Spring 5-22-2017

Policy-agnostic programming on the client-side

Kushal Palesha
San Jose State University

Policy-agnostic programming on the client-side

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

Kushal Palesha

May 2017

The Designated Project Committee Approves the Project Titled

Policy-agnostic programming on the client-side

by

Kushal Palesha

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2017

Dr. Thomas Austin	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. Jenny Lam	Department of Computer Science

ABSTRACT

Policy-agnostic programming on the client-side

by Kushal Palesha

Browser security has become a major concern especially due to web pages becoming more complex. These web applications handle a lot of information, including sensitive data that may be vulnerable to attacks like data exfiltration, cross-site scripting (XSS), etc. Most modern browsers have security mechanisms in place to prevent such attacks but they still fall short in preventing more advanced attacks like evolved variants of data exfiltration. Moreover, there is no standard that is followed to implement security into the browser.

A lot of research has been done in the field of information flow security that could prove to be helpful in solving the problem of securing the client-side. Policy-agnostic programming is a programming paradigm that aims to make implementation of information flow security in real world systems more flexible. In this paper, we explore the use of policy-agnostic programming on the client-side and how it will help prevent common client-side attacks. We verify our results through a client-side salary management application. We show a possible attack and how our solution would prevent such an attack.

Keywords Information flow security, policy-agnostic programming, faceted values

ACKNOWLEDGMENTS

I would like to thank Dr. Thomas Austin for introducing me to the field of information flow security and programming language theory. He has been a great guide throughout this journey.

I would also like to thank Dr. Chris Pollett and Dr. Jenny Lam for accepting to be part of my panel, taking interest in my project, and for being patient throughout this process.

Finally, I want to thank my family who have supported me in all my endeavors and also my friends for tolerating my blabbering whenever something went wrong with my project.

TABLE OF CONTENTS

CHAPTER

LIST OF FIGURES

CHAPTER 1

Introduction

The rapid increase in the number of applications that collect and process private data has made prevention of data leaks an involving task for security professionals. It is hard enough protecting against attacks on web servers, we now have sophisticated web applications that do more than just display static information received from the server. Javascript is the language of choice to develop these dynamic web pages, but there is lot of fragmentation in the Javascript engines used by different browsers, so if one browser may prevent some form of attack, we cannot assume that another browser may prevent the same form of attack.

In this paper, we propose the introduction of policy-agnostic programming (PAP) into Javascript to help protect sensitive data on the client-side. PAP is a programming paradigm introduced by Yang et al. [1] that builds on research efforts in language based information flow security. In their paper, Yang et al. introduced Jeeves, a language to write policy-agnostic programs. The PAP paradigm aims to make the implementation of information flow controls in complex real world systems flexible and intuitive. Yang et al. [2] presented how PAP can be used to protect data on a database backed server. They introduced an MVC framework called Jacqueline, which extends the Django framework [3] with Jeeves for policy-agnostic evaluation. Similarly, we will extend Javascript to support PAP and demonstrate how it can help prevent sensitive data leaks on the browser.

In the rest of this chapter we give a survey of the current state of browser security, proposals to protect against various client-side attacks and where our solution would fit into the client-side architecture. In Chapter 2 we give a brief background about information flow security, policy-agnostic programming, Jeeves, and related concepts. In Chapter 3 we review our solution, provide details of our implementation, and show

a sample application. In the final chapter we conclude.

1.1 Efforts to secure browser content

Ever since the first web browser was introduced [4] in 1990, the kind of content rendered has evolved dramatically. First it was only static html pages that were acquired from web servers. The introduction of the Common Gateway Interface (CGI) in 1993 [5], added the capability of generating dynamic web pages based on client requests made by the web browser. "Dynamic" here was pages that were created by web servers based on user requests and hence served personalized content that may include sensitive data. This meant there was content worth protecting, but even then most of the security measures were focused on the server-side (where all the data resided) since that was where attackers also focused their attention. With the introduction of Javascript in 1995 [6] web browsers became really powerful since it enabled web developers to create web pages with client-side interactivity without the need to make requests to a web server.

With further iterations of Javascript and the technologies around it, web pages have now evolved into web applications giving great control over sensitive user data to the client-side. The flexibility of Javascript that makes it possible to develop sophisticated web applications also makes content rendered by web browsers vulnerable to attacks. Over the years browser vendors have come up with various security measures to protect browser content.

