Spring 5-16-2019

Declassification of Faceted Values in JavaScript

Shreya Gangishetty
San Jose State University

Declassification of Faceted Values in JavaScript

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Shreya Gangishetty

May 2019

The Designated Project Committee Approves the Project Titled

Declassification of Faceted Values in JavaScript

by

Shreya Gangishetty

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2019

ABSTRACT

Declassification of Faceted Values in JavaScript

by Shreya Gangishetty

This research addresses the issues with protecting sensitive information at the language level using information flow control mechanisms (IFC). Most of the IFC mechanisms face the challenge of releasing sensitive information in a restricted or limited manner. This research uses faceted values, an IFC mechanism that has shown promising flexibility for downgrading the confidential information in a secure manner, also called declassification.

In this project, we introduce the concept of first-class labels to simplify the declassification of faceted values. To validate the utility of our approach we show how the combination of faceted values and first-class labels can build various declassification mechanisms.

***Keywords*- Data confidentiality, Declassification, Downgrading policies, Dimensions of declassification**

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Thomas Austin for continuously motivating me and guiding me in the right path. I would also like to thank Professor Fabio Di Troia and Professor Ben Reed for their constant support in completing the project.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

As technology is growing continuously, many third-party applications are also increasing. These applications may misuse sensitive information like social security numbers, passwords, credit cards, or browser history. Providing security to such information on the internet is a challenging task. There is a risk for data confidentiality with data leakage. Due to data leakage, 191 million voters of US [1] were affected.

Information flow control enforces the rules and policies to the flow of data within the program or when the data is sent to a third party application [2]. It can secure sensitive information by imposing rules within the architecture to prevent data leakage from trusted sources to untrusted sources.

Several IFC mechanisms were introduced to ensure the confidentiality of data and to provide non-interference [3][4]. Some of the mechanisms such as permissive-upgrade [4] and no-sensitive-upgrade [5] guarantees non-interference property. However, these techniques get stuck when data flows from a high level to a low level. Often relaxation of non-interference property in secure conditions is needed in the real-world. For example, sensitive information like credit card number (high level) needs to be disclosed partially to validate a transaction. Often the last four digits of a credit card are displayed to validate the payment. In this scenario, non-interference property is clearly violated. But, this is necessary for a transaction to be successful. The relaxation of the non-interference property by enforcing security rules and policies is termed as declassification [6].

Determining whether it is safe to release sensitive information to a lower level and to structure the rules is a major challenge in the declassification of data [7]. Some of the dimensions to release information without compromising security are 'what', 'when', 'where', and 'who' [8]. Wing [7] has shown that faceted values are flexible to downgrade the non-interference property. This project focuses on scenarios where there is a need for declassification and how it can be achieved using a prominent IFC mechanism - faceted values (faceted evaluation) [9].

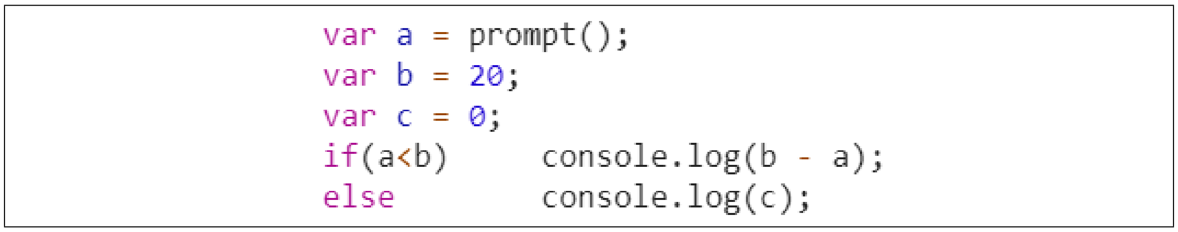
The project is organized as follows: Chapter 2 gives a background on information flow control, types and techniques of information flow control; Chapter 3 describes the faceted language with the support of faceted values along with its operational semantics; Chapter 4 describes declassification and various methods of releasing sensitive information and dimensions; Chapter 5 shows an implementation for faceted language with support for dynamic label creation, faceted values and declassification, and examples of dimensions of declassification using faceted values in JavaScript; Chapter 6 discusses the conclusion and future scope of this project.

CHAPTER 2

Background

2.1 Information Flow

Information flow is the transfer of information from a variable `var1` to another variable `var2` (`var1` → `var2`) in the program. Information about sensitive data might propagate to a lower level data. In Figure 1, the information of `a` and `b` is flowing through the conditional statement and this code leaks information about `a` and `b`. It is easy to infer the value of `a` based on this code snippet. If the output is `0` then it is understood that `a` is greater than or equal to `20`. Otherwise, the `20 - output` gives the value of `a`.

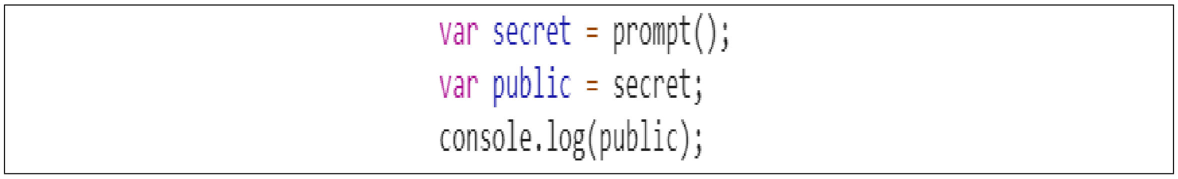


```
var a = prompt();
var b = 20;
var c = 0;
if(a<b)     console.log(b - a);
else        console.log(c);
```

Figure 1: Example of information flow

2.1.1 Explicit Flow

Explicit flow can be described as the direct flow of information from a high-security level to a low-security level[10]. Figure 2 represents explicit data leakage via direct assignment of a high-security level (`secret`) to a low-security level (`public`). The value of `secret` is assigned to `public` directly and the value of `secret` can be easily interpreted.

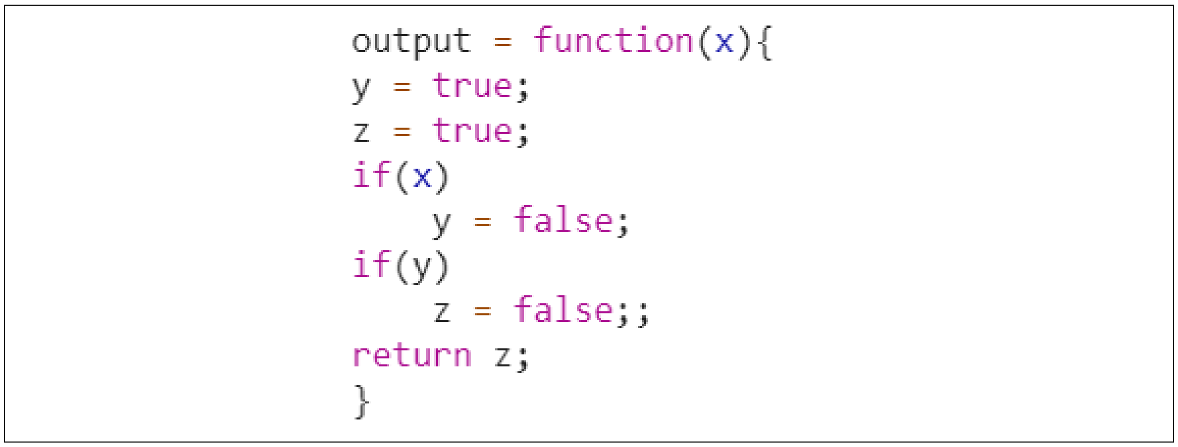


```
var secret = prompt();
var public = secret;
console.log(public);
```

Figure 2: Example of an explicit flow

2.1.2 Implicit Flow

In implicit flow, the high level information doesn't flow directly to a low level information but high level value can be implicitly understood[10]. This flow can leak partial information or complete information about the secret. In Figure 3, x is a high level information, the value of x is determined with implicit flow of data. For example, if the return value of function is false, that is `z = false`, this implies `y = true`, this implies `x = false`.



```
output = function(x){
y = true;
z = true;
if(x)
    y = false;
if(y)
    z = false;;
return z;
}
```

