Spring 5-20-2016

Secure Declassification in Faceted JavaScript

Tam Wing
San Jose State University

Secure Declassification in Faceted JavaScript

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Wing Cheong Tam

May 2016

The Designated Project Committee Approves the Project Titled

Secure Declassification in Faceted JavaScript

by

Wing Cheong Tam

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2016

Dr. Thomas Austin	Department of Computer Science
Dr. Robert Chun	Department of Computer Science
Dr. Frank Butt	Department of Computer Science

ABSTRACT

Information leaks currently represent a major security vulnerability. Malicious code, when injected into a trusted environment and executed in the context of the victim's privileges, often results in the loss of sensitive information. To address this security issue, this paper focuses on the idea of information flow control using faceted execution [3]. This mechanism allows the interpreter to efficiently keep track of variables across multiple security levels, achieving termination-insensitive non-interference (TINI). With TINI, a program can only leak one bit of data, caused by the termination of a program. One key benefit of having faceted execution is that flow policy can be enforced automatically on the basis of its architecture, rather than relying on filtering, validation, and encoding, over user inputs.

Despite the fact that information flow control ensures strong confidentiality, such a model is too restrictive for many real-world applications. Declassification offers one way of releasing sensitive information in a controlled manner. This paper introduces Faceted JS, a modified JavaScript language that supports basic JavaScript features as well as faceted executions. To demonstrate the proper way to release sensitive data, a declassification mechanism is implemented, based on the concept of the object capability model [12] and policy-agnostic programming [4]. Finally, we cover the aspect of implementation and offer some practical examples.

ACKNOWLEDGMENTS

I would like to express my deep appreciation to my thesis advisor, Dr. Thomas Austin. With his in-depth knowledge of information security, information flow control and operational semantics, he patiently provided me with valuable guidance, ongoing encouragement and support throughout the past year.

I would also like to thank my other committee members, Dr. Robert Chun and Dr. Frank Butt, for monitoring the progress of the project, contributing their expert feedback.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Current Security Challenges

With the growing importance of web technology, information sharing has become faster and more convenient. Although it brings us many benefits, it can also create many security problems. For example, a massive data breach in June 2015 compromised the sensitive information of more than 21 million federal employees, prompting the government to propose a 19 billion USD cyber security plan to defend against such attacks [11]. This illustrates the special importance of information security in environments such as government networks, online banking, and e-commerce. Due to the flexibility of modern computer systems, untrusted data, when injected into a trusted environment, can be executed in the context of the victim's privileges. Indeed, current web technology relies on the developers to validate and encode untrusted input before placing it into a trusted context. It is rather difficult to cover the vast majority of use cases in a large system. According to a 2016 study, about 87% of open-source vulnerabilities are comprised of XSS (cross-site scripting) and SQL injection[15]. As a result, security vulnerabilities are still very common.

Because of this security vulnerability, an attacker can easily inject malicious code into a web application, thereby obtaining sensitive information (such as passwords, credit card numbers, and personal information). Similarly, an attacker can also insert into an insecure system SQL queries that are later executed by the database as those of an authorized user. SQL injection represents a threat to data-driven applications, as it allows attackers to manipulate the database by executing unauthenticated SQL commands.

1.2 Information Flow Control

It has been demonstrated that information flow control is a promising solution to these problems [6, 3, 8]. On this basis, several information flow techniques have been proposed [6]. Unfortunately, most mechanisms rely on static information flow analysis [6], which is ill-suited to dynamic scripting language and provides no security guarantees on information propagation. To overcome these limitations, recent research has focused on dynamic information flow control, a concept that enforces security policies in the runtime phase. Taking inspiration from faceted execution [3], we introduce here a modified JavaScript language, Faceted JS. For demonstration purposes, we also implement a lightweight JavaScript interpreter. In addition to basic JavaScript features, this interpreter supports first-class security labels, output channels, release policies, and functions for classification and declassification of data.

1.3 Declassification

Real-world applications often involve the release of sensitive information. For instance, a login application has to inform users about the correctness of the password. Unfortunately, an attacker may exploit this mechanism by causing the system to release more data than necessary for the intended purpose. Determining whether it is safe to release sensitive data and to formalize these rules represents a substantial challenge [4]. Various approaches have been proposed in recent research in order to overcome this challenge. One interesting idea is formalization of the information release into four different dimensions [14]. These dimensions are:

- what information is released
- who releases the information
- where the information is released

- when information can be released

Although handling declassification might seem to be intuitive, it is challenging to do this correctly. Taking inspiration from the information release metric [14] and policy-agnostic programming [4], our solution uses object-capability techniques [12] in the creation of security labels. This provides strong confidentiality and integrity guarantees without sacrificing the flexibility of modern computer systems. Finally, this thesis also explores different approaches for safely handling these elements of declassification.

1.4 Thesis Overview

This paper has two focuses, information flow control using faceted values [6, 3] and a declassification mechanism using techniques from the object-capability model. In Chapter 2, we describe previous research on information flow control. We also present an information flow control model that allows classification and declassification of faceted values. In Chapters 3 and 4, we discuss our modified JavaScript language, followed by the syntax and evaluation rules using operational semantics. Chapter 5 explains the design of the JavaScript interpreter in terms of its architecture and implementation. In Chapter 6, we analyze runtime performance and demonstrate practical use cases. We also explain the way to utilize the concept for real world applications. In Chapter 7, in addition to offering a concise conclusion, we describe some possibilities for future work on information flow control.

CHAPTER 2

Background

Many applications involve sensitive information such as credit card numbers, social security numbers, passwords, and personal information. Due to the growing number of Internet applications, securing these data is now a huge security challenge. The concept of information flow control refers to securing sensitive information by enforcing information flow policies within the architecture, preventing data flow from a high-security level (H) into a low-security level (L). In this paper, `h` denotes a high-confidentiality variable, whereas `l` denotes a low-confidentiality variable.

This paper considers two types of information flow: explicit flow and implicit flow. The following situation represents a simple explicit flow of a secret from a high-security level to a low-security level, through direct assignment.

```
1 // suppose h is secret
2 var l = h;
```

The following example represents implicit flow, whereby the secret does not flow into `l` explicitly. Instead, we can still determine information about the secret, since `l` depends on the conditional statement `h === l`.

```
1 // suppose h is secret
2 var l = false;
3 if (h === l)
4     l = true;
```

2.1 Information Flow Control

Prior works on information flow control have focused mainly on static information flow analysis. While static flow analysis plays an important role in information flow control, it is a poor fit for dynamically typed JavaScript.

2.1.1 Static Information Flow Analysis

Static information flow analysis approaches [6] work by analyzing the source code during compile time and rejecting programs that violate flow policies. The certification process works by deducing a relationship between information flow and variables. Although the approach is proven to be effective and to minimize the need for run-time checking, it is too restrictive for dynamically typed JavaScript.

2.1.2 Dynamic Information Flow Control

To overcome these limitations, many discussions focus on the concept of dynamic information flow control [1, 13]. This type of mechanism enforces flow policies during runtime, preventing leaks from unexpected implicit flows. In contrast to a static type system, dynamic analyses are often slower in terms of performance, while still providing security guarantees such as non-interference, a property that ensures that a low-security output does not depend on any high-security inputs. For instance, the no-sensitive-upgrade (NSU) strategy [1, 16] rejects all updates of public variables where execution depends on private data as in the implicit flow example discussed previously. The permissive-upgrade (PU) strategy [2] allows implicit flows of private data, but marks the data as partially leaked; if partially leaked data are used in an unsafe manner, execution will halt.

2.1.3 Secure Multi-Execution

Most dynamic flow control systems rely on stuck evaluations, which halt the execution of any possible information leak. The consequence is that valid programs might also be rejected in such a system. For this reason, more comprehensive designs, such as secure multi-execution, were introduced [8]. Using secure multi-execution, a program is split into multiple copies, whereby each of them is associated with a particular security level. Each copy is then executed independently, hence providing the non-interference property without stuck computations. Consider the following example from secure multi execution [8]:

```
1 var text = document.getElementById("email-input").text;
2 var abc = 0;
3 if(text.indexOf("abc") != -1) {
4     abc = 1
5 };
6 var url = "http://example.com/img.jpg?t=" + escape(text) + abc;
7 document.getElementById("banner-img").src = url;
```

During the runtime, the JavaScript program is split into a high-security level version and a low-security level version. Suppose `document.getElementById("email-input").text` is a high security input and `document.getElementById("banner-img").src` loads an image from a untrusted channel (low security level). High viewers can see the actual email address, whereas low viewers see only the `undefined` value.

Execution at the low-security level:

```
1 var text = undefined;
2 var abc = 0;
3 if(text.indexOf("abc") != -1) {
4     abc = 1
5 };
```

```
6 var url = "http://example.com/img.jpg?t=" + escape(text) + abc;
7 document.getElementById("banner-img").src = url;
```

Execution at the high-security level:

```
1 var text = document.getElementById("email-input").text;
2 var abc = 0;
3 if(text.indexOf("abc") != -1) {
4     abc = 1
5 };
6 var url = "http://example.com/img.jpg?t=" + escape(text) + abc;
```

2.1.4 Faceted Execution

Faceted execution simulates secure multi-execution through the use of special faceted values. A faceted value consists of a security label `k`, private data `H`, and public data `L`. The following expression represents a typical faceted value wherein a public observer sees only the low facet V_l, whereas a private observer sees only the high facet V_h.

$$\langle k \ ? \ V_h \ : \ V_l \rangle$$

Suppose h is the sensitive boolean value, consider the implicit flow example caused by a conditional assignment:

```
1 var h = ... // some secret
2 var l = false;
3 if (h){
4     l = true;
5 }
```

Given `h = true`, the resulting value of `l` is a faceted value $\langle k \ ? \ true \ : \ false \rangle$. Intuitively, `l = false` if `h = false`, as neither the high facet nor the low facet execute the assignment statement `l = true`.

Faceted values might also be nested and consist of multiple security labels. Consider a medical record in which only the medical department and the government have the permission. The value can be represented by associating a nested faceted value with two security labels. A dummy default value is represented by $\perp$. For example:

$$\langle k_{\text{gov}} \texttt{ ? } \langle k_{\text{med}} \texttt{ ? V : } \perp \rangle \texttt{ : } \perp \rangle$$

Faceted values allow a single value to appear as multiple values simultaneously, where high viewers can access the secret value, while the low viewer sees only dummy default values. Instead of executing multiple highly redundant programs, this mechanism allows the interpreter to simultaneously keep track of the variables across different security levels, while the mechanism achieves termination-insensitive non-interference with minimal overhead.

Consider the same example from secure multi execution [8]. Suppose the user input is abc@example.com. The line comment describes the state after each assignment/conditional statement.

```
1  var text = document.getElementById("email-input").text;
2  // text = < k ? "abc@example.com" : undefined >
3  var abc = 0;
4  // abc = 0
5
6  if(text.indexOf("abc") != -1) {
7      abc = 1;
8      // abc = < k ? 1 : 0 >
9  };
10
11 var url = "http://example.com/img.jpg?t=" + escape(text) + abc;
12 // url = < k ? "http://example.com/img.jpg?t=abc@example.com1"
13 //            : "http://example.com/img.jpg?t=undefined0" >
```

```
14
15 document.getElementById("banner-img").src = url;
16 // Dom object : <img src="http://example.com/img.jpg?t=undefined0" />
```

Instead of executing two copies of the program, faceted execution maintains branches in variables. In this example, the variable `text` is initialized to a faceted value $\langle k$ `?` "*abc@example.com*" `:` $undefined\rangle$ and the variable `abc` is initialized to `0`. A private observer sees the condition `text.indexOf("abc") != -1` as `true`, while a public observer sees the same condition as `false`. Consequently, the assignment `abc = 1` updates only the private facet of `abc`, resulting in a faceted value $\langle k$ `? 1 : 0`$\rangle$. Finally, since `example.com` is not considered to be a trusted output channel, the dom element `banner-img` is set to `http://example.com/img.jpg?t=undefined0`, which prevents the email address from being leaked to the public.

2.1.5 Policy-Agnostic Programming

Using only security labels and faceted values is not sufficient for expressing confidentiality and integrity policies. Furthermore, it is also hard to justify and enforce these policies across the entire program. A more comprehensive solution uses faceted execution for policy-agnostic programming [4]. This approach allows programmers to implement core functionality independently of the information flow policies and to enforce these policy rules automatically throughout the program.

