2 Script inclusions 17
7.1.3 Global namespace pollution 19
7.1.4 Conclusion 19
7.2 The PidginQL language 19
7.2.1 Only CMS administrators can send a message to all CMS users 20
7.2.2 A database is opened only after the master password is checked or when creating a new database 22
7.2.3 Public outputs do not depend on a users's password, unless it has been cryptographically hashed 22
7.2.4 conclusion 22
7.3 The Conscript language 22
7.3.1 No string arguments to setInterval, setTimeout 22
7.3.2 HTTP-cookies only 23
7.3.3 Prevent resource abuse 23
7.4 Evaluation 23

8 Conclusion and future work **24**
8.1 Summary 24
8.2 Future work 24

Appendices **25**

A Compound Policies **26**

Chapter 1

Introduction

1.1 Context

1.2 Motivation

1.3 Objective

1.4 Overview

Chapter 2

Background

2.1 Program representations and querying mechanisms

Programs can be represented in several ways. These representations can then be queried to detect all kinds of information, such as control- and data-flow properties. In this section, three program representation approaches are presented, together some means to query those representations.

2.1.1 Exploring and Enforcing Security Guarantees via Program Dependence Graphs

2.1.2 GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code

2.1.3 Parametric regular path queries

2.2 Expressing policies using a domain-specific language

This section describes three internal domain-specific languages (*DSL*s). We present one DSL written in Java, a statically typed language, and two DSLs written in dynamically typed languages, namely Ruby and JavaScript.

2.2.1 Fluent Interfaces to a Java-Based Internal Domain-Specific Languages for Graph Generation and Analysis

Many complex systems problems manifest themselves as networks. Reasoning about these networks can be hard to do manually and asks for complex algorithms to perform sometimes even simple calculations. Hawick[7] pleads for the use of some sort of abstraction to perform graph generation and analysis, more specifically the use of an internal domain-specific language. He presents a DSL built using fluent interface techniques and the statically typed Java programming language. Common data structures and repetitive computations often offer an opportunity to abstract over them, as is the case for models based on networks and graphs.

The goal of this graph DSL is to be able to compare individual network sets to detect chatacteristic signature properties. The approach is powerful because a major set of data structures and operations on graphs can be abstracted into a library framework, which can then be used by domain experts.

The first step in setting up the DSL is to set up the common data structures. For the graph DSL, there are three: `Nodes`, `Arcs` and of course the `Graphs` themselves. The fields of these data structures are divided into several categories, as depicted in figure 2.1. Structural fields hold the main graph structure, whereas auxiliaries just exist to facilitate computations. Convenience fields contain information that might come in handy, but isn't necessarily used for computations. Finally, decorative fields just are there to have some means of presenting the data in a clear, distinguishable way.

```
class Node{
  // Structural:
  List<Arc> inputs  = new Vector<>();
  List<Arc> outputs = new Vector<>();

  // Convenience:
  List<Node> dsts = new Vector<>();
  List<Node> srcs = new Vector<>();

  // Decorative:
  int index     = 0;
  int mark      = 0;
  double weight = 1.0;
  String label  = "";

  // Computation Auxiliaries:
  int component   = 0;
  int betweenness = 0;
  int count       = 0;
  boolean visited = false;
  boolean blocked = false;
}
```

Figure 2.1: The 'Node' data structure

Since we now have all information to perform most of the complex computations, the fluent interface can be set up. The approach used here implements the (Java) method chaining technique. This enables the cascading of methods by making each method in the fluent interface return the reference to itself, namely the *this* reference.

Most internal DSL's can be used as a standalone language, as seen in figure 2.2, but the paper also gives some examples in which the DSL is used inside the host language, such as the repetitive removal of the most stressed node in the network to investigate network robustness.

```
public static void main( String args[] ){

  Graph g = Graph.New( args )
  .setLogging(true)
  .report()
  .removeLeaves()
  .computeDegrees()
  .computeClusteringCoefficient()
  .computeAdjacency()
  .computeComponents()
  .computePaths()
  .computeBetweenness()
  .computeDistances()
  .computeCircuits()
  .report()
  .write( "composite.graph" )
  ;

}
```