Figure 3: Example of an implicit flow

2.2 Overview of Information Flow Control (IFC)

With the exponential growth of data, the risk of protecting it is an unceasing problem. Sensitive personal data is often stored on computers. An easy and efficient method is needed to ensure data security. Information flow control (IFC) secures sensitive information by imposing information flow policies to prevent data flow from a private or higher security level to a public or a lower security level. IFC is a viable solution to protect data by preventing data leakage as it can track the outputs of a program. In IFC, the system monitors the flow of data from one place to another and regulates the data flow from a higher security level to a lower security level. IFC guarantees non-interference by preventing implicit and explicit flows [10]. IFC uses type-systems [11] and labels to secure data and enforces this via compile-time checking. The elementary IFC model has a security label - ‘high (h)’or ‘low (l)’associated with data where the system makes sure no data flows from high to low level. IFC can be helpful in securing the manipulation of high level and low level data on both client and server-side [10]. Hence, it can be used to protect data in web applications. IFC tracks the data flow in the web application that gives possible data leakage locations.

2.3 Information Flow Control Types

Several information flow control techniques were introduced to avoid data leakage. IFC techniques can be categorized into two types: Static IFC and Dynamic IFC

2.3.1 Static Information Flow Control

In static information flow control, the analysis of the source code is done during compile time and rejects programs that do not satisfy the rules and policies [10]. Even though this approach is effective and minimizes the run-time checking, it is restrictive for dynamically typed languages such as JavaScript [7].

2.3.2 Dynamic Information Flow Control

Static information flow analysis is not well-suited for dynamic scripting languages and does not guarantee secure information propagation. To overcome these limitations, dynamic IFC is introduced. In Dynamic information flow control, the policies are enforced during runtime to prevent implicit data leakage. Dynamic analysis is often slower than static information flow control in terms of performance but still guarantees non-interference [10] [7].

2.4 Information Flow Control Techniques

No-sensitive-upgrade (NSU) [12] [13], permissive-upgrade (PU) [14] [4], secure multi-execution (SME) [15], and faceted evaluation (FE) [5] are some of the mechanisms of dynamic information flow control which deal with implicit flows. All these mechanisms guarantee non-interference property (termination insensitive non-interference TINI) which is the private data does not flow through the public data.

Even though NSU and PU guarantees TINI, both of them halt execution. The execution gets stuck when subtle implicit flows are present in the code to avoid any sensitive data leakage [9]. This abrupt termination is not due to a web application violation, or the rules and policies defined, but it is due to the limitation of the mechanism to track implicit flows. Therefore, in some cases even valid programs are rejected in dynamic analysis. Therefore, an approach is needed that doesn't terminate the program when it encounters the unsafe implicit flows, instead it should show some counterfeit data. The issue with this approach is that the output might be inconsistent with the standard language semantics.

Devriese and Piessens [15] implemented a mechanism for IFC called secure multi-execution (SME) in which the program is divided into multiple copies and each copy is associated with a security level. In SME, all the copies are executed independently, hence giving the non-interference property. However, the problem with this approach is as two copies of the program are executed for each principal, that is it executes about 2^n copies of the same program for n principals. Wing [7] has explained that since a program needs to run multiple times in SME, the computation is increased by a huge margin thereby decreasing the performance. Additionally, as the copies are executed independently, this provides non-interference property i.e, making it impossible to get the original data. But in the real-world, there is a necessity to release some of the sensitive information.

Austin and Flanagan [9] introduced faceted evaluation (FE) using faceted value data structure to overcome the issue with SME. Faceted values represent multiple states for a value at various security levels which guarantees non-interference. A faceted value is a pair of two raw values which contain the private data and public data. In faceted evaluation, a single process can mimic the two processes that were needed in SME. The benefit of this mechanism is when two raw values are similar, the mechanism combines the two executions into a single execution which reduces overhead. This mechanism can be used for n principals or values rather than only two. This mechanism also guarantees termination insensitive non-interference property(TINI) partly along with avoiding stuck executions. The faster performance of FE over SME is proven by Wing [7].

2.5 Declassification for Dynamic Information Flow Control

In practical scenarios non-interference is not suitable. Some amount of information leak is often needed in real systems. For instance, when a person tries to login to an application, the correctness of the password is known to everyone. This leads to some leakage of data that is needed but the system could still be called secure. Often, the non-interference property needs to be relaxed in the real world scenarios. Downgrading the non-interference property to make the confidential data as a public data in a secured and controlled manner is called declassification of data. For example, in the password scenario, a hashing algorithm can be used to encrypt data with a key that is not known to public. A controlled way of declassifying would be to release only the details about the hash of the password instead of the password itself.

It is difficult to declassify data using SME [9] [7]. The two processes in SME needs to be coordinated, which again reintroduces the stuck evaluation and termination channel. In contrast, it is easy to declassify a faceted value. A single value contains both private and public levels that can be restructured easily to move the data from one level to another level supporting declassification [5].

The rules for determining the restructuring is challenging as declassifying without any restrictions will eventually lead to no security guarantee. Declassification can be broadly classified into dimensions. If dynamic flow control methods such as faceted values can incorporate all the dimensions of declassification, this concept would become state of the art in the field of data security using IFC [8]. This research focuses on incorporating these dimensions in real life scenarios and test the feasibility of implementing all the dimensions. This research also studies the effectiveness of faceted evaluation in releasing the sensitive information when its inevitable.

CHAPTER 3

Faceted Language

This project combines faceted value data structure [16] and native programming language features to construct a Faceted Language (FL). FL supports the native features of a programming language and faceted values data structure. This language creates first-class labels which are created dynamically when needed. These labels are associated with a public and a private values which are wrapped in a faceted value. In this context, labels serve a role similar to object capabilities [17], in that labels grant the *authority* to defacet faceted values.

JavaScript is most widely used in client-side and server-side scripting. It supports dynamic typing, and first-class functions. This language is implemented in JavaScript as the previous work has shown promising support of faceted values with JavaScript [9][16].

3.1 Faceted Values

A faceted value has a security label, a higher level information and a lower level information [16].

`Syntax: < label ? privateValue : publicValue >`

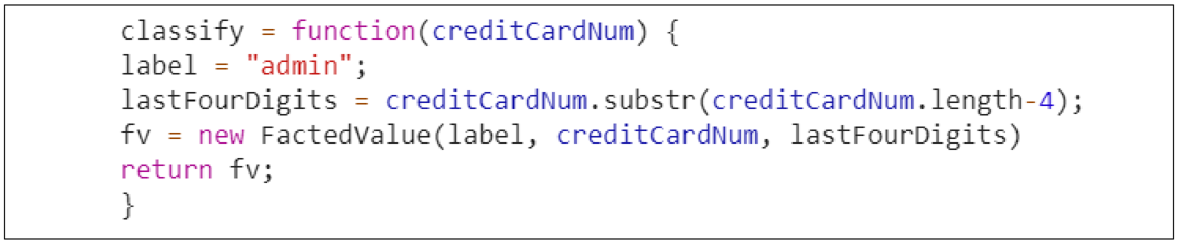
The above syntax represents faceted value. '`label`' is a security label that represents the access privilege of a user. '`privateValue`' represents a higher level data. '`publicValue`' represents a lower level data.The syntax of faceted values is similar to that of a ternary operator. While a ternary operator is an expression, faceted value is a data structure. Based on the security label evaluation, each faceted value evaluates to either a lower level data or a higher level data.

Faceted values can be nested. A single value or variable has multiple facets based on the security labels associated with it.