The following policy simply restricts data access to the high-security clearance user `admin`, while the `restrict` statement limits the possible output of `secret` to any channels where the username is equal to "admin".

```
1 let secret: string = label a in
```

```
    restrict a: f(c: User).(c == admin) in
    < a ? "some admin-level secret" : 0 >
in ...
```

2.2 Object Capabilities

Most modern computer systems rely on the access control list (ACL) to restrict access to sensitive data and operations. These systems validate the user's request based on the access control matrix during runtime. They grant or deny the access, based on the ambient identity of the user. This property often leads to the confused deputy problem, whereby a public user can trick the system into executing scripts on behalf of a high-security level program, accessing unauthorized resources.

Capability-based systems originally presented by Dennis and Van Horn [7], use unforgeable references to describe transferable rights on different object models. A reference is similar to a decryption key that allows the system to determine the source of permission in order to avoid the confused deputy problem.

In a capability-based system, the resources cannot be accessed directly. A user often has to use specific handlers to access any type of resources on the system. The JavaScript below is a simple object capability model consisting of four object models, including the embedded script (play by Alice), the user input, the password tester, and the system object.

```
// The executing program
var pwTester = function(getPwd){
    this.verify = function(p){
        return p === getPwd();
    }
}
```

```
7  var pwt;
8
9  // The system
10 var system = function(){
11     // suppose sysPwd is a secret
12     var sysPwd = ...;
13     ...
14     pwt = new pwTester(function(){
15         return sysPwd;
16     });
17 }();
18
19 // Alice/ embedded script
20 var alice = function() {
21     // suppose Alice hold a reference to user input
22     if (pwt.verify(userInput)){
23         writeToPublic("Login Sucess.");
24     }else{
25         writeToPublic("Login Fail.");
26     }
27 }();
```

Figure 1 is a Granovetter diagram [10] that illustrates the relationship of different computational objects in this capability system. Initially, Alice holds a reference to the user input, while the `pwTester` holds a reference to the system password.

In this system, Alice cannot access the system password directly. Only the `pwTester` holds a reference `getPwd` to the `system`'s internal function by which the checker can simply call this callback function and obtain the system password. By sending a `verify` message to the pwTester, the `pwTester` can obtain a copy of the reference to the user input, which can

then be used to validate against the system password.

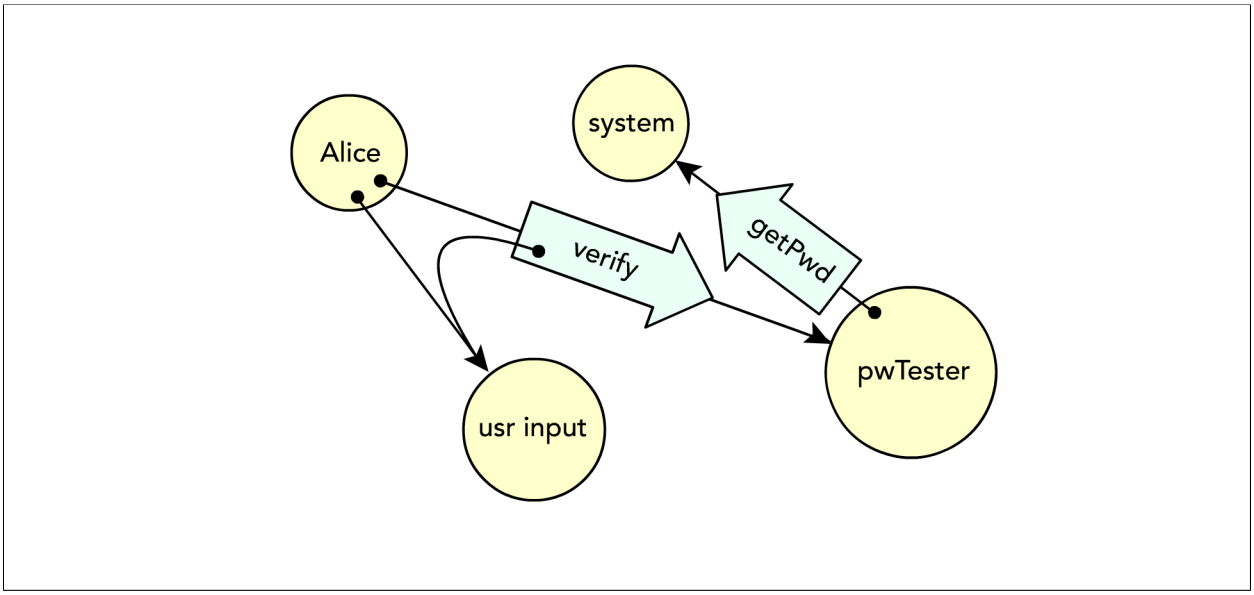

Figure 1: Simple Capability Systems.

2.2.1 Declassification

Although information flow control provides strong security guarantees, this is not an ideal solution in most real-world situations. One illustration would be a login system that is required to inform public users about the correctness of the password. Consider the following example:

```
1 password = < k ? "secret" : 0 >;
2 sysHash = sha1_hash(password);
3 usrHash = sha1_hash(userInput);
4 if (sysHash === usrHash){
5     writeToPublic("Login Sucess.");
6 } else {
7     writeToPublic("Password Mismatch.");
8 }
```

With faceted execution, the password `secret` is considered private, whereas the `userInput` is considered to be a public input. Since `writeToPublic(message)` is a public output function, the user's password will always be compared with the same dummy integer 0, instead of the actual secret, contrary to the purpose of having password checking.

One of the strategies is to integrate a declassification mechanism into a special hash function. Here we assume that the hash function cannot be overwritten and is stored somewhere secure. For example:

```
1 function sha1_hash(password){
2     var hash = sha1(password);
3     var v = defacet(hash);
4     return v;
5 }
```

Due to the one-way property of cryptographic hash functions, it is not possible to reconstruct the password from its hash value. In this case, although one bit of the secret has leaked out, the attacker cannot learn more than the correctness of the password. As a result, we can still claim that the system is secure.

2.3 Security Labels as Object Capabilities

Based on the concept of the object capability model, this paper introduces a more natural way of handling declassification, which also allows programmers to:

- create new security labels
- create faceted values using security labels
- build information release policies
- declassify faceted values based on their release policy

To demonstrate the concept of capability-base declassification, we introduce a modified JavaScript language, named Faceted JS. This language automatically enforces information flow control during runtime and allows application developers to construct and release faceted values using security labels and output channels.

2.3.1 Security Label

In Faceted JS, security labels are immutable and unforgeable first-class elements. Instantiating a label object creates a reference to a new security label. Using the reference, programmers can associate sensitive data with security labels, to create a faceted value.

```
1 var k = new Label();
2 var pwd = setSecurity(k, "secret");
```

The above assignment sets the variable `pwd` to:

$$\langle \texttt{ref}_{\texttt{label}} \texttt{ ? "secret" : 0} \rangle$$

2.3.2 Output Channel

The same-origin policy enforces information flow policy by restricting interactions between two different origins. Taking inspiration from the same-origin policy, we formalize the relationship between outputs and security labels, introducing the concept of security channels. This tool allows programmers to organize the outputs into different security channels and also to associate these channels with different permission sets.

By default, all output channels belong to the public channel that does not associate with any permission:

```
1 var publicChannel = new Channel("*");
```

The following implementation adds permissions to `system.com` by instantiating a `Channel` object and associating the security labels `k1, k2`, using the `addPermission` method:

```
var k1 = new Label();
var k2 = new Label();
var ch = new Channel("system.com");
ch.addPermission(k1);
ch.addPermission(k2);
```

In the following example, we define two security channels (`ch1,ch2`) with different security levels. After that, we simply output the secret information by loading an image from different public servers.

```
var k1 = new Label();
var k2 = new Label();
var secret = setSecurity(k1, "secret");
var top_secret = setSecurity(k2, "top-secret");
var ch1 = new Channel("system.com");
var ch2 = new Channel("system.com/admin");

ch1.addPermission(k1);
ch2.addPermission(k2);

// public channel
img1.setAttribute("src", "evil.com" + secret + top_secret + ".jpg");
// secret channel
img2.setAttribute("src", "system.com" + secret + top_secret + ".jpg");
// top secret channel
img3.setAttribute("src", "system.com/admin" + secret + top_secret + ".jpg"
   );
```

With Faceted JS, all output is validated implicitly through the corresponding security channels at runtime. As a result, `evil.com` can only read the public facet of `secret` and `top_secret`. On the second attempt, `ch1` is associated with `k1`. Therefore, `system.com` can see the private facet of secret and the public facet of `top_secret`. On the last attempt, `system.com/admin` can see both `secret` and `top_secret`, as `ch2` is associated with `k2` and `ch1` is the parent of `ch2`.

2.3.3 Declassify Function

Often times, a system is required to release sensitive data in order to be useful. Using a `defacet` function and the reference of a security label, a programmer might declassify a secret and release it to a public channel. For the sake of simplicity, we simply use the `write` method to write the secret explicitly to the public channel.

```
1 var k = new Label();
2 var secret = setSecurity(k, "secret");
3 var ch = new Channel("*");
4 var s = defacet(k, secret);
5 ch.write(s);
```

Security labels, output channels, and defacet methods allow developers to build secure applications. However, developers might fail to realize the information flow of security labels, thereby allowing attackers to exploit security labels and access sensitive information. Consider the following implementation of password checking, with the reference of security label `k` visible to public observers.

```
1 var publicChannel = new Channel("*");
2 var k = new Label();
```

```
3 var pwd = setSecurity(k, "secret");
4 var input = document.getElementById("pwd ").text;
5 if (defacet(k, md5(pwd)) === md5(input)){
6     publicChannel.write("login successfully");
7 }else{
8     publicChannel.write("password mismatch");
9 }
```

With code injection, an attacker can either add permission to a public channel or simply defacet a classified value with the same authority as application developers.

```
1 // injection 1
2 publicChannel.addPermission(k);
3 publicChannel.write(secret);
4
5 // injection 2
6 s = defacet(k, secret);
7 publicChannel.write(secret);
```

Instead of tracking the flow of all security labels during runtime, we treat labels as capabilities, to grant permission to work with sensitive information. As previously stated, object-capability is a security concept that utilizes unforgeable references to describe transferable rights on different object models [7]. In Faceted JS, we describe a reference as an object that is associated with security labels. A security label is similar to a decryption key, whereby a computer programmer must obtain the reference in order to perform security critical tasks, which include:

- the classification of confidential data
- the declassification of faceted value

- the modification of channel permissions

To protect the security labels and cryptographic functions from unwanted access, we can define the security label `k` and the cryptographic hash function `hash` inside the `SecureContext` object. Although public observers are able to create a password and obtain the corresponding hash value, direct access to the password is strictly prevented.

```
1  function SecureContext(){
2      // hidden within the pwTester function
3      var k = new Label();
4
5      this.makePassword = function(p){
6          return setSecurity(k, p);
7      };
8      this.hash = function(p){
9          var h = md5(p);
10         // Access to k grants permission to deconstruct p
11         return defacet(k, h);
12     };
13     this.md5 = function(pwd){
14         ...
15         return hash;
16     }
17 }
```

CHAPTER 3

Syntax of Faceted JS

3.1 Syntax of the Language

JavaScript is an interpreted, multi-paradigm, dynamic programming language. It is widely used in client-side and server-side scripting environment. It is also an ECMAScript-based language supported by all modern browsers. It allows developers to build interactive web and server applications efficiently. Yet, it also introduces numerous security issues. For this reason, we offer Faceted JS, a modified lightweight JavaScript language with the support of facet values. Faceted JS consists of two syntactic elements: statement and expression.

3.1.1 Statements

Faceted JS is a modified subset of JavaScript, supporting most basic JavaScript syntax, including `if / if-else` statements, `for` statements, `while` statements and `do-while` statements. Moreover, Faceted JS supports compound statements, variable declarations using the `var` keyword, and function declarations. Similar to NodeJS, the language also supports system logging function `system.log`. In Faceted JS, a regular statement is only expected to perform an action, whereas a `return` statement is expected to produce a return value from a function call.