1.1.1 Common security measures found in most modern browsers

Flanagan [7, Section 13.6] talks about browser security in brief and states two competing goals that browser vendors have tried to balance: ‘‘Defining powerful client-side APIs to enable useful web applications.’’ and ‘‘Preventing malicious code from reading or altering your data, compromising your privacy, scamming you, or

wasting your time.'' Flanagan [7, Section 13.6.1] lists some of the common security restrictions imposed on Javascript and notes that ''Different browsers have different security policies and may implement different API restrictions.''.

Same-origin policy

> ''The same-origin policy restricts how a document or script loaded from one origin can interact with a resource from another origin. It is a critical security mechanism for isolating potentially malicious documents.'' [8]

The article [8] gives details about how the same-origin policy controls Javascript behavior for different scenarios like cross-origin network access (control http requests or resource embedding tags), cross-origin script API access (limit access to Window and Location objects) and cross-origin data storage access. The same origin policy is very broad in terms of what it controls primarily because it needs to keep the flexibility that Javascript is known for. Flanagan [7, Section 13.6.2] has more details about the same origin policy and lists some techniques of how the same origin policy can be relaxed in some cases (read Cross-Origin Resource Sharing (CORS) [9]).

Content Security Policy

> ''Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware.'' [10]

In CSP, inline scripts are disabled by default, which would automatically prevent code injection based attacks. Additionally, CSP also allows a server administrator to restrict what sources a web page can import executable scripts from. On the face of it, CSP seems to provide a robust mechanism to prevent XSS and data exfiltration attacks.

1.1.2 Arguments against common browser security mechanisms

The security mechanisms mentioned above are implemented in all modern browsers to help prevent sensitive data leaks among other security breaches but

they are not foolproof solutions as presented by several researchers:

- Chen et al. [11] present cases where the same-origin policy falls short in protecting sensitive data from a particular form of the data-exfiltration attack. They define a data-exfiltration attack as:

  > ''an attack where the adversary exports user's private data to a server controlled by the attacker, possibly using a code injection vulnerability.''

  This form of attack would be prevented by the same origin policy. The authors present *self-exfiltration* as a new ''class'' of the data-exfiltration attack. In this form of attack, the injected script does not directly send the extracted data to an attacker-controlled server; instead it is posted to another location of the victim website itself or to whitelisted origins. The attacker can later log-on/access the victim website or whitelisted site respectively to retrieve the information.

- Acker et al. [12] present a strong case arguing the failure of CSP to prevent data exfiltration. They show that even the strongest CSP policies can be circumvented using DNS and resource prefetching as data exfiltration techniques.

The researchers above reveal flaws in existing browser security mechanisms and suggest possible improvements to fix them. As mentioned at the beginning of this chapter, all browsers are not created equal and so maybe a couple of browser vendors may implement one of the suggested solutions. But this would mean the vulnerability would still exist in the rest of the browsers until said solution becomes a standard and all the vendors adopt it. Moreover, these are just some examples of the many loopholes that researchers (and unfortunately attackers) have been finding in the current browser security architecture.

We propose adding information flow controls to client-side Javascript through PAP and use them to protect content displayed on a web page. We demonstrate

its usefulness in protecting content on the browser by incorporating PAP constructs into the document object model(DOM) [13], which is an application programming interface that defines how programs and scripts can access and update the content, structure, and style of documents.

CHAPTER 2

What is policy-agnostic programming?

Before we talk about policy-agnostic programming, we first need to give an introduction to the domain it belongs to, i.e., information flow security.

2.1 A brief overview of information flow security

Information flow security is a security mechanism that consists of information flow policies and information flow controls to detect and prevent leaking of sensitive data by an application. Information flow policies here are the policies that define where sensitive data can flow. Information flow controls are the mechanisms that enforce them.

2.1.1 Basic principles of information flow security

When designing a system with information flow security, sensitive data needs to be identified and corresponding information flow policies need to be defined. Smith [14] talks about basic principles of information flow security that we mention here in brief.

Security labels

We assign *security* labels to variables according to the level of security they are classified into. The most basic labels are L for low security or public information and H for high security or private information; the goal is to prevent improper leaks of information in H variables to L variables. The flow of data from an L variable into an H variable is legal.