Example:`<label1 ? <label2 ? high : low2 > : low1>`

3.1.1 Classification of data

Classification is wrapping of sensitive data and its associated security labels in a faceted value. Let's consider a scenario of credit card number where admin can view all the digits of credit card number and others can view just the last four digits. Figure 4 represents classify function. For easier understanding, this use-case is implemented in JavaScript instead of FL. This can also be implemented in FL. Faceted value returned in this function can be represented as `fv = <label ? creditCardNum: lastFourDigits>`

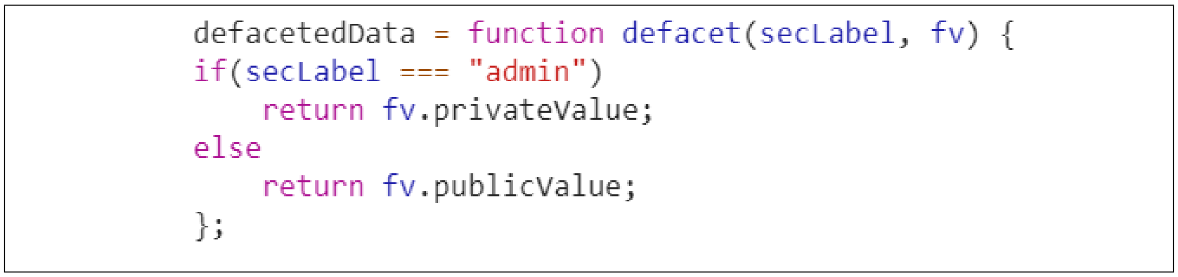


```
classify = function(creditCardNum) {
label = "admin";
lastFourDigits = creditCardNum.substr(creditCardNum.length-4);
fv = new FactedValue(label, creditCardNum, lastFourDigits)
return fv;
}
```

Figure 4: Example implementation of classification of data

3.1.2 Defacet of data

Defacet is unwrapping the faceted value with the help of security labels. Figure 5 represents the defacet function that takes label and a faceted value as the input and returns either a private value or a public value. If an admin wants to see the credit card number, upon calling the defacet function with admin label displays the entire credit card number. For example, `defacet("admin", fv)` will return the credit card number and `defacet("user", fv)` will return the last four digits.



```
defacetedData = function defacet(secLabel, fv) {
if(secLabel === "admin")
    return fv.privateValue;
else
    return fv.publicValue;
};
```

Figure 5: Example implementation of defacet function in JavaScript

3.2 Grammar and Semantics for Faceted Language

3.2.1 Grammar

Figure 6 represents the grammar for FL. This language contains expressions `(e)` and values `(v)`. This language supports boolean values, integer values, strings, faceted values, and labels. The expressions `(e)` of this language are similar to most of the imperative language features such as variables, values, assignment operators, conditional expressions, binary operators, and function (lambdas) application. These features are helpful in implementing functional programming aspect of JavaScript.

This language supports additional expressions for built-in support of faceted values. The `createLabel` expression is used to create a new label dynamically for the faceted values. The developers has the control over this label. Developers can assign these labels to users based on their access privileges. This language also supports classification and declassification of data for boxing and unboxing the faceted values with dynamically created labels.

$e ::=$		*Expressions*
	x	variables
	v	values
	$x := e$	assignment
	`if` e `then` e `else` e	conditional expressions
	$e\ e$	function application
	`binop` (e, e)	binary operators
	`createLabel()`	creates a label dynamically
	`classify`(e, e, e)	classify as faceted value
	`defacet`(e, e)	defacets an expression
binop $::=$	$+ \mid - \mid * \mid / \mid > \mid >= \mid < \mid <=$	Binary Operators
logicalop $::=$	$\&\& \mid \|$	Logical Operators
$v ::=$		*Values*
	b	Boolean
	f	Faceted value
	i	Integer
	s	String
	l	Label
	$\lambda x.e$	Function

Figure 6: The Faceted Language

3.2.2 Operational Semantics

The runtime behavior of any programming language is described by operational semantics. Operational semantics are of two types: big-step and small-step semantics [18]. This project formulates evaluation rules of FL using big-step operational semantics. Figure 7 shows all the evaluation rules followed to delevop this language.

Evaluation Rules: $\boxed{e, \sigma \Downarrow_{pc} v, \sigma'}$

[BS-VAL]
$$\frac{}{v, \sigma \Downarrow_{pc} v, \sigma}$$

[BS-VAR]
$$\frac{}{x, \sigma \Downarrow_{pc} \sigma(x), \sigma}$$

[BS-ASSIGN]
$$\frac{e, \sigma \Downarrow_{pc} v, \sigma'}{x := e, \sigma \Downarrow_{pc} v, \sigma'[x := v]}$$

[BS-OP]
$$\frac{e_1, \sigma \Downarrow_{pc} v_1, \sigma_1 \qquad e_2, \sigma_1 \Downarrow_{pc} v_2, \sigma_2 \\ v \;=\; v_1 \; op \; v_2}{e_1 \; op \; e_2, \sigma \Downarrow_{pc} v, \sigma_2}$$

[BS-IFTRUE]
$$\frac{e, \sigma \Downarrow_{pc} true, \sigma_1 \qquad e_1, \sigma_1 \Downarrow_{pc} v_1, \sigma_2}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2, \sigma \Downarrow_{pc} v_1, \sigma_2}$$

[BS-IFFALSE]
$$\frac{e, \sigma \Downarrow_{pc} false, \sigma_1 \qquad e_2, \sigma_1 \Downarrow_{pc} v_2, \sigma_2}{\texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2, \sigma \Downarrow_{pc} v_2, \sigma_2}$$

[BS-APPLICATION]
$$\frac{e_1, \sigma \Downarrow_{pc} (\lambda x.e^{'}), \sigma_1 \qquad e_2, \sigma_1 \Downarrow_{pc} v, \sigma_2 \\ e'[x \longmapsto v], \sigma_2 \Downarrow_{pc} v', \sigma'}{e_1 \; e_2, \sigma \Downarrow_{pc} v', \sigma'}$$

[BS-LABEL]
$$\frac{v := new\ Symbol()}{createLabel, \sigma \Downarrow_{pc} v, \sigma}$$

[BS-CLASSIFY]
$$\frac{e_1, \sigma \Downarrow_{pc} k, \sigma_1 \qquad e_2, \sigma_1 \Downarrow_{pc} v_1, \sigma_2 \\ e_3, \sigma_2 \Downarrow_{pc} v_2, \sigma_3 \qquad fv := < k \; ? \; v_1 \; : \; v_2 >}{Classify \; e_1 \; e_2 \; e_3, \sigma \Downarrow_{pc} fv, \sigma_3}$$

[BS-DEFACET]
$$\frac{e_1, \sigma \Downarrow_{pc} l, \sigma_1 \qquad e_2, \sigma_1 \Downarrow_{pc} v, \sigma_2 \\ v_1 := defacet(l, v)}{Declassify \; e_1 \; e_2, \sigma \Downarrow_{pc} v_1, \sigma_2}$$

Figure 7: Big-Step Semantics for Faceted Language

e	*denotes*	*an expression*
x	*denotes*	*a variable*
σ	*denotes*	*a store for storing Data*
k	*denotes*	*label*
b	*denotes*	*a boolean value*
f	*denotes*	*afaceted value*
v	*denotes*	*a primitive value*
λx	*denotes*	*afunction*

Figure 8: Notations

The language uses program counter `pc` to effectively manage labels of faceted values.

BS-VAL

This big-step semantics represents that a value always evaluates to a value.

BS-VAR

`BS-VAR` big step semantics represents the variables of faceted language. The program checks if the variable exists in the store and returns its value. According to the evaluation rule, the value of the variable `x` is returned after the lookup of `x` in (σ) data store.

BS-ASSIGN

The expression `x := e` indicates that the value of `e` is assigned to a variable `x`. Firstly, the expression (`e`) is evaluated to a value type that the language supports and if the variable `x` is not present in the store it adds the variable along with its value to the store.

BS-OP

This represents the binary operation of the language. This takes two expressions as inputs and evaluates to its corresponding values. These two values are evaluated with its operator and the result is returned.