Figure 2 shows the statement expressions in Faceted JS. The symbol `stmt` denotes a statement, while the symbol `e` denotes an expression.

$stmt$ ::=	*Statement*
e	expression statement
$\{stmt\}$	compound statement
$stmt_1; stmt_2$	sequential statement
`var` e	variable declaration
`if` (e) $stmt$	if statement
`if` (e) $stmt$ `else` $stmt$	if-else statement
`while` (e) $stmt$	while statement
`do` $stmt$ `while` (e)	do while statement
`for` $(e_1;e_2;e_3)$ $stmt$	for statement
`system.log`(e)	predefined print statement
`function` $f(\bar{e_i})$ $stmt$	function declaration
`return` e	return statement

Figure 2: Statement Syntax.

3.1.2 Expressions

Unlike a statement expression, an expression is always expected to produce a useful value. Faceted JS supports variable expressions, assignments, and simple data types, such as double, boolean, string, array, and JSON object. In terms of property accessors, Faceted JS supports dot notation and bracket notation. As is the case with most programming languages, this language also includes unary/binary operations, function applications, object instantiations, and anonymous functions.

Figure 3 shows the expressions, values, assignment operators, unary operators, and binary operators syntax in Faceted JS. The symbol `e` simply represents an expression.

$e ::=$		*Expressions*
	v	values
	(e)	parenthesized expression
	x	variable name
	f	function name
	$unop\ e$	prefix unary operations
	$e\ unop$	postfix unary operations
	$e\ binop\ e$	binary operations
	$e\ asop\ e$	assignment statement
	$e_1[e_2]$	bracket notation
	$e.attr$	dot notation
	$f(\bar{e_i})$	function application
	`function` $(\bar{e_i})$	anonymous function
	`new` $f(\bar{e_i})$	new object
$v ::=$		*Values*
	`undefined` | `null` | `NaN`	constant
	d	double
	b	boolean
	str	string
$unop ::=$		*Unary Operators*
	$! \mid ++ \mid --$	unary operators
$binop ::=$		*Binary Operators*
	$+ \mid - \mid * \mid / \mid \%$	arithmetic operators
	$> \mid >= \mid < \mid <= \mid$	logical operators
	$== \mid != \mid === \mid !==$	
	$\&\& \mid \mid\mid$	and/or operators
$asop ::=$		*Assignment Operators*
	$= \mid += \mid -= \mid *= \mid /= \mid \%=$	

Figure 3: Expression Syntax.

3.2 Syntactical Differences from JavaScript

To enforce information flow control during runtime, Faceted JS introduces additional syntax for security labels and output channels, as well as functions for classifying and

declassifying confidential data.

The following figure shows the additional syntax in Faceted JS.

$e ::=$		*Expressions*
	`new Label()`	label declaration
	`new Channel`(id)	security channel
	`setSecurity`(l, e_1, e_2)	classify function
	`defacet`(k, e)	declassify function

Figure 4: Syntax for Information Flow Control.

Security label: A security label is a special type of security object that cannot be redefined by application developers. A security label is a first-class value that can be dynamically created, destroyed, or passed as a parameter in a function call. It can do all the things that an object can do. By instantiating a `Label` object using the `new Label()` expression, a reference to a new security label is allocated. In this section, we use R to denote a reference to a security object.

$$\texttt{new Label()} \xrightarrow{evaluate} R_{label}$$

Security channel: A security channel is also a special type of security object that can be created by instantiating the `Channel` object. To associate a security label `k` with an output

channel `ch1`, a programmer can invoke the method `ch1.addPermission(k)`.

$$\texttt{new Channel}() \xrightarrow{evaluate} \{\ R : [\,]\ \}$$

$$\texttt{ch1.addPermission}(\texttt{k1}) \xrightarrow{evaluate} \{\ R_{ch1} : [R_{k1}]\ \}$$

$$\texttt{ch1.addPermission}(\texttt{k2}) \xrightarrow{evaluate} \{\ R_{ch1} : [R_{k1}, R_{k2}]\ \}$$

Classify function: Classification is a process for converting public data into confidential data. The classify function `setSecurity(k, h, l)` takes three arguments. The first argument is the reference of a security label. The second and the third arguments are the high-confidentiality data (H) and the low-confidentiality data (L), respectively. The result of this function call creates a faceted value $\langle k\ \texttt{?}\ H\ \texttt{:}\ L\rangle$ by which an authorized observer can read the confidential data and public observers see only default values.

$$\texttt{setSecurity}(\texttt{k}, \texttt{secret}) \xrightarrow{evaluate} \langle R_k\ \texttt{?}\ secret\ \texttt{:}\ \bot\rangle$$

$$\texttt{setSecurity}(\texttt{k}, \texttt{secret}, \texttt{null}) \xrightarrow{evaluate} \langle R_k\ \texttt{?}\ secret\ \texttt{:}\ null\rangle$$

Declassify function: Declassification is a process for declassifying sensitive data in a controlled manner. Providing the reference of a label and a faceted expression `e`, the declassify function `defacet(k, e)` declassifies a faceted value with respect to the security label `k`.

$$\texttt{defacet}(R_k, \langle R_k\ \texttt{?}\ secret\ \texttt{:}\ null\rangle) \xrightarrow{evaluate} secret$$

$$\texttt{defacet}(R_{k_2}, \langle R_{k_1}\ \texttt{?}\ \langle R_{k_2}\ \texttt{?}\ secret\ \texttt{:}\ \bot\rangle\ \texttt{:}\ \bot\rangle) \xrightarrow{evaluate} \langle R_{k_1}\ \texttt{?}\ secret\ \texttt{:}\ \bot\rangle$$

$$\texttt{defacet}(R_{k_3}, \langle R_{k_1}\ \texttt{?}\ \langle R_{k_2}\ \texttt{?}\ secret\ \texttt{:}\ \bot\rangle\ \texttt{:}\ \bot\rangle) \xrightarrow{evaluate} \langle R_{k_1}\ \texttt{?}\ \langle R_{k_2}\ \texttt{?}\ secret\ \texttt{:}\ \bot\rangle\ \texttt{:}\ \bot\rangle$$

3.3 Standard Encodings

For the sake of simplicity, we can define if statements, while statements, do-while statements, and for loops using standard encoding. Figure 5 presents standard encodings for these common JavaScript constructs.

Standard Encodings

$$
\begin{aligned}
\texttt{if}\ (e_1)\ e_2 &\stackrel{\text{def}}{=\!=} \texttt{if(}e_1\texttt{)}\ e_2\ \texttt{else}\ \bot \\
\texttt{while(}e_1\texttt{)}\ e_2 &\stackrel{\text{def}}{=\!=} \texttt{if(}e_1\texttt{)\{}e_2\texttt{;while(}e_1\texttt{)}\ e_2\texttt{\}} \\
\texttt{do}\ e_1\ \texttt{while(}e_2\texttt{)} &\stackrel{\text{def}}{=\!=} e_1\texttt{;while(}e_2\texttt{)}\ e_1 \\
\texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ e_4 &\stackrel{\text{def}}{=\!=} e_1\texttt{;while(}e_2\texttt{)}\ \texttt{\{}e_4\texttt{;}e_3\texttt{\}} \\
\texttt{function}(x)\{e\} &\stackrel{\text{def}}{=\!=} \lambda x.e \\
\texttt{setSecurity}(e_1, e_2) &\stackrel{\text{def}}{=\!=} \texttt{setSecurity}(e_1, e_2, \bot)
\end{aligned}
$$

Figure 5: Standard Encodings.

CHAPTER 4

Formal Operational Semantics

4.1 Standard Semantics

Runtime Syntax

$$
\begin{array}{lcr}
\phi \in \mathit{Input/Output} & = & \mathit{File} \to \mathit{Value} \\
\sigma \in \mathit{store} & = & \mathit{Address} \to_p \mathit{Value} \\
\theta \in \mathit{subst} & = & \mathit{Var} \to_p \mathit{Value} \\
v \in \mathit{Value} & ::= & \mathit{Constant} \mid a \mid (\lambda x.e) \\
r \in \mathit{Return\ Value} & ::= & \varnothing \mid v
\end{array}
$$

Figure 6: Runtime Syntax and Standard Encodings.

The standard language in this section is a subset of the JavaScript that serves as a basis for Faceted JavaScript in the next section. This paper describes the program execution based on structural operational semantics. Operational semantics, which describes the runtime behavior of a programming language, are classified into two categories: small-step and big-step semantics. Small-step semantics describe the evaluation process in small, individual steps, while big-step semantics describe the evaluation process in one single step.

In this paper, we present the evaluation rules in big-step semantics. To formalize the process, we introduce several runtime syntax:

I/O System - We use ϕ to denote the state of the system resources, which maps a file f to a sequence of values or a URI to a remote resource.

Store - A store σ is similar to the heap of a computer system in which all references are maintained. A store maps an address to an actual value v. The notation σ[a:=v] updates

the store σ, mapping the reference a to the value v. In contrast, $v = \sigma(a)$ dereferences an address a, assigning the value to the variable v.

Value - We use v to represent an actual value, such as a double, boolean, string, function, undefined, NaN, null, or a reference to an array or to an object.

Substitution - The Greek letter θ denotes substitution; the notation θ[x:=v] substitutes v for x during a function call.

Return - The purpose of a regular statement is to perform some actions, while the purpose of a return statement is to evaluate an expression and to return a useful value to the function caller. In this paper, the letter r represents a return value and $\varnothing$ indicates a statement that does not produce a return value.

4.1.1 Statement Evaluations

Statement Rules: $\boxed{\phi, \sigma, \theta, e \downarrow^s \phi', \sigma', r}$

$$[\text{S-EXP}] \quad \frac{\phi, \sigma, \theta, e \downarrow^e \phi', \sigma', v}{\phi, \sigma, \theta, e \downarrow^s \phi', \sigma', \varnothing}$$

$$[\text{S-IFTRUE}] \quad \frac{\begin{array}{c}\phi, \sigma, \theta, e_1 \downarrow^e \phi', \sigma_1, \texttt{true} \\ \phi, \sigma_1, \theta, e_2 \downarrow^s \phi', \sigma', r\end{array}}{\phi, \sigma, \theta, \texttt{if(}e_1\texttt{)}\ e_2\ \texttt{else}\ e_3 \downarrow^s \phi', \sigma', r}$$

$$[\text{S-IFFALSE}] \quad \frac{\begin{array}{c}\phi, \sigma, \theta, e_1 \downarrow^e \phi_1, \sigma_1, v \\ v \in \{\texttt{false}, \texttt{null}, \texttt{nan}, \texttt{undefined}, \bot\} \\ \phi_1, \sigma_1, \theta, e_3 \downarrow^s \phi', \sigma', r\end{array}}{\phi, \sigma, \theta, \texttt{if(}e_1\texttt{)}\ e_2\ \texttt{else}\ e_3 \downarrow^s \phi', \sigma', r}$$

$$[\text{S-RETURN}] \quad \frac{\phi, \sigma, \theta, e \downarrow^e \phi', \sigma', v}{\phi, \sigma, \theta, \texttt{return}\ e \downarrow^s \phi', \sigma', v}$$

$$[\text{S-SEQ-RETURN}] \quad \frac{\phi, \sigma, \theta, e_1 \downarrow^s \phi', \sigma', r \qquad r \neq \varnothing}{\phi, \sigma, \theta, e_1\texttt{;}e_2 \downarrow^s \phi', \sigma', r}$$

$$[\text{S-SEQ}] \quad \frac{\begin{array}{c}\phi, \sigma, \theta, e_1 \downarrow^s \phi_1, \sigma_1, r_1 \qquad r_1 = \varnothing \\ \phi_1, \sigma_1, \theta, e_2 \downarrow^s \phi', \sigma', r_2\end{array}}{\phi, \sigma, \theta, e_1\texttt{;}e_2 \downarrow^s \phi', \sigma', r_2}$$

$$[\text{S-READ}] \quad \frac{\phi, \sigma, \theta, \sigma(f) = v.w \downarrow^e \phi', \sigma', v}{\phi, \sigma, \theta, \texttt{read}(f) \downarrow^s \phi'[f := w], \sigma', v}$$

$$[\text{S-WRITE}] \quad \frac{\phi, \sigma, \theta, e \downarrow^e \phi', \sigma_1, v}{\phi, \sigma, \theta, \texttt{write}(f, e) \downarrow^s \phi'[f := \phi'(f).v], \sigma', v}$$

Figure 7: Statement Evaluation Rules.