Figure 2.2: Example use of the internal DSL

2.2.2 A Little Language for Surveys: Constructing an Internal DSL in Ruby

A Little Language for Surveys [3] 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[2] and the support for blocks[5]. Figure 2.3 shows how function calls 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[8]). It also shows how entire blocks can be attached to method calls. These blocks are passed unevaluated to the called method, enabling *deferred evaluation*.

```
question "What is your gender?" do
  response "Female" { @female = true }
  response "Male" { @female = false }
  action { @male = if @female then false
                              else true end }
end
```

Figure 2.3: Ruby method call syntax

A handy feature in programming languages is reflexive metaprogramming. The survey DSL makes use of the following Ruby reflexive metaprogramming facilities:

obj.instance_eval(str) takes a string `str` and executes it as Ruby code in the context of `obj`. This method allows internal DSL code from a string or file to be executed by the Ruby interpreter.

mod.class_eval(str) takes a string `str` and executes it as Ruby code in the context of module `mod`. This enables new methods and classes to be declared dynamically in the running program.

obj.method_missing(sym, *args) is invoked when there is an attempt to call an undefined method with the name `sym` and argument list `args` on the object `obj`. This enables the object to take appropriate remedial action.

obj.send(sym, *args) calls method `sym` on object `obj` with argument list `args`. In Ruby terminology, this sends a message to the object.

The design of the survey language is fairly simple. A survey consists of a title, some questions, some responses and finally a result. Each of these actions have a corresponding method in the DSL.

The developers of the survey language chose to split up the parsing and interpretation logic, following the *two-pass architecture*. The first-pass layer parses the file (which is read using `instance_eval`) and generates an abstract syntax tree. These parser classes are structured according to the *Object Scoping* pattern[4], using an approach called *sandboxing*. This architecture is depicted in figure 2.4. The `SurveyBuilder` class evaluates the statements it reads from the input file using it's superclass' methods. This evaluation parses the input file and builds the AST, which is stored in the superclass as well. In this way, the object scoping pattern is applied: All calls are directed to a single object (the superclass), and global namespace cluttering is avoided. When creating the AST nodes, all blocks that were passed to the method calls inside the top-level call are stored in the AST nodes, instead of being evaluated directly. These blocks will only be evaluated in the second-pass phase (hence *deferred evaluation*). This is illustrated in figure 2.3, where the block passed to `question` is evaluated in the first-pass layer, and the blocks passed to `response` and `action` are stored in the AST, ready to be evaluated in the second-pass layer. Note that sandboxing occurs by calling `instance_eval` inside the `SurveyBuilder` object. In this way, harm can only be done *inside* this object.

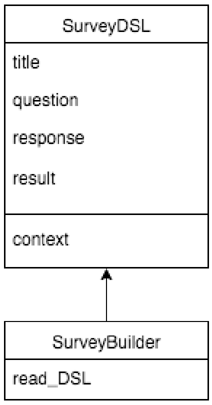

Figure 2.4: Ruby method call syntax

The actual interpretation of the created AST happens in the second-pass layer. This is a design decision which allows for different interpretation layers to be plugged in/swapped on-the-fly. However, for the survey language a simple *visitor* pattern implementation suffices to process the AST. Every AST node must provide an `accept` method which takes a `SurveyVisitor` as an argument. This is the only condition that has to be met by the interpretation layer. In this concrete example, there could be a `SurveyConsoleVisitor` and a `SurveyGUIVisitor` class, each representing the survey in their own specific way.

2.2.3 Dagoba: an in-memory graph database

2.3 Conclusion

Chapter 3

Context

3.1 Static analysis

3.2 Conclusion

Chapter 4

The JS-QL query language

4.1 The query language

4.1.1 Motivation

4.1.2 Syntax and structure

4.1.3 Defining policies

4.2 Conclusion

Chapter 5

The query engine

5.1 Architecture

5.2 Types of queries

5.3 Recursion

5.4 Conclusion

Chapter 6

Implementation

6.1 Used technologies

6.2 Design of the query system

6.3 Conclusion

Chapter 7

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, in our own query language. We will then compare these policies in terms of expressiveness and flexibility. The concept and approach for creating a new, domain-specific language for security policies is explained in chapter 4. Chapter 5 discusses the underlying query engine and how it works together with the query language to process the application-specific policies. In chapter 6 we explain how our approach was instantiated.