BS-IFTRUE and BS-IFFALSE

These semantics represents the `if else` conditional statements. If the expression `e1` evaluates to `true` then the expression `e2` is evaluated or else the expression `e3` is evaluated.

BS-APPLICATION

This evaluation rule is for function application. The expression e_1 evaluates to a function and the expression e_2 evaluates to a value that is consistent with the language. This value is the parameter for the function that can return any primitive value or a function.

BS-LABEL

A new label is created each time `createLabel` function is called. To implement the functionality of `label`, built-in JavaScript objects called Symbols are used. A new value gets generated every time `new Symbol()` is invoked. This implies that any two Symbols are always unequal.

BS-CLASSIFY

Classify takes a label, a private value, and its corresponding public value as the input and wraps these values in a faceted value. The expression `e1` always evaluates to a `label` and `e2`, `e3` evaluate to supported values of the FL.

BS-DEFACET

Declassify unboxes the faceted value and give a private value or a public value based on the access label. Declassify takes two expressions as inputs. The expression `e1` evaluates to a label and expression `e2` can evaluate to a faceted value or any other primitive values supported by the language. If `e2` evaluates to a value other than faceted value then the defacet returns the value without declassifying. If `e2` evaluates to a faceted value then the function declassifies the faceted value and returns a private or a public value based on the user scope and associated security labels.

CHAPTER 4
Declassification

The security properties confidentiality and integrity that are specified by information flow policies can be formalized as non-interference. According to this property the confidential data does not affect the public data [6][19]. Pure non-interference properties allow programs to flow from a low security level to a high security level but not vice-versa. In practice, following non-interference property is often not suitable for real-world scenarios. Often data needs to flow from a high security level to a lower security level by preserving the data confidentiality. For instance, average salary given by a company needs to be displayed from the salaries database which is sensitive for statistical purposes. In this scenario, the sensitive information (salary) is flowing from high security level to low security level [8]. Hence, downgrading of security policies in a controlled manner is often necessary. Downgrading is specification of information flow from a high level to a low level which is also known as declassification.

4.1 Declassification mechanisms

Declassification can be broadly classified into four axes or dimensions [8]. They are: *what* information is released, *where* in the system information is released, *who* releases the information, and *when* the information can be released.

4.1.1 Dimensions of Declassification

Every dimension satisfies the following criteria [8][20]:

- Information can flow from low to high level directly
- Information cannot flow from high to low level directly
- Information can flow from high to declassifier and declassifier to low level where declassifier restricts the information flow based on rules and policies

The following are the four dimensions of declassification [8]:

1. What information is released

This classifier is based on what kind of information is released. This dimension guarantees that there is only partial release of information. Selective or partial release can be specified on exactly what parts of the sensitive information could be released. For instance, based on context the of leaking SSN or password, the ‘what’classifier is password field or the SSN field and the constraints are hash of the password or the last four digits of SSN.

2. When the information can be released

This classifier is based on the temporal dimension of the information to be released. For example, bidding information of an auction can be released only when it is completed. In this classification, the secret information is leaked only if it can be executed in a non-polynomial time. There are three specifications for the when based dimension. They are:

- **Time-complexity based:** This policy states that information can be released only after a specified time.
- **Probabilistic:** This policy states that the if possibility of hacker being able to distinguish among the true values of secret in less than some constant value *epsilon* then the system is secure [8]. *epsilon* is a threshold value which can be chosen when a policy is defined.
- **Relative:** This policy explains about relating the time when declassification may occur and other actions that are being done in the system. As an example, only after the payment is confirmed, the payment information can be released(declassified).

3. Where the information is released

This classifier is based on the location where the information is released in the system. For example, the information can be released only in the conditional statements where the data is evaluated. In this dimension, if a part of the system is authorized to release the information it is assured that no other part can leak the information apart from the authorized part. There are two types of locality that could be possible based on the where dimension of information release.

- **Code locality:** Code locality are a set of policies that can explain possibilities of information leakage in the code.
- **Level locality:** Levels where information would flow in comparison to the security levels of the system are explained by these policies.

4. Who releases the information

This classifier is based on the access levels of the user. If a user has access to the information, he/she can release the data and is considered a valid use case. Information release of data is said to be safe if it is performed by owner who is exactly recorded in the data security label. It is important to specify 'who' controls the release of information. One can use security labels for explicit owner information. Myers and Liskov [21] proposed a combination model of information flow control and access control. This model is used in implementation of Java Jif compiler [22].

Zdancewic and Myers have proposed a technique called robust declassification [23] which ensures that attackers cannot misuse the information. This model assures that if a passive attacker is not able to identify where the secret data is altered in two memories then active attacker also cannot distinguish between those two memories.

CHAPTER 5

Implementation

This project mainly considers three views of writers. The language writer is responsible for writing the language and semantics for Faceted Language. The language writer has a better understanding of security principles than the other writers. The library writer is responsible for writing the libraries for wrapping and unwrapping the sensitive information using faceted values. The library writer is responsible for assigning the dynamically created security labels to sensitive information. The application writer or developer uses the functions given by library writer to classify or declassify sensitive information. The labels creation and usage are not known to application writer.

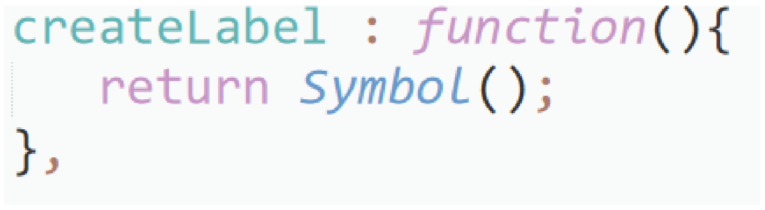
This section shows the implementation code for supporting faceted values in JavaScript (Node.js) and various scenarios where declassification is possible with faceted values. Library writers can use these declassification functions and faceted values to build their own rules and policies based on the context and situations.

5.1 Faceted Language

The below are the code snippets for supporting faceted values in JavaScript. This code follows runtime evaluation rules that are implemented in Chapter 3.

5.1.1 Dynamic label creation

Figure 9 represents the code snippet for `createLabel` functionality in JavaScript. `createLabel` creates a new `Symbol` each time it is invoked. The developer who has explicit access to the label can view the label and also assign these labels to users. The security label gets created during runtime and cannot be modified by any other function thus meeting the criteria to be called as first-class labels. Each high level data can have a single label or an array of labels associated with it.

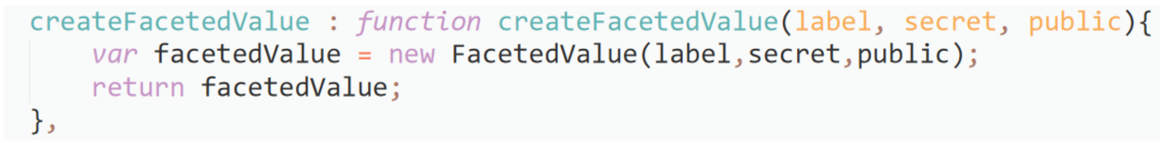


```
createLabel : function(){
    return Symbol();
},
```

Figure 9: Dynamic security label creation

5.1.2 Classification of private data into a faceted value

Figure 10 represents classification of data into faceted values using first-class labels. The function `createFacetedValue` takes secret and public as the input and wraps or classifies those values using a dynamically created label. `createFacetedValue` returns a faceted value with a newly created label and its associated secret and public data.