Explicit and implicit flow

In information flow security, the leak can be in terms of an *explicit flow* or an *implicit flow*. The following is an example of an *explicit flow* where there is a direct flow of data from an H variable to an L variable:

$\texttt{public}_{\texttt{L}}$ = $\texttt{confidential}_{\texttt{H}}$

The code-snippet below is an example of an *implicit flow*:

```
if (confidential_H % 2) == 0
    public_L = 0
else
    public_L = 1
```

Although it may not seem obvious that there is a leak of sensitive data in this example, the last bit of the H variable ($\texttt{confidential}_\texttt{H}$) is being copied into the L variable ($\texttt{public}_\texttt{L}$). Any case where there is a branching statement that depends on an H variable has a potential data leak in terms of an *implicit flow*.

Noninterference

An important property in information flow security is noninterference. Smith [14] defines a program satisfying noninterference as:

> ''Program c satisfies noninterference if, for any memories μ and v that agree on L variables, the memories produced by running c on μ and on v also agree on L variables (provided that both runs terminate successfully).''

It is a formalization of the idea that a program should not leak information about H (private) variables through L (public) variables. All systems that provide information flow analysis need to prove noninterference. Although strict noninterference is not desired since a real world system may need to change labels or declassify (downgrade level of a variable) variables at runtime. One such relaxed form of noninterference is *termination-insensitive noninterference* which still maintains the property of private inputs not influencing public outputs on program execution although private information can influence the termination of the program. Although the termination of a program is a publicly observable fact, Askarov et al. [15] show that an attacker would be reduced to using a brute force approach which would take more than polynomial

time in the size of the sensitive data to exfiltrate.

Confidentiality and integrity

Information flow security is used to ensure *confidentiality* and *integrity* of data. Our solution will mostly focus on *confidentiality*, but here we would briefly like to define the two in the context of information flow security. *Confidentiality* as we have described above ensures that there is no unwanted flow of information from H variables into L variables. For example, you wouldn't want to allow the flow of data from a variable that contains credit card information into a variable that is meant to store payment amount. *Integrity* introduces the concepts of tainted and untainted variables. Variables that contain information received from an external source (network or user input) are marked as tainted. Here the aim is to not allow the flow of information from tainted variables into untainted variables. For example, data in a tainted variable needs to be sanitized before it can be used as a parameter in an SQL query or as part of a string that is used as input to the `eval` function. `eval` is a Javascript function that takes Javascript code (in the form of a string) as input and executes it.

2.1.2 Need for information flow security

Information flow analysis has seen very little adoption in commercial software systems. A majority of these systems rely on standard security mechanisms like access control, encryption, and firewalls. Sabelfeld and Myers [16] show how these mechanisms fall short in completely preventing sensitive data leaks. Access control is an important part of any security infrastructure that is used to control access of data to legitimate users, but once access is granted, there is no way to control how the data is used. Similarly, encryption will ensure that data will remain confidential while in transit between two end-points but once the data is decrypted at the receiving

end, there is no way to control the flow of data from that point onwards. A similar argument can be made for firewalls.

Yang [17] in her blog post talks about a very popular privacy leak in recent times that involved FBI director James Comey's ''secret'' Twitter account being discovered by Feinberg [18]. The leak was the result of an information flow vulnerability that exists in Instagram. Feinberg found the private account of James Comey's son Brian Comey. Ideally, a private account's ''following'' list should not be available to an external observer which is the case in a normal flow. But, when Feinberg requested to follow Brian's account, Instagram's ''helpful'' recommendation algorithm presented a list of suggested accounts to follow. All of them included the other Comey family members except for one account with an unusual name (''reinholdniebuhr''). With some online research, Feinberg was able to figure out that James Comey had written a thesis on ''Reinhold Niebuhr'' in law school. She found a Twitter account with the same name and verified that it was in fact his account due to a public statement he made specifying the number of people his ''secret'' Twitter account was following. While Instagram does protect the identity of users who wish to remain private, it does not prevent how their recommendation algorithms make use of this private data. The result of computations done by these algorithms is not protected which is what led to the leak in this particular case. Note: While Instagram was not directly responsible for this leak, Feinberg did exploit this undetected implicit flow that resulted in her findings.

2.1.3 Language based information flow security

There have been several approaches that researchers have tried to deal with the problem of information flow security, primarily categorized into static and dynamic analysis [19, 20]. In recent years, a lot of the research has been focussed on language-

based approaches with the objective of making information flow security a part of the programming language used for development. These are mostly extensions to existing programming languages that provide constructs to define labels, to specify policies, and to check those policies. As Sabelfeld and Myers [16] discuss, the use of type systems for information flow analysis presents a promising approach to get a practical implementation of information flow control. Here, every expression has a security type with two parts: an ordinary type and a label that describes how the value may be used. It is the job of the compiler to perform type checking; whenever a program containing labelled types is read, the compiler also makes sure that there will be no illegal flow of information at run-time. The authors call such a type system that enforces information flow policies a *security-type system*.