Evaluation rules as shown in Figure 7 evaluate a statement expression `e`, a store σ, an I/O system ϕ, and a substitution θ into a return value `r`, a possibly modified store σ', and a modified I/O system ϕ'.

$$\textit{Statement evaluation}: \ \phi, \sigma, \theta, e \downarrow^s \phi', \sigma', r$$

S-EXP - This rule is similar to an expression evaluation, except that it does not produce a valid return value.

S-IFTRUE - Provided that the expression e_1 is true in an if-else statement, this evaluation rule simply evaluates the corresponding expression e_2.

S-IFFALSE - Provided that the expression e_1 is not true (that it is false, undefined, null, or $\perp$), this evaluation rule evaluates the else expression e_3. The symbol $\perp$ is a value that represents "nothing".

S-RETURN - This rule takes the expression e in a return statement and evaluates it into a value. This non-empty value v immediately breaks out of a function call, returning to the function caller.

S-SEQ-RETURN - Running a JavaScript program is equivalent to executing a sequence of statements. If the evaluation of the statement expression e_1 creates a return value r such that $r \neq \varnothing$, the rest of the code in the current function application, denoted by e_2, will not be executed.

S-SEQ - This rule implies that if the statement expression e_1 does not produce a valid return value, the rest of the statements e_2 are evaluated in the context of the file system ϕ_1 and the store σ_1.

S-READ - This rule simply reads a value v from the file system ϕ.

S-WRITE - This rule evaluates the expression e into a value v. This value is thus written to the file system. We identify such a modified file system as ϕ'.

4.1.2 Expression Evaluations

Expression Rules: $\boxed{\phi, \sigma, \theta, e \downarrow^e \phi', \sigma', v}$

$$[\text{S-BOT}] \quad \frac{}{\phi, \sigma, \theta, \bot \downarrow^e \phi, \sigma, \bot}$$

$$[\text{S-VAL}] \quad \frac{}{\phi, \sigma, \theta, v \downarrow^e \phi, \sigma, v}$$

$$[\text{S-OP}] \quad \frac{\begin{array}{c}\phi, \sigma, \theta, e_1 \downarrow^e \phi_1, \sigma_1, v_1 \\ \phi_1, \sigma_1, \theta, e_2 \downarrow^e \phi', \sigma', v_2 \\ v_1 \ op \ v_2 \downarrow v\end{array}}{\phi, \sigma, \theta, e_1 \ op \ e_2 \downarrow^e \phi', \sigma', v}$$

$$[\text{S-VAR}] \quad \frac{x \in domain(\theta) \qquad \theta(x) = v}{\phi, \sigma, \theta, x \downarrow^e \phi', \sigma', v}$$

$$[\text{S-FUNC}] \quad \frac{}{\phi, \sigma, \theta, \lambda x.e \downarrow^e \phi, \sigma, (\lambda x.e, \theta)}$$

$$[\text{S-APP}] \quad \frac{\begin{array}{c}\phi, \sigma, \theta, e_1 \downarrow^e \phi_1, \sigma_1, (\lambda x.e, \theta_1) \\ \phi_1, \sigma_1, \theta, e_2 \downarrow^e \phi_2, \sigma_2, v \\ \phi_2, \sigma_2, \theta_1[x := v], e \downarrow^s \phi', \sigma', v'\end{array}}{\phi, \sigma, e_1(e_2), e \downarrow^e \phi', \sigma', v'}$$

$$[\text{S-APP-BOT}] \quad \frac{\phi, \sigma, \theta, e_1 \downarrow^e \phi_1, \sigma_1, \bot \qquad \phi_1, \sigma_1, \theta, e_2 \downarrow^e \phi', \sigma', v}{\phi, \sigma, e_1(e_2), e \downarrow^e \phi', \sigma', \bot}$$

$$[\text{S-ASSIGN}] \quad \frac{\phi, \sigma, \theta, e_1 \downarrow^e \phi_1, \sigma_1, a \qquad \phi_1, \sigma_1, \theta, e_2 \downarrow^e \phi', \sigma', v}{\phi, \sigma, \theta, e_1 = e_2 \downarrow^e \phi', \sigma'[a := v], v}$$

$$[\text{S-ASSIGN-BOT}] \quad \frac{\phi, \sigma, \theta, e_1 \downarrow^e \phi_1, \sigma_1, \bot \qquad \phi_1, \sigma_1, \theta, e_2 \downarrow^e \phi', \sigma_2, v}{\phi, \sigma, \theta, e_1 = e_2 \downarrow^e \phi', \sigma', v}$$

$$[\text{S-NEW}] \quad \frac{\begin{array}{c}\phi, \sigma, \theta, e_1 \downarrow^e \phi_1, \sigma_1, (\lambda x.e, \theta_1) \\ \phi_1, \sigma_1, \theta, e_2 \downarrow^e \phi_2, \sigma_2, v \\ \phi_2, \Sigma_2, \theta_1[x := v], e \downarrow^s \phi', \sigma_3, v' \\ fresh \ a \qquad a \notin domain(\sigma_3) \\ \sigma' = \sigma_3[a = v']\end{array}}{\phi, \sigma, \theta, \texttt{new} \ e_1(e_2) \downarrow^e \phi', \sigma', a}$$

Figure 8: Statement Evaluation Rules.

Evaluation rules as shown in Figure 8 evaluate an expression $\mathtt{e}$, a store σ, a file system ϕ, and a substitution θ into an actual value $\mathtt{v}$, a possibly modified store σ', and a modified file system ϕ'.

$$\textit{Expression evaluation}: \ \phi, \sigma, \theta, e \downarrow^{e} \phi', \sigma', v$$

S-BOT - The symbol $\perp$ stands for nothing or no value in our semantics. As a result, evaluation of a $\perp$ value is also a $\perp$ value and does not change the state of the file system or the store.

S-VAL - Given a value v as argument, the evaluation simply returns the same value v.

S-OP - This rule takes the expressions e_1 and e_2, returning the result of the binary operation of v_1 op v_2.

S-VAR - If the variable x belongs to the domain of θ, then it returns the value where $v = \theta(x)$.

S-FUNC - This evaluation rule takes a function declaration as argument and returns a closure $(\lambda x.e, \theta_1)$, composed of a λ expression and a substitution θ.

S-APP - If given an expression e_1, this expression evaluates to a closure $(\lambda x.e, \theta_1)$. As a result, the function block e_2 is evaluated in the context of the substitution $\theta_1[x := v]$, possibly returning a value from the function call.

S-ASSIGN - This rule evaluates a variable expression e_1 and an expression e_2 into a and a value v, where the notation $\sigma_2[a := v]$ denotes a new store that is identical to σ_2, except that it maps a to v.

S-NEW - Instantiating an object is similar to the evaluation process of a function application, except that it allocates an object, returning a reference to an object.

4.2 Faceted Semantics

Runtime Syntax

$$
\begin{array}{rcll}
\Phi & \in & \mathit{Input/Output} & = & (\mathit{File/URI} \to \mathit{Value}) \\
\Sigma & \in & \mathit{Store} & = & \mathit{Address} \to_p \mathit{Value} \\
\Theta & \in & \mathit{Subst} & = & \mathit{Var} \to_p \mathit{Value} \\
\mathit{pc} & \in & \mathit{Program\ Counter} & = & 2^{\mathit{Branch}} \\
 & & & & \\
V & \in & \mathit{Value} & ::= & V_{raw} \mid k \mid c \mid \langle k \ ?\ e_1 : e_2 \rangle \\
V_{raw} & \in & \mathit{Raw\ Value} & ::= & \mathit{Constant} \mid a \mid (\lambda x.e) \\
R & \in & \mathit{Return\ Value} & ::= & \varnothing \mid V \\
a & \in & \mathit{Address} & & \\
k & \in & \mathit{Security\ Label} & ::= & k \mid \bar{k} \\
c & \in & \mathit{Security\ Channel} & ::= & \{\bar{k_i}\}
\end{array}
$$

Figure 9: Runtime Syntax and Standard Encodings.

To enforce dynamic information flow control in JavaScript, faceted semantics extends the standard semantics by introducing new security-related syntax. Although this additional set of semantics enhances the expressiveness of the language, it also introduces many complexities. In order to distinguish the faceted semantics from the original semantics, we use Φ, Σ, and Θ, to denote an I/O system, a store, and a substitution, respectively. In addition to basic data types, Value V now also includes first-class security labels k and faceted values $\langle k \ ?\ \mathit{secret} : \bot \rangle$, as well as security channels c.

4.2.1 Program Counter

To manage the complexity of implicit flow, a program counter is introduced. A program counter *pc* is a list of security labels for keeping track of the influence of private data across different security levels. Consider the following example, in which the variable *hi* is a faceted

value that has a private facet `true` and a public facet `false`. Since the conditional statement only appears to be `true` in the private facet of `hi`, the interpreter sets the program counter *pc* to reflect the influence of the private data. More specifically, the if-statement executes twice with $pc = \{k\}$ and $pc = \{\bar{k}\}$. The assignment `lo` = `true` updates only the private facet of `lo`, while the assignment `lo` = `false` updates only the public facet of `lo`. As a result, the value of `lo` evaluates to the faceted value $\langle k \texttt{ ? } true \texttt{ : } false \rangle$, which accurately reflects the actual value of `lo` across different execution branches.

```
1  var k = new Label();
2  var lo;
3  var hi = setSecurity(k, true, false);
4  // pc={}, hi: < k ? true : false >
5  if (hi){
6      // pc: {k}, lo: undefined
7      lo = true;
8      // pc: {k}, lo: < k ? true : undefined >
9  }else{
10     // pc: {k̄}, lo: < k ? true : undefined >
11     lo = false;
12     // pc: {k̄}, lo: < k ? true : false >
13 }
```

4.2.2 Statement Evaluations

Statement Rules: $\boxed{\Phi, \Sigma, \Theta, e \Downarrow_{pc}^{stmt} \Phi', \Sigma', R}$

$$[\text{F-EXP}] \quad \frac{\Phi, \Sigma, \Theta, e \Downarrow_{pc}^{expr} \Phi', \Sigma', V}{\Phi, \Sigma, \Theta, e \Downarrow_{pc}^{stmt} \Phi', \Sigma', \varnothing}$$

$$[\text{F-IFTRUE}] \quad \frac{\begin{array}{c}\Phi, \Sigma, \Theta, e_1 \Downarrow_{pc}^{expr} \Phi', \Sigma_1, \texttt{true} \\ \Phi, \Sigma_1, \Theta, e_2 \Downarrow_{pc}^{stmt} \Phi', \Sigma', R\end{array}}{\Phi, \Sigma, \Theta, \texttt{if(}e_1\texttt{)}\ e_2\ \texttt{else}\ e_3 \Downarrow_{pc}^{stmt} \Phi', \Sigma', R}$$

$$[\text{F-IFFALSE}] \quad \frac{\begin{array}{c}V \in \{\texttt{false}, \texttt{null}, \texttt{nan}, \texttt{undefined}, \bot\} \\ \Phi, \Sigma, \Theta, e_1 \Downarrow_{pc}^{expr} \Phi_1, \Sigma_1, V \\ \Phi_1, \Sigma_1, \Theta, e_3 \Downarrow_{pc}^{stmt} \Phi', \Sigma', R\end{array}}{\Phi, \Sigma, \Theta, \texttt{if(}e_1\texttt{)}\ e_2\ \texttt{else}\ e_3 \Downarrow_{pc}^{stmt} \Phi', \Sigma', R}$$