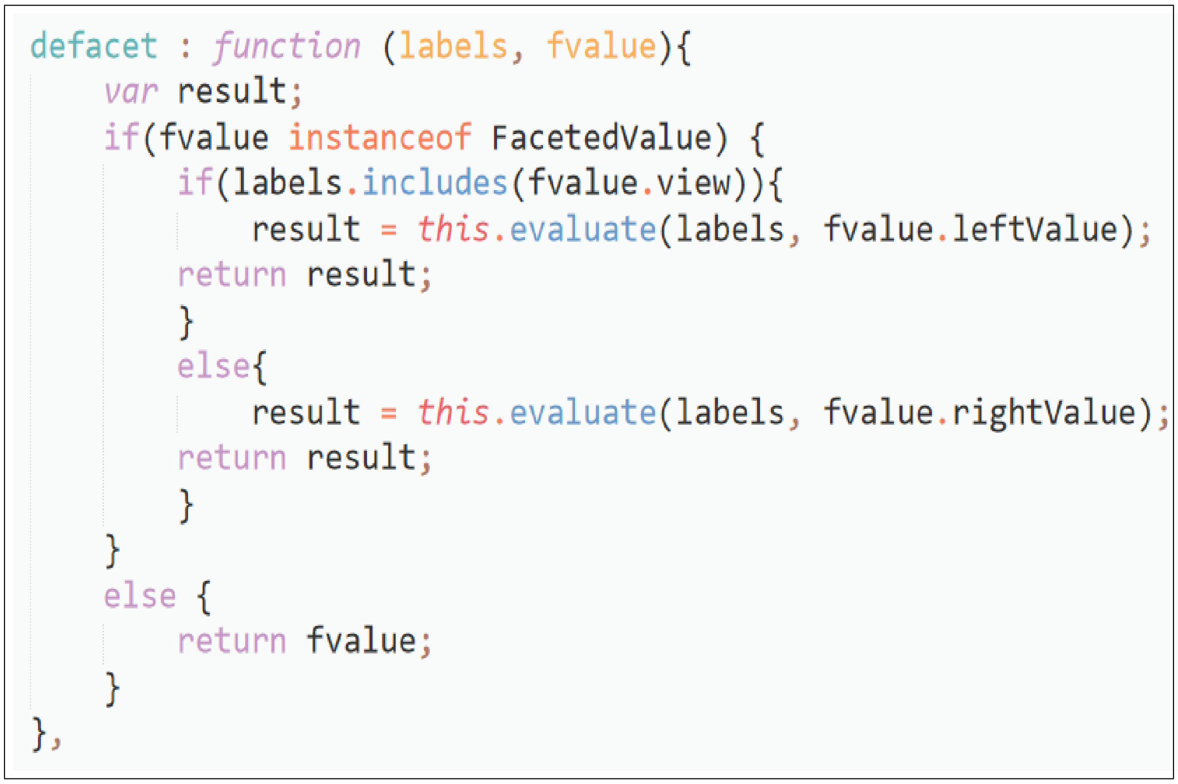


```
createFacetedValue : function createFacetedValue(label, secret, public){
    var facetedValue = new FacetedValue(label,secret,public);
    return facetedValue;
},
```

Figure 10: Classification of data using first-class labels

5.1.3 Defaceting of faceted value based on the labels

Figure 11 represents the function that returns a value to the users based on their access privileges on that faceted value. The caller gives a label or array of labels as the input to the defacet function along with the faceted value for which he/she wants to get the private information. Based on the given label or labels `defacet` function returns a private value (`leftValue`) or a public value (`rightValue`).



```
defacet : function (labels, fvalue){
    var result;
    if(fvalue instanceof FacetedValue) {
        if(labels.includes(fvalue.view)){
            result = this.evaluate(labels, fvalue.leftValue);
        return result;
        }
        else{
            result = this.evaluate(labels, fvalue.rightValue);
        return result;
        }
    }
    else {
        return fvalue;
    }
},
```

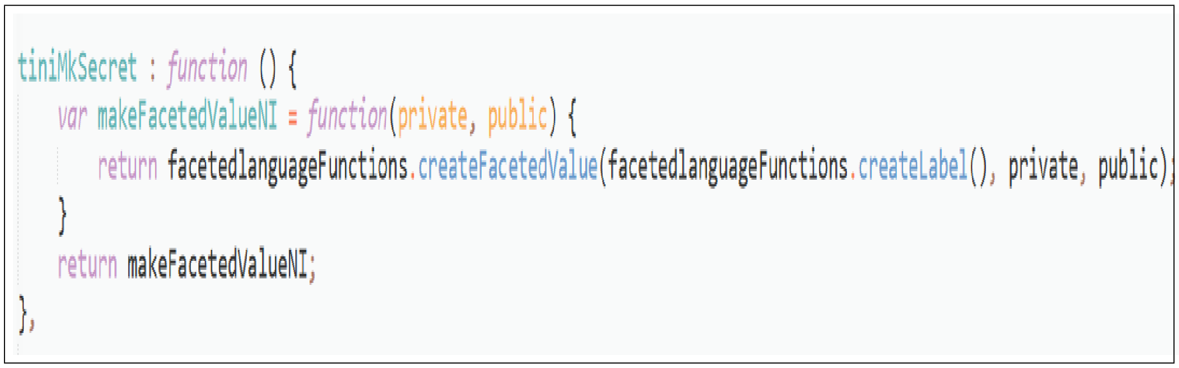
Figure 11: Defacet function

5.2 Implementation of declassification scenarios in faceted values

This section shows flexibility of faceted values by implementing some of the real-world scenarios for declassification.

5.2.1 Non-interference

Figure 12 shows how non-interference can be achieved in FV. The function `tiniMkSecret` takes private and public data as the input and returns a faceted value. In this function, there is no way in which declassification is possible for the faceted value created by `makeFacetedValueNI` as the scope of the label is within the function. Hence, in this example, we can create faceted values but we cannot get the private value back, thus guaranteeing non-interference.

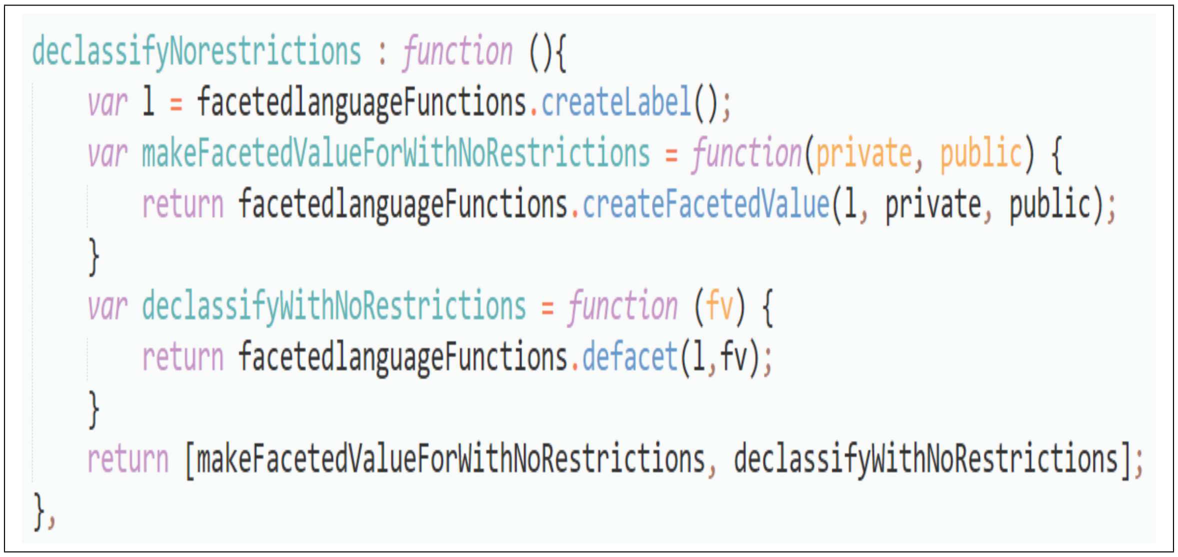


```
tiniMkSecret : function () {
    var makeFacetedValueNI = function(private, public) {
        return facetedlanguageFunctions.createFacetedValue(facetedlanguageFunctions.createLabel(), private, public)
    }
    return makeFacetedValueNI;
},
```

Figure 12: non-interference (No declassification)