We start this chapter by expressing 9 security policies distilled from 3 papers in sections 7.2, 7.1 and 7.3 respectively. Every JS-QL policy will be evaluated by comparing how well it matches the policy expressed in the original paper. Finally, in section 7.4, we evaluate the query framework by specifying its advantages, disadvantages and limitations. We will also briefly compare the query languages presented in this chapter in terms of expressiveness, verbosity and conciseness (*LOC*).

7.1 the GateKeeper language

In this chapter we attempt to express 3 policies originally presented in [6].

7.1.1 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 7.1. This function can be configured to always return a ”good” location. In this way, the login function will be called in the environment of a malicious website, possibly leaking sensitive information.

```
String.prototype.toString = function(){
    //Always return "spoofed" url
    return "www.goodwebsite.com";
}

var login = function(){
  if(document.location.toString() === "www.goodwebsite.com"){
    //leak information on untrusted website
  }
}
```

Listing 7.1: Prototype poisoning example

Gatekeeper expresses policies by defining a set of rules in datalog. In order to detect writes to prototypes of frozen objects, they define the `FrozenViolation(v)` predicade, as shown in listing 7.2. 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.

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

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

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

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

Listing 7.2: 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 7.3 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 7.3: 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 7.3, with the only difference in the `mainObjectName` property, which corresponds to a different builtin object name.

7.1.2 Script inclusions

A well known exploit in JavaScript environments is *heap spraying*. 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 **??** gives an example of malicious script inclusions.

```
var evilScript;
var scripts = ["<script>bad1</script>","<script>bad2</script>"];

for(var i = 0; i < scripts.length; i++){
  evilScript = scripts[i];
  document.write(evilScript); //violation
}

var o = {};
o.f = document.writeln;
o.f("<script>bad3</script>"); //Violation
```

Listing 7.4: 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 7.5 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.

```
DocumentWrite(i) :- GlobalSym("document",d),
                    HeapPtsTo(d,"write",m),
                    Calls(i,m).

DocumentWrite(i) :- GlobalSym("document",d),
                    HeapPtsTo(d,"writeln",m),
                    Calls(i,m).
```

Listing 7.5: Policy 2 in GateKeeper

JS-QL also proves to be suitable to express such a policy. 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 node 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.

```
G.skipZeroOrMore()
.lBrace()
.fCall({
  name: '?name',
  lookup:{
    '?name'    : '?addr',
    '_global.document.write': '?addr',
    }
})
.or()
.fCall({
  name: '?name',
  lookup:{
    '?name'    : '?addr',
    '_global.document.writeln': '?addr',
    }
})
.rBrace()
```

Listing 7.6: Policy 2 in JS-QL

7.1.3 Global namespace pollution

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

Listing 7.7: Policy 3 in GateKeeper

7.1.4 Conclusion

7.2 The PidginQL language

In this chapter we attempt to express 3 policies originally presented in [9].

7.2.1 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 listed in [9] that addresses this issue can be found in listing 7.8. 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.

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

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

Listing 7.8: 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 the `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 `if-statement`), 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. However, for values with the value of $\{Bool\}$, we cannot decide on which branch is the true-branch and which one is the false-branch. We can solve this in two ways: We can assume that the condition in the conditional is a direct call to `isCMSAdmin`, which enables us to find the false-branch. From there on we can search for all calls to `addNotice` to find violations. The JS-QL policy for this case is defined in listing 7.9. We skip all states untill we reach the beginning of a false branch of a conditional. We bind the condition *test* to the metavariable *?cond*, the context *kont* to *?kont* and the stack *lkont* to *?lkont*. We further restrict condition *?cond* to contain the 'callee' property, of which we take the name and match it to the 'isCMSAdmin' literal. Next, we skip some states until we find a call to the `addNotice` method. Since we only want to detect these calls within the false-branch, we end the policy with an `endIf` predicate with matching stack and context metavariables.