$$[\text{F-IFSPLIT}] \quad \frac{\begin{array}{ll}\multicolumn{2}{c}{\Phi, \Sigma, \Theta, e_1 \Downarrow_{pc}^{expr} \Phi_1, \Sigma_1, \langle k\ \texttt{?}\ V_h\ \texttt{:}\ V_l\rangle} \\ k \notin pc & \Phi_1, \Sigma_1, \Theta, \texttt{if(}V_h\texttt{)}\ e_2\ \texttt{else}\ e_3 \Downarrow_{pc\cup\{k\}}^{stmt}, \Phi_2, \Sigma_2, R_1 \\ \overline{k} \notin pc & \Phi_2, \Sigma_2, \Theta, \texttt{if(}V_l\texttt{)}\ e_2\ \texttt{else}\ e_3 \Downarrow_{pc\cup\{\overline{k}\}}^{stmt}, \Phi', \Sigma', R_2\end{array}}{\Phi, \Sigma, \Theta, \texttt{if(}e_1\texttt{)}\ e_2\ \texttt{else}\ e_3 \Downarrow_{pc}^{stmt} \Phi', \Sigma', \langle\langle k\ \texttt{?}\ R_1\ \texttt{:}\ R_2\rangle\rangle}$$

$$[\text{F-IFLEFT}] \quad \frac{\begin{array}{ll}\multicolumn{2}{c}{\Phi, \Sigma, \Theta, e_1 \Downarrow_{pc}^{expr} \Phi_1, \Sigma_1, \langle k\ \texttt{?}\ V_h\ \texttt{:}\ V_l\rangle} \\ k \in pc & \Phi_1, \Sigma_1, \Theta, \texttt{if(}V_h\texttt{)}\ e_2\ \texttt{else}\ e_3 \Downarrow_{pc}^{stmt} \Phi', \Sigma', R\end{array}}{\Phi, \Sigma, \Theta, \texttt{if(}e_1\texttt{)}\ e_2\ \texttt{else}\ e_3 \Downarrow_{pc}^{stmt} \Phi', \Sigma', R}$$

$$[\text{F-IFRIGHT}] \quad \frac{\begin{array}{ll}\multicolumn{2}{c}{\Phi, \Sigma, \Theta, e_1 \Downarrow_{pc}^{expr} \Phi_1, \Sigma_1, \langle k\ \texttt{?}\ V_h\ \texttt{:}\ V_l\rangle} \\ \overline{k} \in pc & \Phi_1, \Sigma_1, \Theta, \texttt{if(}V_l\texttt{)}\ e_2\ \texttt{else}\ e_3 \Downarrow_{pc}^{stmt} \Phi', \Sigma', R\end{array}}{\Phi, \Sigma, \Theta, \texttt{if(}e_1\texttt{)}\ e_2\ \texttt{else}\ e_3 \Downarrow_{pc}^{stmt} \Phi', \Sigma', R}$$

$$[\text{F-READ-IGNORE}] \quad \frac{pc\ \texttt{not visible to}\ view(f)}{\Phi, \Sigma, \Theta, \texttt{read(}f\texttt{)} \Downarrow_{pc}^{stmt} \Phi', \Sigma', \bot}$$

$$[\text{F-READ}] \quad \frac{\begin{array}{cc}\Phi(f) = v.w & L = view(f) \\ pc\ \texttt{visible to}\ L & pc' = L \cup \{\overline{k} | k \notin L\}\end{array}}{\Phi, \Sigma, \Theta, \texttt{read(}f\texttt{)} \Downarrow_{pc}^{stmt} \Phi[f := w], \Sigma, \langle\langle pc'\ \texttt{?}\ v\ \texttt{:}\ \bot\rangle\rangle}$$

Figure 10: Faceted Statement Rules 1.

$$[\text{F-WRITE-IGNORE}] \quad \frac{\Phi, \Sigma, \Theta, e \downarrow\downarrow_{pc}^{expr} \Phi', \Sigma', V \quad pc \texttt{ not visible to } view(f)}{\Phi, \Sigma, \Theta, \texttt{write}(f, e) \downarrow\downarrow_{pc}^{stmt} \Phi', \Sigma', V}$$

$$[\text{F-WRITE}] \quad \frac{\Phi, \Sigma, \Theta, e \downarrow\downarrow_{pc}^{expr} \Phi', \Sigma_1, V \quad pc \texttt{ visible to } view(f) \quad L = view(f) \quad v = L(V)}{\Phi, \Sigma, \Theta, \texttt{write}(f, e) \downarrow\downarrow_{pc}^{stmt} \Phi'[f := \Phi'(f).v], \Sigma', V}$$

$$[\text{F-RETURN}] \quad \frac{\Phi, \Sigma, \Theta, e \downarrow\downarrow_{pc}^{expr} \Phi', \Sigma', V}{\Phi, \Sigma, \Theta, \texttt{return } e \downarrow\downarrow_{pc}^{stmt} \Phi', \Sigma', V}$$

$$[\text{F-SEQ-RETURN}] \quad \frac{\Phi, \Sigma, \Theta, e_1 \downarrow\downarrow_{pc}^{stmt} \Phi', \Sigma', R \quad R \neq \varnothing}{\Phi, \Sigma, \Theta, e_1 \texttt{;} e_2 \downarrow\downarrow_{pc}^{stmt} \Phi', \Sigma', R}$$

$$[\text{F-SEQ}] \quad \frac{\Phi, \Sigma, \Theta, e_1 \downarrow\downarrow_{pc}^{stmt} \Phi_1, \Sigma_1, R_1 \quad R_1 = \varnothing \quad \Phi_1, \Sigma_1, \Theta, e_2 \downarrow\downarrow_{pc}^{stmt} \Phi', \Sigma', R_2}{\Phi, \Sigma, \Theta, e_1 \texttt{;} e_2 \downarrow\downarrow_{pc}^{stmt} \Phi', \Sigma', R_2}$$

Figure 11: Faceted Statement Rules 2.

Based on the state of the program counter `pc`, evaluation rules as shown in Figure 10 and Figure 11 evaluate a statement expression `e`, a store Σ, an I/O system Φ, and a substitution Θ into an actual value `V`, a possibly modified store Σ', and a modified file system Φ'.

$$\textit{Statement evaluation}: \ \Phi, \Sigma, \Theta, e \downarrow\downarrow_{pc}^{stmt} \Phi', \Sigma', V$$

F-IFSPLIT - Given a faceted condition $\langle k \texttt{ ? } V_h \texttt{ : } V_l \rangle$, an if-statement may evaluate differently at high and low security levels. To keep track of the information across two branches simultaneously, this rule evaluates the high-security expression with the program counter $pc \cup \{k\}$ and the low-security expression with the program counter $pc \cup \{\overline{k}\}$, combining their results into a single faceted value $\langle\langle k \texttt{ ? } R_1 \texttt{ : } R_2 \rangle\rangle$.

F-READ-IGNORE - In Faceted JS, files are associated with a set of permissions. This rule simply ignores all read operations when the program counter pc is not consistent

with the file permissions.

F-READ - Provided that the program counter is consistent with the file permission, this rule reads a raw value v from the file system, turning it into a faceted value $\langle\langle pc' \texttt{ ? } v \texttt{ : } \bot\rangle\rangle$. Although the `read` operation might be repeatedly executed across different execution branches, only one operation is executed, while the remainder of the operations are ignored by the evaluation rule [F-READ-IGNORE].

F-WRITE-IGNORE - Similar to the evaluation rule [F-READ-IGNORE], this rule simply ignores all write operations if the program counter is not visible to the view of the file.

F-WRITE - This semantics indicates that the program counter is consistent with the file permission. Based on the file permissions, this rule projects a faceted value V into a non-faceted value v, writing it into the file system.

4.2.3 Expression Evaluations

Expression Rules: $\boxed{\Phi, \Sigma, \Theta, e \Downarrow_{pc}^{expr} \Phi', \Sigma', V}$

$$[\text{F-BOT}] \quad \frac{}{\Phi, \Sigma, \Theta, \bot \Downarrow_{pc}^{expr} \Phi, \Sigma, \bot}$$

$$[\text{F-VAL}] \quad \frac{}{\Phi, \Sigma, \Theta, V \Downarrow_{pc}^{expr} \Phi, \Sigma, V}$$

$$[\text{F-VAR}] \quad \frac{\begin{array}{c} x \in domain(\Theta) \\ \Theta(x) = V \end{array}}{\Phi, \Sigma, \Theta, x \Downarrow_{pc}^{expr} \Phi', \Sigma', V}$$

$$[\text{F-NEW-LABEL}] \quad \frac{\begin{array}{c} fresh\ a \qquad a \notin domain(\Sigma) \\ v = fresh\ label \qquad \Sigma' = \Sigma[a := v] \end{array}}{\Phi, \Sigma, \Theta, \texttt{new Label}() \Downarrow_{pc}^{expr} \Phi, \Sigma', a}$$

$$[\text{F-CLASSIFY-INVALID}] \quad \frac{\Phi, \Sigma, \Theta, e_1 \Downarrow_{pc}^{expr} \Phi', \Sigma', a \qquad \Sigma'(a)\ is\ not\ a\ label}{\Phi, \Sigma, \Theta, \texttt{setSecurity}(e_1, e_2, e_3) \Downarrow_{pc}^{expr} \Phi', \Sigma', \texttt{undefined}}$$

$$[\text{F-CLASSIFY}] \quad \frac{\begin{array}{c} \Phi, \Sigma, \Theta, e_1 \Downarrow_{pc}^{expr} \Phi_1, \Sigma_1, a \qquad k = \Sigma_1(a) \\ \Phi, \Sigma_1, \Theta_1, \langle k\ ?\ e_2\ :\ e_3 \rangle \Downarrow_{pc}^{expr} \Phi', \Sigma', V \end{array}}{\Phi, \Sigma, \Theta, \texttt{setSecurity}(e_1, e_2, e_3) \Downarrow_{pc}^{expr} \Phi', \Sigma', V}$$

$$[\text{F-DECLASSIFY-INVALID}] \quad \frac{\Phi, \Sigma, \Theta, e_1 \Downarrow_{pc}^{expr} \Phi', \Sigma', a \qquad \Sigma'(a)\ is\ not\ a\ label}{\Phi, \Sigma, \Theta, \texttt{defacet}(e_1, e_2) \Downarrow_{pc}^{expr} \Phi', \Sigma', \texttt{undefined}}$$

$$[\text{F-DECLASSIFY}] \quad \frac{\begin{array}{c} \Phi, \Sigma, \Theta, e_1 \Downarrow_{pc}^{expr} \Phi_1, \Sigma_1, a \qquad k = \Sigma_1(a) \\ \Phi_1, \Sigma_1, \Theta, e_2 \Downarrow_{pc}^{expr}, \Phi', \Sigma', V \\ V' = declassify(k, V) \end{array}}{\Phi, \Sigma, \Theta, \texttt{defacet}(e_1, e_2) \Downarrow_{pc}^{expr} \Phi', \Sigma', V}$$

$$[\text{F-LEFT}] \quad \frac{k \in pc \qquad \Phi, \Sigma, \Theta, e_1 \Downarrow_{pc}^{expr} \Phi', \Sigma', V}{\Phi, \Sigma, \Theta, \langle k\ ?\ e_1\ :\ e_2 \rangle \Downarrow_{pc}^{expr} \Phi', \Sigma', V}$$

$$[\text{F-RIGHT}] \quad \frac{\overline{k} \in pc \qquad \Phi, \Sigma, \Theta, e_2 \Downarrow_{pc}^{expr} \Phi', \Sigma', V}{\Phi, \Sigma, \Theta, \langle k\ ?\ e_1\ :\ e_2 \rangle \Downarrow_{pc}^{expr} \Phi', \Sigma', V}$$

Figure 12: Faceted Expression Rules 1.