5.2.2 Declassification with no restrictions

Figure 13 represents a function that returns the private data to anyone who has access to `declassifyWithNoRestrictions` function. Library writers can create a function that returns `makefacetedValueForWithNoRestrictions` and its corresponding declassification function `declassifyWithNoRestrictions` without any rules or policies. As seen from the code, FV are flexible to declassify without imposing any constraints.



```
declassifyNorestrictions : function (){
    var l = facetedlanguageFunctions.createLabel();
    var makeFacetedValueForWithNoRestrictions = function(private, public) {
        return facetedlanguageFunctions.createFacetedValue(l, private, public);
    }
    var declassifyWithNoRestrictions = function (fv) {
        return facetedlanguageFunctions.defacet(l,fv);
    }
    return [makeFacetedValueForWithNoRestrictions, declassifyWithNoRestrictions];
},
```

Figure 13: Declassification with no restrictions

5.2.3 Temporal dimension based declassification

Figure 14 represents an example of time based declassification. Auction scenario can be taken as an example for `when` based declassification. A policy that is relevant to this scenario is “release the bid information only after the auction is completed”. `timebasedMkSecret` takes private and public facets as input along with auction closing time. The developer can design a function as Figure14 and give access to the label to all the bidders. If anyone tries to access the bid data before the auction ends, they won’t be able to see the actual bid information. However, once the auction ends, everyone can view the bid information. Faceted values are functioning as required in this scenario as well. There are no changes to the language semantics to perform this type of declassification operation.

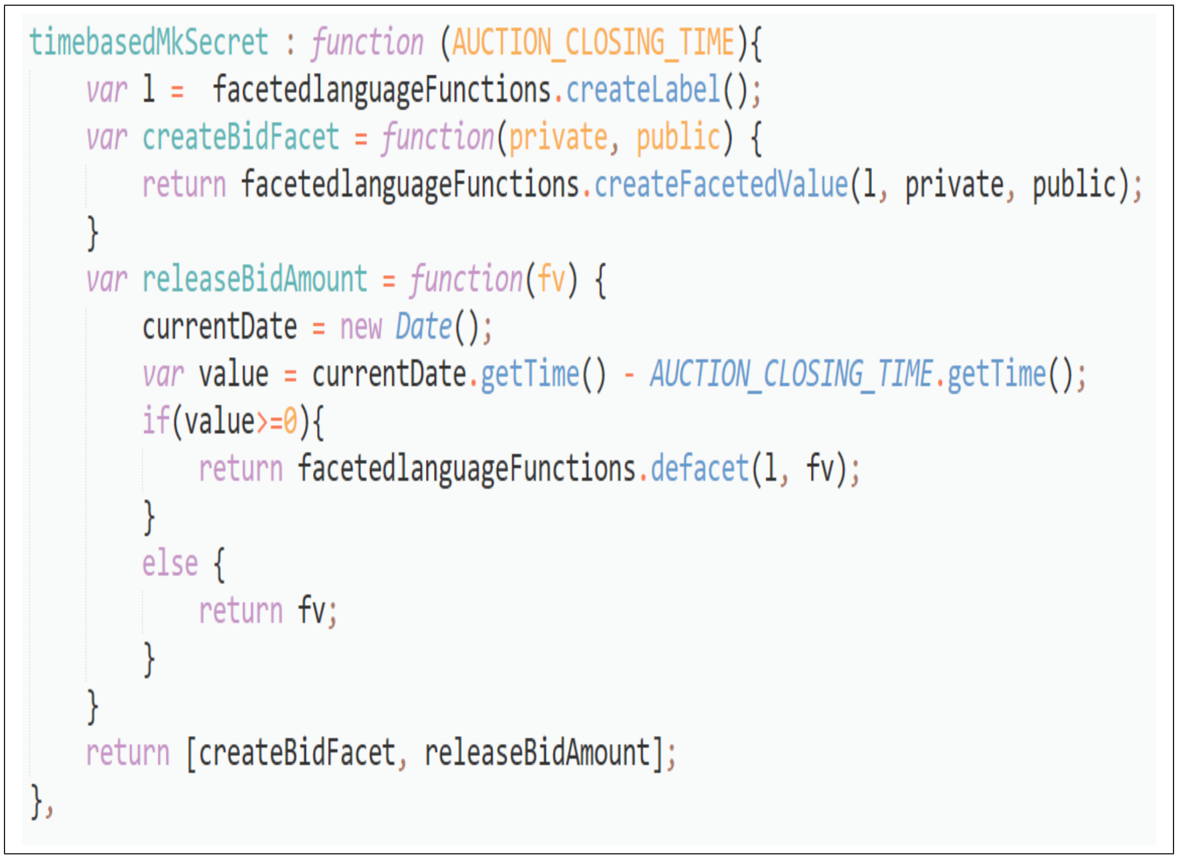
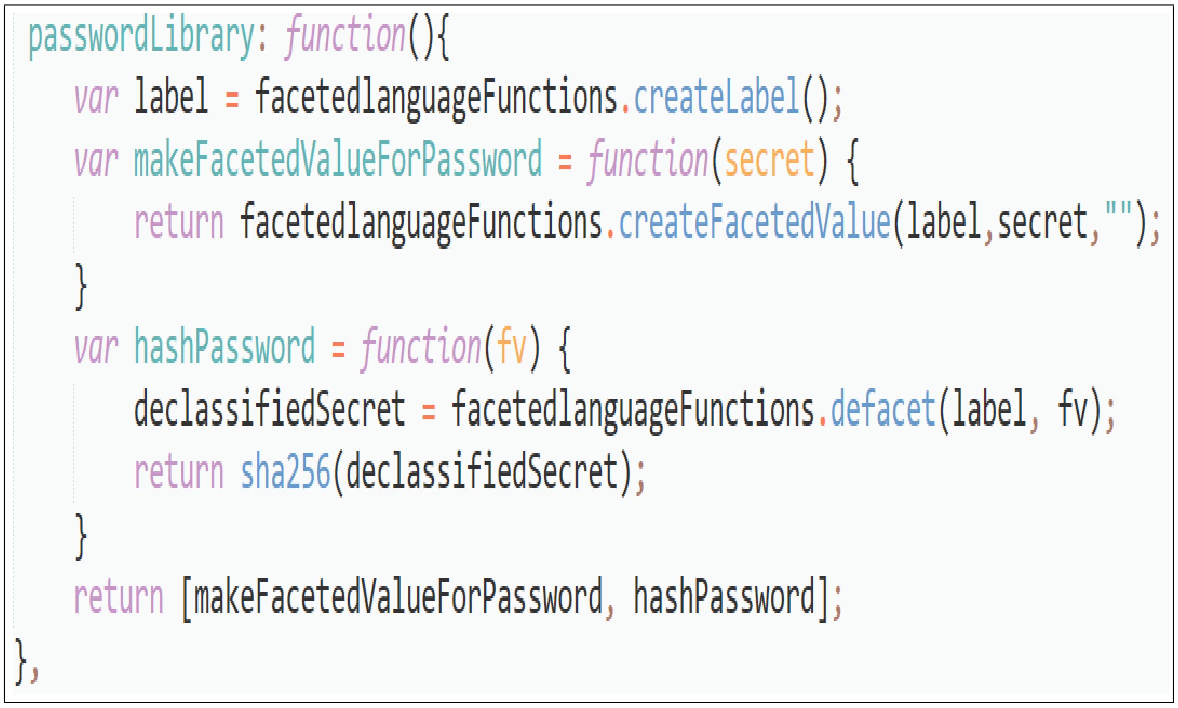

```
timebasedMkSecret : function (AUCTION_CLOSING_TIME){
    var l =  facetedlanguageFunctions.createLabel();
    var createBidFacet = function(private, public) {
        return facetedlanguageFunctions.createFacetedValue(l, private, public);
    }
    var releaseBidAmount = function(fv) {
        currentDate = new Date();
        var value = currentDate.getTime() - AUCTION_CLOSING_TIME.getTime();
        if(value>=0){
            return facetedlanguageFunctions.defacet(l, fv);
        }
        else {
            return fv;
        }
    }
    return [createBidFacet, releaseBidAmount];
},
```

Figure 14: Time based declassification

5.2.4 Declassification to release hash of the password

A canonical example of what based declassification is release of password. The password is still protected even if the hash is leaked. As hashing is a one-way encryption, there is no way to get the password using the hash. A policy that states "only hash of the password may be released when a user tries to login" can be implemented using faceted values. Developers can use this function to create faceted values for password with the help of `makeFacetedValueForPassword` and also release the hash of the password with the help of `hashPassword`. If a hacker/third-party system requests to view password or validate password, the hash of the password is returned. Here, the password itself is not leaked. Hence, the system is still secure.