Jif (Java + information flow) is an example of a language with a *security-type system* [21], it extends Java with information flow controls and access controls that are enforced at compile-time and run-time. It is based on the JFlow language [22], which is the first usable programming model that provided static information flow analysis.

The programming model introduced with Jif provides a robust set of features along with the ability to specify information flow policies that are enforced by the Jif compiler, but it is still not adopted in many practical, real-world systems. This is largely due to the fact that a programmer must still have policy checking logic all over the program whenever a sensitive value is used. When access to a sensitive value is forbidden by a policy and the programmer has not handled such cases, the program is likely to behave in an unexpected manner or get stuck. As Yang puts it in her thesis [23], handling all cases where sensitive variables are used leads to ‘‘programmer burden from policy spaghetti’’. That is, using a system like Jif results in policy checking logic scattered throughout the code. This leads to code that is

difficult to maintain and prone to human error. Another problem is realized when these policies need to be changed, which would mean changes everywhere the sensitive variable is used.

2.2 The policy-agnostic programming model

In an attempt to make the implementation of information flow security more flexible, Yang et al [1] introduced the policy-agnostic programming (PAP) model. PAP is an approach where the developer only needs to focus on writing core functionality without the additional burden of thinking about data privacy constraints on sensitive values.

PAP is introduced using Jeeves, a domain specific language that provides constructs to define sensitive labels, policies, and to mark variables as sensitive. In Jeeves, sensitive data has two views associated with it, a high confidentiality view and a low confidentiality view. What view of the sensitive data is revealed to a particular output channel depends on the `context` of the channel and the policy associated with the sensitive data. `Context` here is an object that contains relevant information that a policy may refer to. This can vary depending on the application. Austin et al [24] provide an example of a `context` object for a health database application:

```
HealthContext {viewer: User, time: Date}
```

Here, the policies attached to sensitive variables will be resolved based on the user who is trying to access a health record while time allows certain policies to define expiration and activation times for visibility. The goal of PAP is that you define all policies on sensitive values when they are defined. After this, everywhere the sensitive value needs to be sent to an output channel, the language runtime is responsible for enforcing and checking the policies associated with them.

The initial implementation of Jeeves involved symbolic execution and constraint

solving to produce outputs adhering to the policies associated with sensitive data. This approach had limitations in terms of implementation feasibility and expressiveness, which were later addressed by extending it with faceted values [24]. The faceted execution of policy-agnostic programs is based on work by Austin and Flanagan [25].

2.2.1 What is a faceted value?

A faceted value as defined in [25] is ‘‘a triple consisting of a principal k and two values V_H and V_L.’’ It is represented as:

$$\langle k \ ? \ V_H \ : \ V_L \rangle$$

Here you can imagine the principal k to be the owner/guard of the sensitive value. The faceted value appears as V_H (private facet) to private observers that are allowed to view k's private data, and as V_L (public facet) to other public observers. In Jeeves, principals of faceted values have policies associated with them which are rules that the ‘‘guard’’ will check before providing access to the private facet. These policies need not be checked till the value needs to be revealed to an output channel. So, while faceted values are flowing through a program, there are special semantics that define how they should be evaluated. The following is an example of a faceted value that specifies a sensitive email address:

$$\langle k \ ? \ \text{‘jon@sjsu.edu’} \ : \ \text{‘[redacted]’} \rangle$$

When the policy associated with k can be resolved to true, the value ‘jon@sjsu.edu’ is visible and if the policy is resolved to false, then the value ‘[redacted]’ is visible.

2.2.2 Evaluation semantics of faceted values

The semantics of Jeeves are modeled using λ^{jeeves} which is the core language that extends the faceted execution semantics of Austin and Flanagan [25]. Figure 1 shows the λ^{jeeves} language. Notable here are the faceted expressions, label declarations

and policy specification expressions which help build policy agnostic programs. We briefly discuss some of the faceted evaluation semantics to give a gist of a how a policy agnostic program might behave in certain scenarios. In Chapter 3, we will talk about the Jeeves constructs that enable PAP.