$$[\text{F-SPLIT}] \quad \frac{k \notin pc \qquad \Phi, \Sigma, \Theta, e_1 \Downarrow^{expr}_{pc \cup \{k\}}, \Phi_1, \Sigma_1, V_1 \qquad \overline{k} \notin pc \qquad \Phi_1, \Sigma_1, \Theta, e_2 \Downarrow^{expr}_{pc \cup \{\overline{k}\}}, \Phi', \Sigma', V_2}{\Phi, \Sigma, \Theta, \langle k \ ? \ e_1 \ : \ e_2 \rangle \Downarrow^{expr}_{pc} \Phi', \Sigma', \langle\langle k \ ? \ V_1 \ : \ V_2 \rangle\rangle}$$

$$[\text{F-OP}] \quad \frac{\Phi, \Sigma, \Theta, e_1 \Downarrow^{expr}_{pc} \Phi_1, \Sigma_1, V_1 \qquad \Phi_1, \Sigma_1, \Theta, e_2 \Downarrow^{expr}_{pc} \Phi', \Sigma', V_2 \qquad V = \langle\langle V_1 \ op \ V_2 \rangle\rangle}{\Phi, \Sigma, \Theta, e_1 \ op \ e_2 \Downarrow^{expr}_{pc} \Phi', \Sigma', V}$$

$$[\text{F-FUNC}] \quad \frac{}{\Phi, \Sigma, \Theta, \lambda x.e \Downarrow^{expr}_{pc} \Phi, \Sigma, (\lambda x.e, \Theta)}$$

$$[\text{F-APP}] \quad \frac{\Phi, \Sigma, \Theta, e_1 \Downarrow^{expr}_{pc} \Phi_1, \Sigma_1, V_1 \qquad \Phi_1, \Sigma_1, \Theta, e_2 \Downarrow^{expr}_{pc} \Phi_2, \Sigma_2, V_2 \qquad \Phi_2, \Sigma_2, \Theta, (V_1 \ V_2)^{func} \Downarrow^{expr}_{pc} \Phi', \Sigma', V'}{\Phi, \Sigma, \Theta, e_1(e_2) \Downarrow^{expr}_{pc} \Phi', \Sigma', V}$$

$$[\text{F-NEW-OBJ}] \quad \frac{\Phi, \Sigma, \Theta, e_1 \Downarrow^{expr}_{pc} \Phi_1, \Sigma_1, V_1 \qquad \Phi_1, \Sigma_1, \Theta, e_2 \Downarrow^{expr}_{pc} \Phi_2, \Sigma_2, V_2 \qquad \Phi_2, \Sigma_2, \Theta, (V_1 \ V_2)^{new} \Downarrow^{expr}_{pc} \Phi', \Sigma', a}{\Phi, \Sigma, \Theta, \texttt{new} \ e_1(e_2) \Downarrow^{expr}_{pc} \Phi', \Sigma', a}$$

$$[\text{F-NEW-CH}] \quad \frac{fresh \ a \qquad a \notin domain(\Sigma) \qquad v = fresh \ channel \qquad \Sigma' = \Sigma[a := v]}{\Phi, \Sigma, \Theta, \texttt{new Channel()} \Downarrow^{expr}_{pc} \Phi, \Sigma', a}$$

$$[\text{F-ASSIGN}] \quad \frac{\Phi, \Sigma, \Theta, e \Downarrow^{expr}_{pc} \Phi', \Sigma_1, V \qquad \Sigma' = assign(\Sigma_1, pc, x, V)}{\Phi, \Sigma, \Theta, x = e \Downarrow^{expr}_{pc} \Phi', \Sigma', V}$$

Binary Operation: $\boxed{op \ : \ Value \times Value \ \rightarrow \ Value}$

$$\begin{aligned}
\bot \ op \ rest &= \ \bot \\
rest \ op \ \bot &= \ \bot \\
v_1 \ op \ v_2 &= \ v_3 \ \ \texttt{where} \ v_i \ \notin \ \{\bot, \langle k \ ? \ V_h \ : \ V_l \rangle\} \\
v \ op \ \langle k \ ? \ rest_h \ : \ rest_l \rangle &= \langle k \ ? \ v \ op \ rest_h \ : \ v \ op \ rest_l \rangle \ \ \texttt{where} \ v \ \notin \ \{\bot, \langle k \ ? \ V_h \ : \ V_l \rangle\} \\
\langle k \ ? \ rest_h \ : \ rest_l \rangle \ op \ rest &= \langle k \ ? \ rest_h \ op \ rest \ : \ rest_l \ op \ rest \rangle
\end{aligned}$$

Figure 13: Faceted Expression Rules 2.

Declassify Function: $\boxed{declassify \ : \ Label \times Value \ \rightarrow \ Value}$

$$
\begin{aligned}
declassify(k, v) = \ & v \ \texttt{where} \ v \ \neq \ \langle k \ \texttt{?} \ V_h \ \texttt{:} \ V_l \rangle \\
declassify(k_1, \langle k_2 \ \texttt{?} \ rest_h \ \texttt{:} \ rest_l \rangle) = \ & rest_h \ \texttt{where} \ k_1 = k_2 \\
declassify(k_1, \langle k_2 \ \texttt{?} \ rest_h \ \texttt{:} \ rest_l \rangle) = \ & \langle k_2 \ \texttt{?} \ declassify(k_1, rest_h) \ \texttt{:} \ declassify(k_1, rest_l) \rangle \\
& \texttt{where} \ k_1 \neq k_2
\end{aligned}
$$

Assignment: $\boxed{assign: \ Store \times PC \times Value \times Value \rightarrow \ Value}$

$$
\begin{aligned}
assign(\Sigma, pc, a, V) &= \Sigma[a := \langle \langle pc \ \texttt{?} \ V \ \texttt{:} \ \Sigma(a) \rangle \rangle] \\
assign(\Sigma, pc, \bot, V) &= \Sigma \\
assign(\Sigma, pc, \langle k \ \texttt{?} \ V_H \ \texttt{:} \ V_L \rangle, V) &= \Sigma' \ \texttt{where} \ \Sigma_1 \ = \ assign(\Sigma, pc \cup \{k\}, V_H, V) \\
& \qquad \texttt{and} \ \Sigma' \ = \ assign(\Sigma_1, pc \cup \{\overline{k}\}, V_L, V)
\end{aligned}
$$

Function Application: $\boxed{\Phi, \Sigma, (V_1 \ V_2)^{type} \downarrow\downarrow_{pc}^{func} \Phi', \Sigma', V'}$

$$
[\text{FA-BOT}] \qquad \frac{}{\Phi, \Sigma, (\bot \ V) \downarrow\downarrow_{pc}^{func} \Phi', \Sigma', V'}
$$

$$
[\text{FA-FUNC}] \qquad \frac{\Phi, \Sigma, \Theta[x := V], e \downarrow\downarrow_{pc}^{stmt} \Phi', \Sigma', V'}{\Phi, \Sigma, ((\lambda x.e, \Theta) \ V)^{func} \downarrow\downarrow_{pc}^{func} \Phi', \Sigma', V'}
$$

$$
[\text{FA-NEW-OBJ}] \qquad \frac{\Phi, \Sigma, \Theta[x := V], e \downarrow\downarrow_{pc}^{stmt} \Phi', \Sigma_1, V' \quad fresh \ a \quad a \notin domain(\Sigma_1) \quad \Sigma' = \Sigma_1[a := V']}{\Phi, \Sigma, ((\lambda x.e, \Theta) \ V)^{new} \downarrow\downarrow_{pc}^{func} \Phi', \Sigma', a}
$$

$$
[\text{FA-LEFT}] \qquad \frac{k \ \in pc \quad \Phi, \Sigma, (V_H \ V_2) \downarrow\downarrow_{pc}^{func} \Phi', \Sigma', V}{\Phi, \Sigma, (\langle k \ \texttt{?} \ V_H \ \texttt{:} \ V_L \rangle \ V_2) \downarrow\downarrow_{pc}^{func} \Phi', \Sigma', V}
$$

$$
[\text{FA-RIGHT}] \qquad \frac{k \ \notin pc \quad \Phi, \Sigma, (V_L \ V_2) \downarrow\downarrow_{pc}^{func} \Phi', \Sigma', V}{\Phi, \Sigma, (\langle k \ \texttt{?} \ V_H \ \texttt{:} \ V_L \rangle \ V_2) \downarrow\downarrow_{pc}^{func} \Phi', \Sigma', V}
$$

$$
[\text{FA-SPLIT}] \qquad \frac{\begin{array}{ll} k \ \notin pc & \Phi, \Sigma, (V_H \ V_2) \downarrow\downarrow_{pc \cup \{k\}}^{func} \Phi_1, \Sigma_1, V'_H \\ \overline{k} \ \notin pc & \Phi_1, \Sigma_1, (V_L \ V_2) \downarrow\downarrow_{pc \cup \{\overline{k}\}}^{func} \Phi', \Sigma', V'_L \end{array}}{\Phi, \Sigma, (\langle k \ \texttt{?} \ V_H \ \texttt{:} \ V_L \rangle \ V_2) \downarrow\downarrow_{pc}^{func} \Phi', \Sigma', \langle \langle k \ \texttt{?} \ V'_H \ \texttt{:} \ V'_L \rangle \rangle}
$$

Figure 14: Faceted Evaluation Semantics.

Based on the state of the program counter `pc`, evaluation rules as shown in Figure 12,13 and 14 evaluate an expression `e`, a store Σ, a file system Φ, and a substitution Θ into an actual value `V`, a possibly modified store Σ', and a modified file system Φ'.

$$\textit{Expression evaluation}: \ \Phi, \Sigma, \Theta, e \downarrow\downarrow_{pc}^{expr} \Phi', \Sigma', V$$

F-NEW-LABEL - The security label, which is the most important element in the faceted execution, can be created at runtime and passed as an argument. The operations $fresh\ a$ and $\Sigma[a := v]$ allocate a memory address a in the store Σ for a new label object. This rule evaluates the expression `new Label()` to a reference of the newly created security label.

F-CLASSIFY-INVALID - This rule evaluates the classification expression to an `undefined` value, if the expression e_1 does not evaluate to a security label.

F-CLASSIFY - This rule evaluates the expression e_1 to a security label k, returning a faceted value $\langle k\ ?\ V_h\ :\ V_l \rangle$. This rule allows the developers to specify the private facet and the public facet of the value.

F-DECLASSIFY-INVALID - This rule evaluates the declassification expression to an `undefined` value, if the expression e_1 does not evaluate to a security label.

F-DECLASSIFY - In contrast to [F-CLASSIFY], this rule evaluates the expression e_1 to a security label k and the expression e_2 to a possibly faceted value $\langle k\ ?\ V_h\ :\ V_l \rangle$. It declassifies the faceted value with respect to k, returning the private facet V_h of the value.

F-LEFT - This rule evaluates only the expression e_1, if the current execution branch depends on some sensitive information with respect to the security label k.

F-RIGHT - This rule evaluates only the expression e_2, if the current execution branch depends on some public information with respect to the security label k.

F-SPLIT - Using the program counter, this rule evaluates both the expressions e_1 and e_2, to V_1 and V_2, respectively. The result of operation $\langle\langle k \texttt{ ? } V_1' \texttt{ : } V_2'\rangle\rangle$ combines the value of two execution branches into a single faceted value.

F-NEW-CH - This rule simply allocates a memory address for a new channel object. The result of the evaluation thus returns the reference a to this security channel.

CHAPTER 5

Implementation

5.1 The Antlr4 Grammar of Faceted JS

Faceted JS interpreter supports basic JavaScript features, file input, command line interface, as well as faceted execution. The interpreter is implemented in Java and Antlr. Antlr is a powerful framework for parsing formal languages, widely used by many companies, including Apple, Twitter, and Oracle. Using Antlr, we can simplify the parsing and evaluation process of Faceted JS.

The grammar of this modified language shares many similarities with JavaScript, except that it has several runtime syntax to enforce information flow control. The `new Label()` expression creates a security label and the `new Channel()` expression creates a security channel. The `setSecurity` function classifies a value into a faceted value, while the `defacet` function declassifies a faceted value by removing the associated security label. For completeness, Appendix A presents the Antlr 4 grammar of Faceted JS.

5.2 Architecture of the interpreter

The implementation of Faceted JS interpreter consists of three parts: the formal description of the language; the abstract syntax tree (AST) builder; and the evaluation rules. Based on the grammatical structure of Faceted JS, the lexer breaks user input into tokens. The parser uses these tokens to build an Antlr parse tree, which is later transformed to our predefined AST structure using the syntax tree visitor pattern from Antlr. The interpreter evaluates the abstract syntax tree following the big-step operational semantics defined in chapter 4. During the evaluation process, all references are maintained in a store (Σ), while all file references and remote resources are managed in an I/O System (Φ).

The Faceted JS interpreter is composed of two major components: the parser and the executor. Figure 15 clearly describes the process of translating the source code of Faceted JS into a set of specific actions.