Another option is to find all calls to the `addNotice` method that follow a return of `isCMSAdmin`. Since we only know that the return value of `isCMSAdmin` returns a value of $\{Bool\}$, we are unable to rule out any of the branching options. This will result in false-positives. Listing 7.9 gives an implementation of the policy. We again match the stack and context to metavariables *?lkont* and *?kont*, but this time to indicate the start of a function application. Next we specify that we want to find all return statements within that function application. This is done by indicating that these return statements must follow a node which is not the end of the function application, parametrized with the same metavariables for stack and context. Finally, some states can be skipped before finding a function call to `addNotice`.

```
//First solution
G.skipZeroOrMore()
.beginIfFalse({test: '?cond', kont: '?kont', lkont:'?lkont',
               properties:{
                 'isCMSAdmin' : '?cond.callee.name'
               }})
.skipZeroOrMore()
.fCall({name:'addNotice'})
.skipZeroOrMore()
.endIf({kont: '?kont', lkont:'?lkont'})

//Second solution
G.skipZeroOrMore()
.beginApply({name:'isCMSAdmin',kont:'?kont',lkont:'?lkont'})
.not()
    .endApply({ kont:'?kont',lkont:'?lkont'})
    .star()
.returnStatement()
.skipZeroOrMore()
.fCall({name:'addNotice'})
```

Listing 7.9: Policy 4 in JS-QL

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

```
TODO
```

Listing 7.10: Policy 5 in PidginQL

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

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

Listing 7.11: Policy 6 in PidginQL

7.2.4 conclusion

7.3 The Conscript language

7.3.1 No string arguments to setInterval, setTimeout

```
let onlyFnc : K x U x U -> K =
function (setWhen : K, fn : U, time : U) {
    if ((typeof fn) != "function") {
        curse();
        throw "The time API requires functions as inputs.";
    } else {
        return setWhen(fn, time);
    }
};
around(setInterval, onlyFnc);
around(setTimeout, onlyFnc);
```

Listing 7.12: Policy 7 in ConScript

7.3.2 HTTP-cookies only

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

Listing 7.13: Policy 8 in ConScript

7.3.3 Prevent resource abuse

```
let err : K -> K = function () {
    curse();
    throw 'err';
});
around(prompt, err);
around(alert, err);
```

Listing 7.14: Policy 9 in ConScript

7.4 Evaluation

Language	Prop1	Prop2	Prop3
JS-QL			
GateKeeper			
PidginQL			
Conscript			

Chapter 8

Conclusion and future work

8.1 Summary

8.2 Future work

Appendices

Appendix A

Compound Policies

```
RegularPathExpression.prototype.writeToBuiltinObjectPrototype =
    function(obj){
  var obj = obj || {};
  var states = [];
  var frozenObjects = ['Array', 'Boolean', 'Date', 'Function', '
      Document', 'Math', 'Window','String'];
  var ret = this.lBrace();
  var objProps = this.getTmpIfUndefined();
  for(var i = 0; i < frozenObjects.length; i++){
    var s = {};
    this.setupStateChain(s, ['node', 'expression', 'left','
        properties'], objProps);
    this.setupStateChain(s, ['node', 'expression', 'left','
        mainObjectName'], frozenObjects[i]);
    this.setupFilter(s, 'contains', objProps, 'prototype');
    this.finalize(s, obj);
    states.push(s);
  }
  for(var j = 0; j < states.length; j++){
    if(j !== states.length - 1){
      ret = ret.state(states[j]).or()
    }
    else{
      ret = ret.state(states[j]).rBrace();
    }
  }
  return ret;
}
```

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. URL: http://dl.acm.org/citation.cfm?id=1855876.1855880.

[2] Jamis Buck. *Writing Domain Specific Languages*. 2006. URL: http://weblog.jamisbuck.org/2006/4/20/writing-domain-specific-languages.html (visited on 04/16/2016).

[3] 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. DOI: 10.1145/1593105.1593181. URL: http://dx.doi.org/10.1145/1593105.1593181.

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

[5] Jim Freeze. *Creating DSLs with Ruby*. 2006. URL: http://www.artima.com/rubycs/articles/ruby_as_dsl2.html (visited on 04/16/2016).

[6] 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. URL: http://dl.acm.org/citation.cfm?id=1855768.1855778.

[7] 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.

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

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