A *program counter (pc)* is used to track when program execution is being influenced by a public or private facet. Any expression involving a faceted value becomes a faceted expression. So if we have something like:

$$\langle k \ ? \ V_H \ : \ V_L \rangle + number$$

This can be though of as the faceted expression:

$$\langle k \ ? \ V_H + number \ : \ V_L + number \rangle$$

which is of the form:

$$\langle k \ ? \ e_1 \ : \ e_2 \rangle$$

The faceted evaluation semantics shown in Figure 2 defines [F-SPLIT] as the rule to evaluate faceted expressions. Here, assuming neither k nor $\bar{k}$ are in the current pc, we evaluate both expressions e_1 and e_2 one after the other. First, k is added to pc to obtain V_1 from e_1 and then $\bar{k}$ is added to pc to obtain V_2 from e_2. Finally, we create a new faceted value:

$$\langle k \ ? \ V_1 \ : \ V_2 \rangle$$

On the other hand, if the pc already contains either k or $\bar{k}$, then only one expression is evaluated as defined in the rules [F-LEFT] and [F-RIGHT].

In Figure 3, the [F-IF-SPLIT] rule defines how a conditional statement is handled for faceted values. Figure 4 shows a code snippet which has a potential to leak data through an implicit flow. Line 8 defines a faceted value which is sent as input to the function `f`. Line 3 has a conditional statement which is dependent on this faceted

Syntax:

$e ::=$		*Term*
	x	variable
	c	constant
	$\lambda x.e$	abstraction
	$e_1\ e_2$	application
	ref e	reference allocation
	$!e$	dereference
	$e := e$	assignment
	$\langle k\ ?\ e_1 : e_2 \rangle$	faceted expression
	label k in e	label declaration
	restrict(k, e)	policy specification
$S ::=$		*Statement*
	let $x = e$ in S	let statement
	print $\{e\}\ e$	print statement
$c ::=$		*Constant*
	f	file handle
	b	boolean
	i	integer
x, y, z		*Variable*
k, l		*Label*

Figure 1: The λ^{jeeves} source language [24]

value. Here, the conditional statement would become a faceted expression of the form `<h? 25 > 0 : 0 > 0>`. The [F-SPLIT] rule would be used to evaluate this expression to `<h?true:false>`. Now, the [F-IF-SPLIT] rule would be used to evaluate the `if` block from line 3 to line 5. First, h is added to the `pc` to evaluate the `if` block under the influence of the private facet which would be the assignment expression: `E1 = v1 := false`. Next, the `if` block is evaluated under the influence of the public facet by adding $\bar{h}$ to the `pc`. This would be a no-op since there is no else block. Note, under the influence of the private facet, we assigned `false` to the variable `v1` which already had `true` assigned to it. At the end of evaluation of the `if` block, the final value of `v1` is `<h?false:true>`.

The faceted evaluation semantics of Jeeves gives it a few desirable properties;