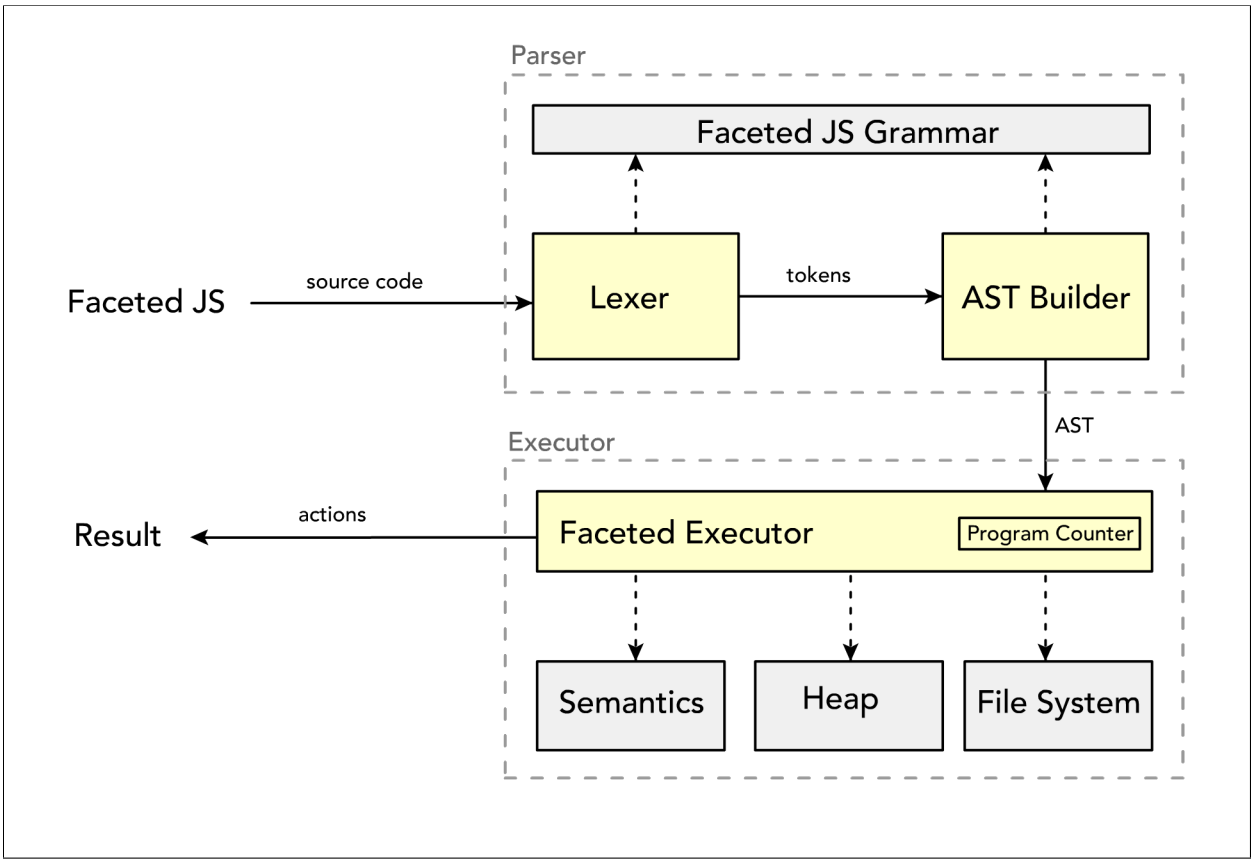

Figure 15: Structure of the Interpreter.

CHAPTER 6

Performance and Usages

6.1 Performance

Benchmark in milliseconds (ms)			
# security labels	Secure Multi-Execution		Faceted Execution
	Sequential	Concurrent	
0	3.3274	3.2903	3.3394
1	6.7151	3.3190	3.4772
2	13.1607	5.6400	3.4394
3	25.980	10.8941	3.5260
4	52.2259	22.1634	3.6480
5	109.4591	44.0636	3.9860
6	226.9655	86.6831	4.5178
7	467.1801	177.0852	6.2320
8	948.8010	357.3211	10.9784

Table 1: Performance of Secure Multi-Execution and Faceted Execution

In order to demonstrate the benefit of faceted execution, we implement three different versions of JavaScript interpreter, which simulate sequential secure multi-execution, concurrent secure multi-execution, and faceted execution. The test cases are performed on a MacBook Pro with 2.7Ghz Intel Dual-core i5 processor and 8 GB of DDR3 memory. Each test involves security labels, and channels, as well as all other programming constructs in Faceted JS. To understand the impact of security labels on performance, we perform nine different test cases, with the first test involving no security labels and the last test involving eight security labels.

The benchmark clearly demonstrates the impact of security labels across different approaches. Due to the simplicity of secure multi-execution, faceted execution is outperformed in the test that involves no security labels. Sequential secure multi-execution is relatively fast in the first test. The performance time suffers, however, as the number of security

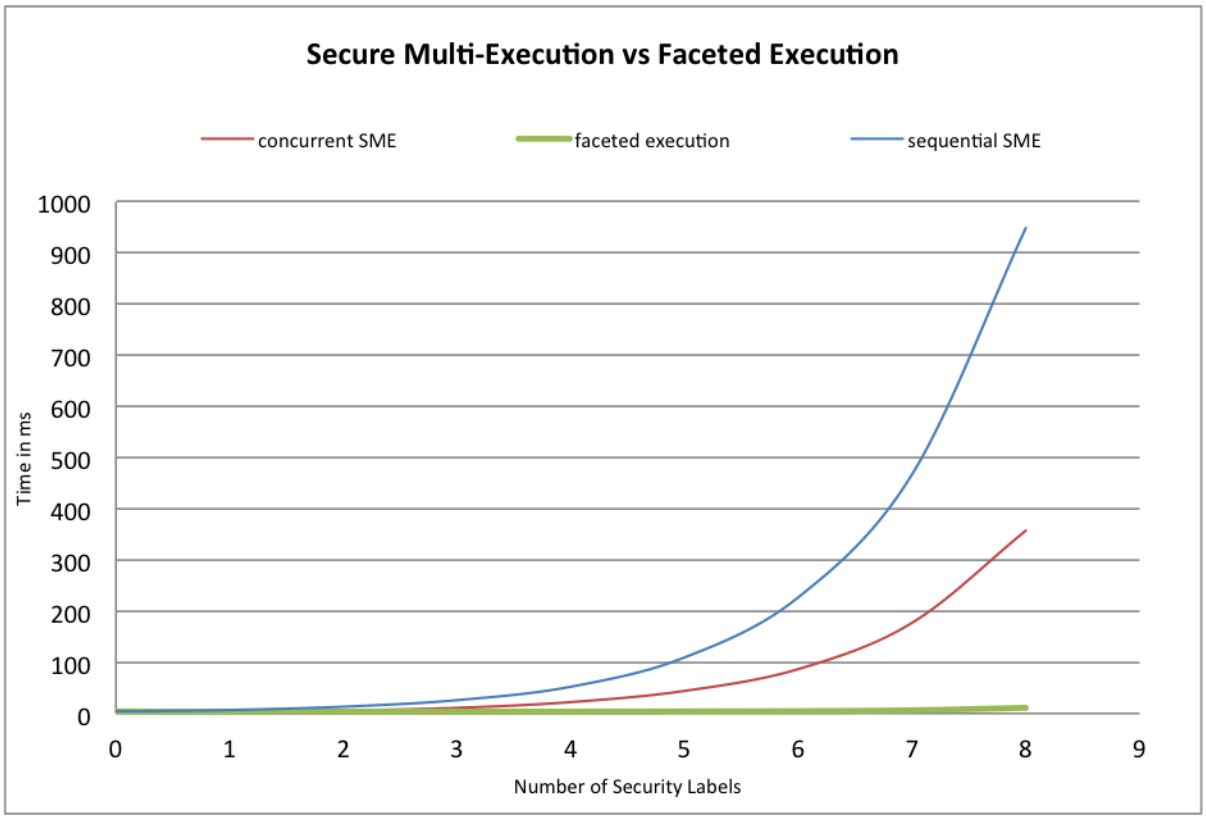

Figure 16: Performance of Faceted Execution.

labels increases. Despite the fact that concurrent secure multi-execution outperforms the faceted execution in the tests that involve only a few security labels, the benefit of multi-core processing cannot outweigh the disadvantages of executing highly redundant programs. As the number of security labels increases, the performance time of concurrent secure multi-execution still increases exponentially. Figure 16 shows that the performance time of faceted execution increases relatively slowly, with the introduction of each new security label. Faceted execution utilizes faceted values, to keep track of the sensitive information across different execution branches, which greatly reduces the overhead, achieving better performance in most real-world situations.

6.2 Real-World Usages

6.2.1 XSS Protection

Consider a cross-site scripting (XSS) targeting an e-commerce application, as in the example below. This injection attack works by wrapping the original pay function with malicious code, which leaks the payment information by simply requesting an image from a compromised website.

```
1  <input type="text" id="creditcard-input" fjs-security="high" fjs-channel="
       trusted.com"/>
2  <input type="text" id="securitycode-input" fjs-security="high" fjs-channel
       ="trusted.com"/>
3  ...
4  <button onclick="pay()">Place Order</button>
5
6  <script type="text/javascript">
7  var pay = function() { ... };
8  var pos = document.location.href.indexOf("username=") + 9;
9  var username = document.location.href.substring(pos);
10 document.getElementById("message").innerHTML = "Welcome, " + username;
11 </script>
12
13 <p id="message"></p>
```

Very often, victims are tricked by an attacker into clicking on a link (e.g., `http://www.example.com/?id=<script>...</script>`), thereby injecting malicious code into the dom element during runtime. The following example shows the result of such an injection:

```
1  <input type="text" id="creditcard-input" fjs-security="high" fjs-channel="
       trusted.com"/>
2  <input type="text" id="securitycode-input" fjs-security="high" fjs-channel
```

```
   ="trusted.com"/>
...
<button onclick="pay()">Place Order</button>

<script type="text/javascript">
var pay = function() { ... };
var pos = document.location.href.indexOf("username=") + 9;
var username = document.location.href.substring(pos);
document.getElementById("message").innerHTML = "Welcome, " + username;
</script>

<p id="message">
Welcome Back,
<script>
var oldPay = pay;
pay = function(){
    // cc and code are secret
    var cc = document.getElementById("creditcard-input").text;
    var code = document.getElementById("securitycode-input").text;
    var flag = false;
    if (code > 500){
        flag = true; // implicit flow from code to flag
    }
    var url = "http://evil.com/image.jpg?cc=" + cc + "&flag=" + flag;
    document.getElementById("img").src = url;
    return oldPay();
}
</script>
</p>
```

The attribute `fjs-security` specifies the security level of the control com-

ponent, while the attribute `fjs-channel` specifies the trusted output channel. Considering `document.getElementById("creditcard-input").text` and `document.getElementById("securitycode-input").text` as high-credential inputs, without any input validations, the credit card number and the security code might be easily leaked. With faceted execution, explicit and implicit flows are properly handled via faceted values. `evil.com` sees only the public facet of the credit card number and security code. Such a mechanism can be built into the interpreters, providing privacy without the need for implementing excessive validation mechanisms on the client side. This architecture-based approach not only reduces the development time of a software application. This approach also prevents, in an elegant way, private data from leaking.

6.2.2 Secure Declassification

Suppose we have a simple authentication system running in a server-side JavaScript environment. The application simply checks the user input against the system password during runtime, notifying public users about the correctness of the password.

```
1  var publicChannel = new Channel("*");
2  function pwTester(){
3      // hidden within the pwTester function
4    var k = new Label();
5
6    this.makePassword = function(p){
7          return setSecurity(k, p);
8    };
9    this.hash = function(p){
10         var h = md5(p);
11         // Access to k grants permission to deconstruct p
12         return defacet(k, h);
13   };
```

```
14      this.md5 = function(pwd){
15          ...
16          return hash;
17      }
18  }
19  var pwt = new pwTester();
20  var pwd = pwt.makePassword("secret");
21
22  if (pwt.hash(pwd) === md5(input)){
23      publicChannel.write("Login successfully");
24  }else{
25      publicChannel.write("Password mismatch !");
26  }
27
28  // injection 1
29  var password = defacet(k, pwd);
30  publicChannel.write(password);
31
32  // injection 2
33  pwt.getPassword = function(p){
34      return defacet(k, h);
35  }
```

The injections attempt to declassify the password to the public channel. However, the reference `k` is undefined outside the object `pwTester`. Only the method `makePassword` and `hash` hold the reference. Because the injected code cannot access the internal elements of `pwTester`, the password cannot be declassified properly. As a result, public observers see only an undefined value, instead of the actual password.