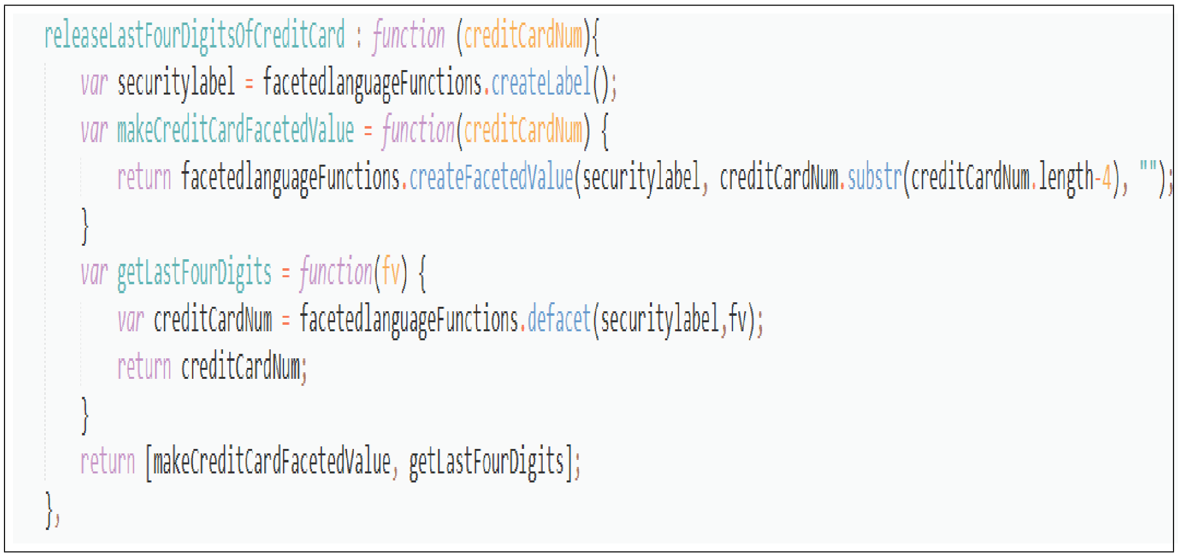


```
passwordLibrary: function(){
    var label = facetedlanguageFunctions.createLabel();
    var makeFacetedValueForPassword = function(secret) {
        return facetedlanguageFunctions.createFacetedValue(label,secret,"");
    }
    var hashPassword = function(fv) {
        declassifiedSecret = facetedlanguageFunctions.defacet(label, fv);
        return sha256(declassifiedSecret);
    }
    return [makeFacetedValueForPassword, hashPassword];
},
```

Figure 15: Declassification for releasing Hash of the password

5.2.5 Release last four digits for Credit Card

This scenario is another example of what based declassification. Here, we need to release the information about credit card but the policy is “release only the last four digits to the authorized users”. This can be easily implemented using FV. Figure 16 represents the code snippet for this policy. `makeCreditCardFacetedValue` gives the faceted value that represents the credit card number. The function `getLastFourDigits` gives the last four digits of the credit card number if the user has access to the security label, else an empty string is displayed.

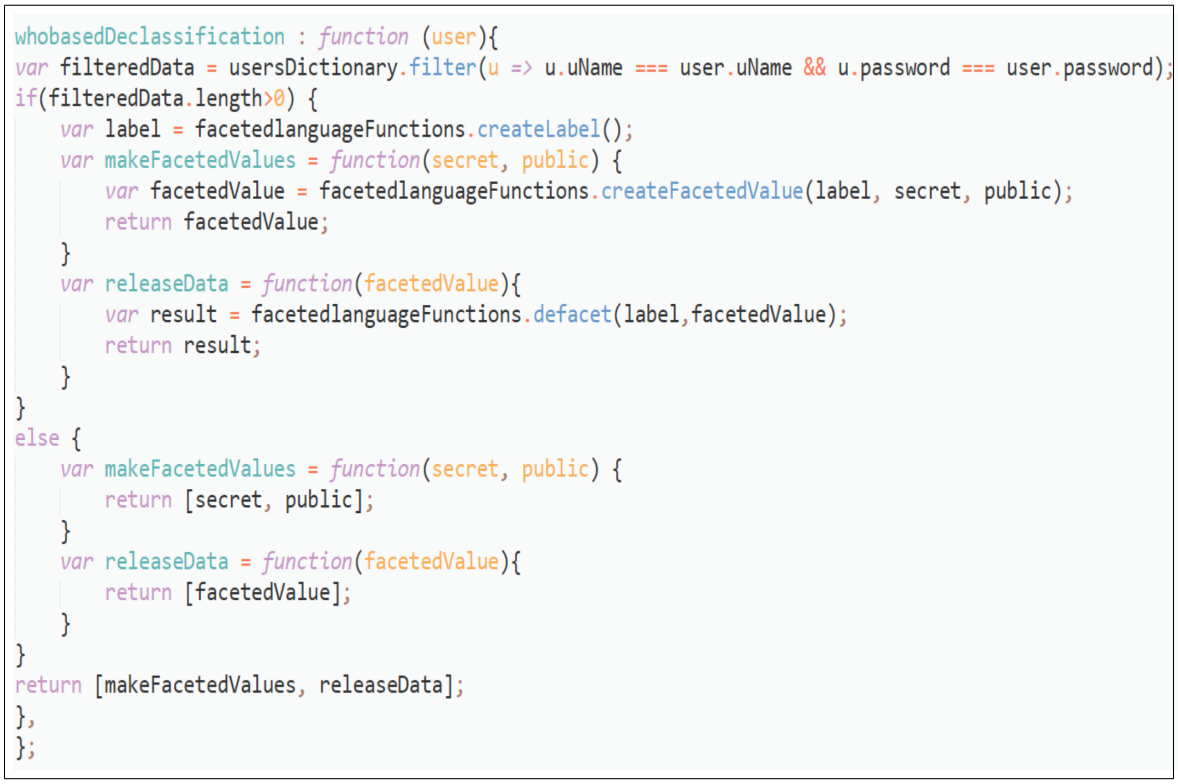


```
releaseLastFourDigitsOfCreditCard : function (creditCardNum){
    var securitylabel = facetedlanguageFunctions.createLabel();
    var makeCreditCardFacetedValue = function(creditCardNum) {
        return facetedlanguageFunctions.createFacetedValue(securitylabel, creditCardNum.substr(creditCardNum.length-4), "");
    }
    var getLastFourDigits = function(fv) {
        var creditCardNum = facetedlanguageFunctions.defacet(securitylabel,fv);
        return creditCardNum;
    }
    return [makeCreditCardFacetedValue, getLastFourDigits];
},
```

Figure 16: Declassification for releasing last four digits of the credit card

5.2.6 Who based Declassification example

Figure 17 represents an example scenario of `who` based declassification. The library writer explicitly specifies or stores the list of valid users who can classify and declassify sensitive information. The function `makeFacetedValues` can be used for creating faceted values and the function `releaseData` can be used for declassifying the sensitive information.



```
whobasedDeclassification : function (user){
var filteredData = usersDictionary.filter(u => u.uName === user.uName && u.password === user.password);
if(filteredData.length>0) {
    var label = facetedlanguageFunctions.createLabel();
    var makeFacetedValues = function(secret, public) {
        var facetedValue = facetedlanguageFunctions.createFacetedValue(label, secret, public);
        return facetedValue;
    }
    var releaseData = function(facetedValue){
        var result = facetedlanguageFunctions.defacet(label,facetedValue);
        return result;
    }
}
else {
    var makeFacetedValues = function(secret, public) {
        return [secret, public];
    }
    var releaseData = function(facetedValue){
        return [facetedValue];
    }
}
return [makeFacetedValues, releaseData];
},
};
```

Figure 17: Who based declassification

5.2.7 Limitations of Declassification with faceted values

Library writer bears an additional responsibility of carefully writing the rules of declassification. If the functions written by library writer are compromised then the system becomes vulnerable to attacks.