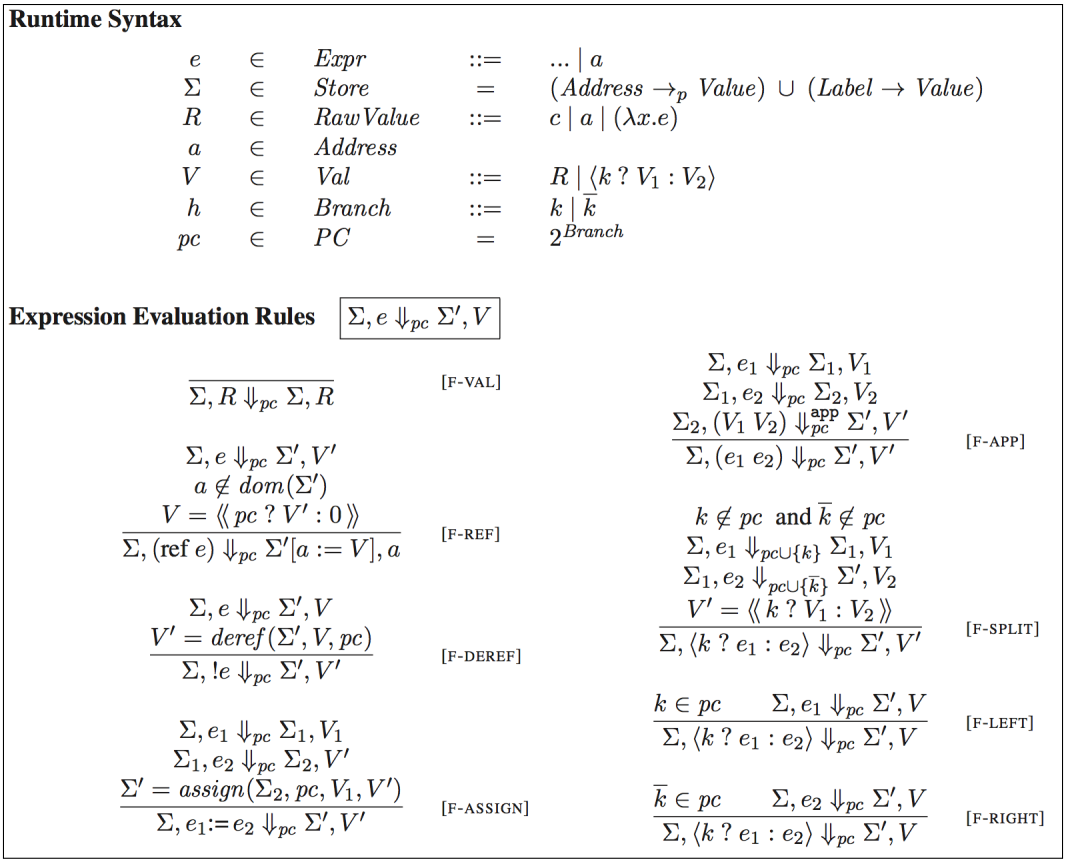


Figure 2: Faceted evaluation semantics [24]

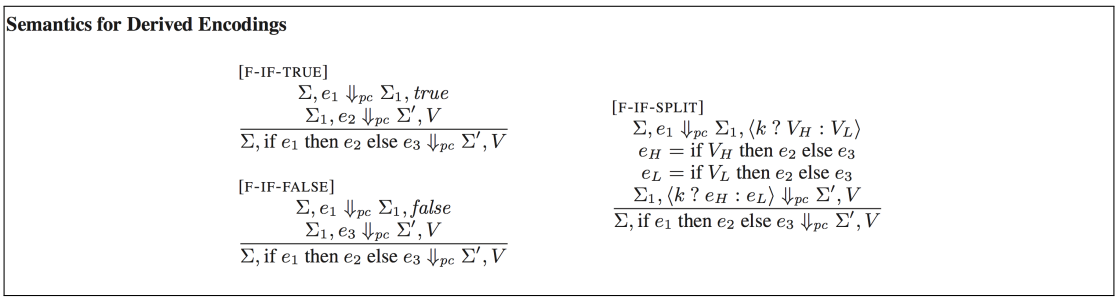


Figure 3: Semantics of Derived Encodings [24]

the **projection property** which states that a single execution with faceted values can be projected onto multiple different executions without faceted values, the termination-insensitive noninterference (the projection property helps to prove this), and termination-insensitive policy compliance which states that data is revealed to an

```
1 function f(val) {
2   var v1 = true;
3   if (val > 0) {
4     v1 = false;
5   }
6   return v1;
7 }
8 var f1 = <h?25:0>;
9 f(f1);
```

Figure 4: Faceted Evaluation of a potential implicit flow

external observer only if it is allowed by the policy specified in the program.

2.2.3 The Jacqueline Framework

Yang et al. [2] demonstrate the practical feasibility of the policy-agnostic programming paradigm for database-backed server-side applications. They introduce Jacqueline, an MVC framework with an aim to provide a platform to easily implement information flow security in server side applications. Figure 5 shows a snippet of a ''Model'' definition in Jacqueline, how a sensitive value is defined, and what a policy looks like. Here, `project_name` is the field we are marking as sensitive by using the annotation `@label_for`. The method `jeeves_restrict_projectlabel` defines the policy which is a boolean function while the method `jeeves_get_private_project_name` defines the public facet of the sensitive value. In `jeeves_restrict_projectlabel`, `ctxt` is the context object representing the output channel which in the case of this application is the currently logged-in user. Once the policy for a sensitive variable is defined, the programmer need not worry about where or how they are using this value; the Jeeves runtime will take care of any potential leaks and the policy is only resolved when the value is going out to an output channel like the client-side.