CHAPTER 7

Conclusion

Current technology relies on intensive filtering and encoding mechanisms, to protect sensitive data from being leaked to the public. However, it is quite difficult to cover every use case in a large system. This may allow an attacker to evade control, causing the leak of sensitive data. This paper introduces Faceted JS, a modified JavaScript language that supports first-class security labels, output channels, and classification functions. These security concepts enforce information flow control from an architecture perspective, providing flexibility and security, as well as offering a performance benefit.

Although non-interference information flow controls provide strong security guarantees, this is too strong for many practical uses. For this reason, we introduce a robust declassification mechanism that allows the system to declassify sensitive data in a controlled manner. A mechanism utilizes the concept of unforgeable reference from the object capability model [7], thereby enforcing security through the construction of a security label. Finally, we describe the usage of this security model and demonstrate that the performance of faceted execution in Faceted JS has a clear advantage over a secure multi-execution approach.

LIST OF REFERENCES

[1] Thomas H Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. *ACM Sigplan Notices*, 44(8):20--31, 2009.

[2] Thomas H Austin and Cormac Flanagan. Permissive dynamic information flow analysis. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, page 3. ACM, 2010.

[3] Thomas H Austin and Cormac Flanagan. Multiple facets for dynamic information flow. *ACM SIGPLAN Notices*, 47(1):165--178, 2012.

[4] Thomas H Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. In *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 15--26. ACM, 2013.

[5] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236--243, 1976.

[6] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504--513, 1977.

[7] Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143--155, 1966.

[8] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 109--124. IEEE, 2010.

[9] Jeffrey Stewart Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143--147, 1974.

[10] Mark S Granovetter. The strength of weak ties. *American journal of sociology*, pages 1360--1380, 1973.

[11] Sarah Kuranda. New federal budget proposal raises government security spending, ups opportunity for vars - page: 1 | crn. `http://www.crn.com/news/security/300079648/new-federal-budget-proposal-raises-government-security-spending-ups-opportunity-for-vars.htm`, Feb 2016.

[12] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. Capability myths demolished. Technical report, Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals, 2003.

[13] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5--19, 2003.

[14] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517--548, 2009.

[15] Tara Seals. 87% of open-source vulns are xss and sql injection - infosecurity magazine. `http://www.infosecurity-magazine.com/news/87-of-opensource-vulns-are-xss-and/`, Feb 2016.

[16] Stephan Arthur Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.

APPENDIX A

Antlr4 Grammar of Faceted JS

```
grammar FacetedJavaScript;

@header {
    package edu.sjsu.facetedJS.interpreter.parser;
}

// Reserved words
IF         : 'if' ;
ELSE       : 'else' ;
DO         : 'do' ;
WHILE      : 'while' ;
FOR        : 'for' ;
LABEL      : 'Label' ;
WRTCHL     : 'writeToChannel' ;
SECURE     : 'setSecurity' ;
CREATECHL  : 'createChannel' ;
ALERT      : 'alert' ;
WINOPEN    : 'window.open' ;
FUNCTION   : 'function' ;
VAR        : 'var' ;
PRINT      : 'system.log' ;
RETURN     : 'return' ;
NEW        : 'new' ;
DEFACET    : 'defacet' ;
PUSH       : 'push' ;
LENGTH     : 'length' ;
MD5        : 'md5' ;
```

```
28
29 // Literals
30 INT        : [1-9][0-9]* | '0' ;
31 BOOL       : 'true' | 'false' ;
32 STRING     : '"' (~[\\"\r\n])* '"';
33 NULL       : 'null' ;
34 UNDEFINED  : 'undefined' ;
35
36 // Identifiers
37 ID         : [a-zA-Z_] [a-zA-Z0-9_]* ;
38
39 // Arithm Symbols
40 MUL        : '*' ;
41 DIV        : '/' ;
42 ADD        : '+' ;
43 SUB        : '-' ;
44 MOD        : '%' ;
45
46 // Logical Symbols
47 GT         : '>' ;
48 GE         : '>=' ;
49 LT         : '<' ;
50 LE         : '<=' ;
51 EQ         : '==' ;
52 NEQ        : '!=' ;
53 S_EQ       : '===' ;
54 S_NEQ      : '!==' ;
55 AND        : '&&' ;
56 OR         : '||' ;
57
58 // Assignment Symbols
```

```
59 INC       : '++' ;
60 DEC       : '--' ;
61 ASSIGN_REG : '=' ;
62 ASSIGN_ADD : '+=' ;
63 ASSIGN_SUB : '-=' ;
64 ASSIGN_MUL : '*=' ;
65 ASSIGN_DIV : '/=' ;
66 SEPARATOR : ';' ;
67
68 // Whitespace and comments
69 NEWLINE   : '\r'? '\n' -> skip ;
70 BLOCK_COMMENT : '/*' .*? '*/' -> skip ;
71 LINE_COMMENT  : '//' ~[\n\r]* -> skip ;
72 WS            : [ \t]+ -> skip ; // ignore whitespace
73
74 // Expressions
75 expr: expr args
76     | DEFACET '(' expr ',' expr ')'
77     | SECURE '(' expr ',' expr (',' expr)? ')'
78     | MD5 '(' expr ')'
79     | FUNCTION params '{' stat* '}'
80     | expr '[' expr ']'
81     | expr '.' ID
82     | expr '.' PUSH '(' expr ')'
83     | expr '.' LENGTH
84     | op=( '+' | '-') expr
85     | op=( INC | DEC ) expr
86     | expr op=( INC | DEC )
87     | NEW expr
88     | NEW LABEL '(' ')'
89     | expr op=( '*' | '/' | '%' ) expr
```

```
90      | expr op=( '+' | '-' ) expr
91      | expr op=( '<' | '<=' | '>' | '>=' | '==' | '!=' | '===' | '!==')
            expr
92      | expr op=( '&&' | '||') expr
93      | VAR ID op=( '=' | '+=' | '-=' | '*=' | '/=' ) expr
94      | expr op=( '=' | '+=' | '-=' | '*=' | '/=' ) expr
95      | array
96      | object
97      | INT
98      | BOOL
99      | STRING
100     | NULL
101     | UNDEFINED
102     | ID
103     | '(' expr ')'
104     ;
105
106 // Paring rules
107 prog: stat+ ;
108
109 // Statements
110 stat: expr SEPARATOR                                    # bareExpr
111     | FUNCTION ID params '{' stat* '}'                  # namedFunction
112     | IF '(' expr ')' block ELSE block                  # ifThenElse
113     | IF '(' expr ')' block                             # ifThen
114     | WHILE '(' expr ')' block                          # while
115     | DO block WHILE '(' expr ')'                       # doWhile
116     | FOR '(' expr ';' expr ';' expr ')' block          # for
117     | WRTCHL '(' expr ',' expr ')' SEPARATOR            # writeChl
118     | PRINT '(' expr ')' SEPARATOR                      # printExpr
119     | ALERT '(' expr ')' SEPARATOR                      # alert
```

```
120     | WINOPEN '(' expr ')' SEPARATOR                    # winOpen
121     | RETURN expr SEPARATOR                             # returnExpr
122     | SEPARATOR                                         # blank
123     ;
124
125 // Compound Structures
126 perms : '(' ')'
127       | '(' STRING (',' STRING)* ')'
128       ;
129 block : '{' stat* '}'                                   # fullBlock
130       | stat                                            # simpBlock
131       ;
132 params : '(' ')'
133        | '(' ID (',' ID)* ')'
134        ;
135 args : '(' ')'
136      | '(' expr (',' expr)* ')'
137      ;
138 array : '[' expr (',' expr)* ']'
139       | '[' ']'
140       ;
141 object : '{' pair (',' pair)* '}'                       # fullObj
142        | '{' '}'                                        # emptObj
143        ;
144 pair : ID ':' expr                                      # keyPair
145     ;
```

APPENDIX B

Test Cases

```
1
2  k1 = new Label();
3  k2 = new Label();
4  k3 = new Label();
5  k4 = new Label();
6  k5 = new Label();
7  k6 = new Label();
8  k7 = new Label();
9  k8 = new Label();
10 s1 = setSecurity(s1,true, false);
11 s2 = setSecurity(s1,true, false);
12 s3 = setSecurity(s1,true, false);
13 s4 = setSecurity(s1,true, false);
14 s5 = setSecurity(s1,true, false);
15 s8 = setSecurity(s1,true, false);
16 s7 = setSecurity(s1,true, false);
17 s6 = setSecurity(s1,true, false);
18
19 var BenchMark = function() {
20     this.init = function() {
21         out = 0; out1 = 0; out2 = 0; out3 = 0;
22     };
23 };
24
25 var benchmark = new BenchMark();
26
27 benchmark.condTest = function() {
```

```
28      if (s1) {
29        if (s2) {
30          if (s3) { out1 = 1; } else { out = 2; }
31        } else {
32          if (s3) { out1 = 3; } else { out = 4; }
33        }
34      } else {
35        if (s2) {
36          if (s3) { out1 = 5;} else { out = 6; }
37        } else {
38          if (s3) { out1 = 7; } else { out = 8; }
39        }
40      }
41      if (s4) {
42        if (s5) {
43          if (s6) { out2 = 9; } else { out2 = 10; }
44        } else {
45          if (s6) { out2 = 11; } else { out2 = 12; }
46        }
47      } else {
48        if (s5) {
49          if (s6) { out2 = 13; } else { out2 = 14; }
50        } else {
51          if (s6) { out2 = 15; } else { out2 = 16; }
52        }
53      }
54      if (s7) {
55        if (s8) { out3 = 17; } else { out3 = 18; }
56      } else {
57        if (s8) { out3 = 19; } else { out3 = 20; }
58      }
```

```
59      out = out1 + out2 + out3;
60  };
61
62  benchmark.logicTest = function() {
63      var i = 0;
64      var j = 0;
65      var z = 0;
66      if (i === 0 && out > 10 && out < 20) { i++; }
67      if (j === 0 && (out <= 10 || out >= 20)) { j++; }
68      z = i * j;
69  };
70
71  benchmark.opTest = function() {
72      var i = out;
73      i = ++i; i++;
74      --i; i--;
75      i = i++ + i--;
76      i = --i + ++i;
77      var j = out;
78      j += i + 101; j -= i + 202; j *= i + 303;
79  };
80
81  benchmark.loopTest = function() {
82      var max = out / 4;
83      var i = 0; var j = 0; var z = 0;
84      for ( i = 0; i < max; i ++) { for ( j = 0; j < max; j++ ){ z++; } }
85      i = 0; j = 0;
86      while (i < max) { while (j < max) { z -= i; j++; } i++; }
87      i = 0; j = 0;
88      do { i++; do { j++; } while (j < max) } while ( i < max)
89  };
```

```
90
91  benchmark.funcTest = function() {
92      function fact (i, acc){
93          if (i === 1){
94              return acc;
95          }
96          return fact(i - 1, i * acc);
97      }
98      var j = fact(3, out);
99      var sum = function(i){
100         if (i > 0){
101             return i + sum(i-1);
102         }
103     };
104     var z = sum(out);
105 };
106
107 benchmark.arrTest = function() {
108     var i = out;
109     var arr = [1,2,3,4,5,6];
110     for (idx = 0; idx < arr.length; idx++){
111         i += arr[idx];
112     }
113 };
114
115 benchmark.declsTest = function() {
116     var d1 = defacet(k1,out);
117     var d2 = defacet(k2,out);
118     var d3 = defacet(k3,out);
119     var d4 = defacet(k4,out);
120     var d5 = defacet(k5,out);
```

```
121     var d6 = defacet(k6,out);
122     var d7 = defacet(k7,out);
123     var d8 = defacet(k8,out);
124     d = defacet(k8,
125       defacet(k7,
126         defacet(k6,
127           defacet(k5,
128             defacet(k4,
129               defacet(k3,
130                 defacet(k2,
131                   defacet(k1,out))))))));
132 };
133
134 benchmark.init();
135 benchmark.condTest();
136 benchmark.logicTest();
137 benchmark.opTest();
138 benchmark.loopTest();
139 benchmark.funcTest();
140 benchmark.arrTest();
141 benchmark.declsTest();
```