In the `who` based declassification mechanism implemented in this project, the library writer explicitly specifies which user can classify or declassify the data. But, ideally this is not a suitable approach and is prone to attacks. Zdancewic and Myers [23] implemented Robust declassification system. Our implementation doesn't support this system as it needs an overhead of adding a new data structure `pair` which has a pair of labels. Each pair has two labels. One label represents which user can modify the sensitive information (Integrity) and the other label represents the authorization of user to view the sensitive information (Confidentiality).

In this project, level locality of where based declassification is internally implemented by nested faceted values. This project has not explored the code locality where based declassification. This could be a future direction for this project.

JavaScript introspection exposes all the functions and properties associated with FV object. We need a mechanism that doesn't allow the introspection related function calls. Our approach assumes that all the JavaScript Reflection function calls are not allowed.

CHAPTER 6

Conclusion and Future Work

The necessity for an optimal data security mechanism is increasing with the amount of data and huge number of web applications. This project explored declassification for information flow analysis specifically focusing on faceted values.

The combination of first-class labels and faceted values have shown promising results for implementing different declassification scenarios. We validated our approach by implementing different scenarios that covers the 'what' [8], 'when' [8], 'who' [8] [23], non-interference, and unrestricted declassification for faceted values. With the help of the implemented examples, we can infer that faceted values and first-class labels are flexible for implementing most of the dimensions of declassification without changing the semantics of the language.

Future work for this project is to combine multiple dimensions of declassification to define a policy for practical scenarios. Another interesting direction could be to implement Robust declassification [23] to make the `who` based declassification secure. Another idea is to implement 'where' based declassification. Another idea is to incorporate all the policies and rules for an entire web application. This would give conclusive proof of the flexibility of faceted values for declassification as a whole. This research can also be extended by formulating concrete mathematical proofs to guarantee the security properties of the language design.

LIST OF REFERENCES

[1] “Top 10 biggest government data breaches of all time in the u.s.” 2015. [Online]. Available: https://digitalguardian.com/blog/top-10-biggest-us-government-data-breaches-all-time

[2] A. Birgisson, A. Russo, and A. Sabelfeld, “Capabilities for information flow,” in *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, ser. PLAS ’11. New York, NY, USA: ACM, 2011, pp. 5:1--5:15. [Online]. Available: http://doi.acm.org/10.1145/2166956.2166961

[3] J. A. Goguen and J. Meseguer, “Security policies and security models,” in *1982 IEEE Symposium on Security and Privacy*, April 1982, pp. 11--11.

[4] T. H. Austin and C. Flanagan, “Permissive dynamic information flow analysis,” in *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, ser. PLAS ’10. New York, NY, USA: ACM, 2010, pp. 3:1--3:12. [Online]. Available: http://doi.acm.org/10.1145/1814217.1814220

[5] T. H. Austin, T. Schmitz, and C. Flanagan, “Multiple facets for dynamic information flow with exceptions,” *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 3, pp. 10:1--10:56, May 2017. [Online]. Available: http://doi.acm.org/10.1145/3024086

[6] P. Li and S. Zdancewic, “Downgrading policies and relaxed noninterference,” in *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’05. New York, NY, USA: ACM, 2005, pp. 158--170. [Online]. Available: http://doi.acm.org/10.1145/1040305.1040319

[7] T. Wing, “Secure declassification in faceted javascript (2016). master’s projects. 472.” *M.S. Thesis. 472, San Jose State University, San Jose, CA, United States, 2016*. [Online]. Available: http://scholarworks.sjsu.edu/etd_projects/472

[8] A. Sabelfeld and D. Sands, “Dimensions and principles of declassification,” in *18th IEEE Computer Security Foundations Workshop (CSFW’05)*, June 2005, pp. 255--269.

[9] T. H. Austin and C. Flanagan, “Multiple facets for dynamic information flow,” in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’12. New York, NY, USA: ACM, 2012, pp. 165--178. [Online]. Available: http://doi.acm.org/10.1145/2103656.2103677

[10] A. Russo and A. Sabelfeld, ‘‘Dynamic vs. static flow-sensitive security analysis,’’ in *2010 23rd IEEE Computer Security Foundations Symposium*, July 2010, pp. 186--199.

[11] V. Rajani, I. Bastys, W. Rafnsson, and D. Garg, ‘‘Type systems for information flow control: The question of granularity,’’ *ACM SIGLOG News*, vol. 4, no. 1, pp. 6--21, Feb. 2017. [Online]. Available: http://doi.acm.org/10.1145/3051528.3051531

[12] S. A. Zdancewic, ‘‘Programming languages for information security,’’ Ph.D. dissertation, Ithaca, NY, USA, 2002, aAI3063751.

[13] T. H. Austin and C. Flanagan, ‘‘Efficient purely-dynamic information flow analysis,’’ *SIGPLAN Not.*, vol. 44, no. 8, pp. 20--31, Dec. 2009. [Online]. Available: http://doi.acm.org/10.1145/1667209.1667223

[14] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, ‘‘Generalizing permissive-upgrade in dynamic information flow analysis,’’ in *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security*, ser. PLAS’14. New York, NY, USA: ACM, 2014, pp. 15:15--15:24. [Online]. Available: http://doi.acm.org/10.1145/2637113.2637116

[15] D. Devriese and F. Piessens, ‘‘Noninterference through secure multi-execution,’’ in *2010 IEEE Symposium on Security and Privacy*, May 2010, pp. 109--124.

[16] A. Kalenda, ‘‘Implementation of faceted values in node.js, (2017). master’s projects. 564.’’ *M.S. Thesis. 564, San Jose State University, San Jose, CA, United States, 2017*. [Online]. Available: http://scholarworks.sjsu.edu/etd_projects/564

[17] M. S. Miller, K.-P. Yee, and J. Z. Shapiro, ‘‘Capability myths demolished,’’ 2003. [Online]. Available: http://www.erights.org/elib/capability/duals/capmyths.txt

[18] C. Bach Poulsen and P. D. Mosses, ‘‘Deriving pretty-big-step semantics from small-step semantics,’’ in *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 270--289. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54833-8_15

[19] J. A. Goguen and J. Meseguer, ‘‘Security policies and security models,’’ *1982 IEEE Symposium on Security and Privacy*, pp. 11--11, 1982.

[20] A. Askarov and A. Sabelfeld, ‘‘Localized delimited release: Combining the what and where dimensions of information release,’’ in *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, ser. PLAS ’07. New York, NY, USA: ACM, 2007, pp. 53--60. [Online]. Available: http://doi.acm.org/10.1145/1255329.1255339

[21] A. C. Myers and B. Liskov, ‘‘Protecting privacy using the decentralized label model,’’ *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 410--442, Oct. 2000. [Online]. Available: http://doi.acm.org/10.1145/363516.363526

[22] K. Pullicino, ‘‘Jif: Language-based information-flow security in java,’’ *CoRR*, vol. abs/1412.8639, 2014. [Online]. Available: http://arxiv.org/abs/1412.8639

[23] S. Zdancewic and A. C. Myers, ‘‘Robust declassification,’’ in *Proceedings of the 14th IEEE Workshop on Computer Security Foundations*, ser. CSFW ’01. Washington, DC, USA: IEEE Computer Society, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=872752.873524