As seen in the example above, the Jeeves runtime in the Jacqueline framework took care of information flow control and resolved the policy for sensitive data only when going out to the browser. But, once data is on the browser, the concept of

```
class Project(JacquelineModel):
    project_name = models.CharField(max_length = 128)
    code_name = models.CharField(max_length = 128)
    start_date = models.DateTimeField('date started')
    end_date = models.DateTimeField('date ended')
    department = ForeignKey(Department, on_delete=models.CASCADE)

    @staticmethod
    def jeeves_get_private_project_name(project):
        return project.code_name

    @staticmethod
    @label_for('project_name')
    @jeeves
    def jeeves_restrict_projectlabel(project, ctxt):
        return project.department == ctxt.department
```

Figure 5: Model definition in the Jacqueline framework

facets is lost. However, we may want to further protect our data against exfiltration attacks. This is where our solution would help by persisting faceted values and policies associated with them on the client-side.

2.3 Related Work

Austin and Flanagan's [25] original work showed the benefits of faceted values for dynamic information flow control by giving an example of how it could help reduce the power of an XSS attack. Rajani et al. [26] implement information flow controls for event handling and the DOM API which is based on work done by Bichhawat et al. [27] in which they build information flow controls into the WebKit Javascript engine. The drawback of both these approaches is that when an information flow control is violated, the execution is halted which may not be desirable for dynamic web pages.

Koskela et al. [28] present an interesting approach to browser security by presenting an actor based approach where the various content providers (actors) that make up a web page are accountable for the content they send. They confine each actor

within a `<div>` tag and based on their track record, the user/browser can decide how much restriction to enforce on a particular `<div>` node.

Policy agnostic programming along with faceted values provides a very flexible approach (which is desirable on the client-side) to information flow control while still providing strong guarantees. We discuss more about our solution in Chapter 3 along with a demonstration of how it would help protect browser content.

CHAPTER 3

Implementing policy-agnostic programming on the client side

We incorporated policy-agnostic programming into Javascript by adding constructs defined for the Jeeves language as a subsystem[1]. We chose Narcissus [29], a Javascript interpreter written in Javascript, to create the proof of concept. Narcissus was built by its developers to be able to prototype new language features for Javascript.

3.1 Implementing faceted values in Narcissus

The implementation of faceted values is integral to implementing Jeeves. Our implementation of faceted values derives heavily from the work done by Austin and Flanagan [25] and is inspired from the concept presented by Kerchove et al. [30] for ‘‘modular instrumentation’’ of interpreters. They talk about how many dynamic analysis approaches for information flow security have prototypes that are implemented in very specific ways making it difficult to compare and reuse. They derive some specific criteria to follow in order to achieve ‘‘modular instrumentation’’ using the implementation [31] of faceted values [25] as a case study. While we borrow a few ideas from this, our implementation does not follow the criteria specified because achieving ‘‘modular instrumentation’’ would involve non-trivial changes to the Narcissus interpreter making it out of scope for our problem. What we did achieve is the untangling of the concerns of the core interpreter for non-faceted evaluation from the concerns of faceted evaluation. This makes it easier to relate principles of faceted evaluation with the implemented prototype and also to extend or reuse it.

Our implementation of faceted values is independent from the core Narcissus interpreter. This involved making the core interpreter modular to be able to modify existing evaluation mechanisms to behave differently for faceted values. The Narcissus interpreter has an execute function that consists of a switch case control flow that is

[1]The code is available at: `https://github.com/kushalpalesha/narcissus`

set to perform the appropriate set of operations based on the type of node identified by the parser. Wherever there is need for faceted behavior, instead of making the core interpreter code handle faceted behavior, we moved the part we would need to change into a function and later override it to handle the faceted behavior. Figure 6 shows how this overriding is implemented for the [F-IF-SPLIT] evaluation rule. Here, `BaseExecContext` is an object that stores a copy of all the fucntions from the core interpreter which we need to override. `ExecutionContext` is an object from the core interpreter that keeps track of the current flow of execution. `FacetExecContext` is the `ExecutionContext` object extended with the *pc* to help keep track of the influence of public or private facets on the current flow of execution. In the `evalIfBlock` function, notice we call the base `evalIfBlock` function if `cond` is not a faceted value. Otherwise, we call the `evaluateEach` function which implements the [FA-SPLIT] rules from Figure 7. We override behavior of the rest of the functions in `BaseExecContext` in a similar manner.

3.2 Implementing Jeeves

Once we had faceted values working, adding support for Jeeves constructs was pretty straightforward. All of the Jeeves constructs are encapsulated in the prototype of the `PolicyEnvironment` object which includes the [F-LABEL] and [F-RESTRICT] evaluation rules as shown in Figure 8. Every instance of the `PolicyEnvironment` object has a `policyMap` that is used to store the 'label':'policy' mapping.

The Jeeves constructs available in the `PolicyEnvironment` prototype are:

1. `mkLabel:` This function is roughly based on the [F-LABEL] rule. It creates a label and associates a default true policy to it.

2. `restrict:` This function associates the given policy function to the given label in the `policyMap` of the current `PolicyEnvironment`.

```
var BaseExecContext = {
  getValue : interpreter.ExecutionContext.prototype.getValue,
  putValue : interpreter.ExecutionContext.prototype.putValue,
  evalBinOp : interpreter.ExecutionContext.prototype.evalBinOp,
  evalUnaryOp : interpreter.ExecutionContext.prototype.evalUnaryOp,
  evalIfBlock : interpreter.ExecutionContext.prototype.evalIfBlock,
  evalDot : interpreter.ExecutionContext.prototype.evalDot,
  evalFunctionCall : interpreter.ExecutionContext.prototype.
     evalFunctionCall,
  runWhileLoop : interpreter.ExecutionContext.prototype.runWhileLoop
};

FacetExecContext.prototype.evalIfBlock = function(cond,thenPart,
    elsePart) {
  var execContext = FacetExecContext.current;
  if (cond instanceof FacetedValue) {
    evaluateEach(cond, function(v, x) {
      if (v) {
        interpreter.execute(thenPart, x);
      } else if (elsePart) {
        interpreter.execute(elsePart, x);
      }
    }, execContext);
  } else {
    BaseExecContext.evalIfBlock.call(this,cond,thenPart,elsePart);
  }
};
```

Figure 6: Independent implementation of faceted behavior for the "if" control flow

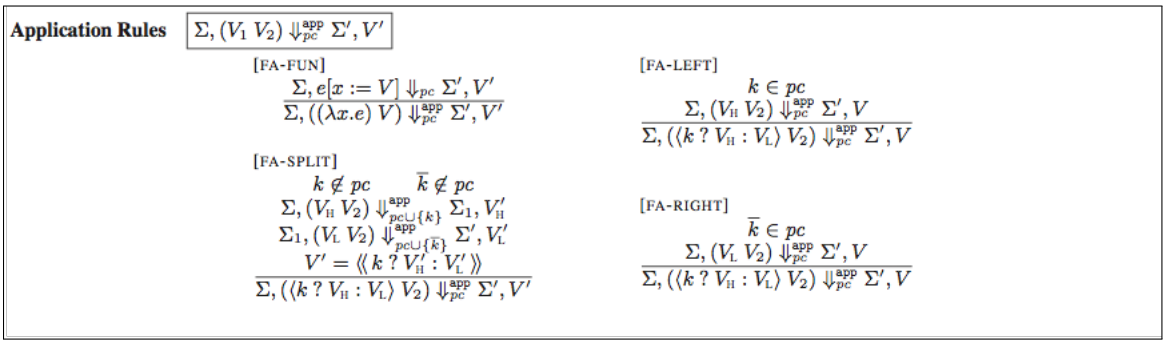
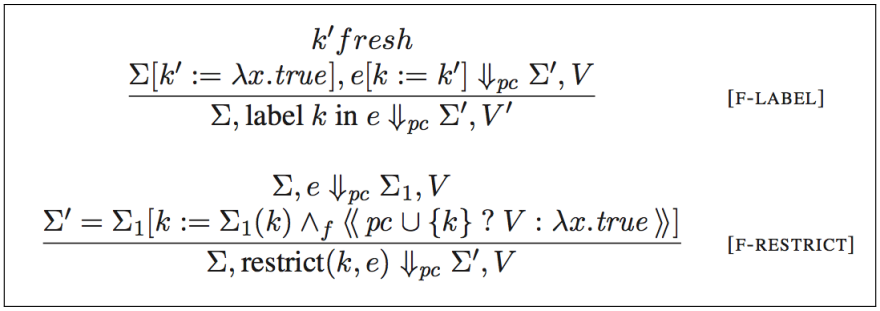


Figure 7: Function application rules [24]

3. `mkSensitive`: This function creates a faceted value. It takes the label, private value, and public value as input and returns a faceted value. This function would only return the private value or the public value in cases where the current



$$\frac{k' fresh \quad \Sigma[k' := \lambda x.true], e[k := k'] \Downarrow_{pc} \Sigma', V}{\Sigma, \text{label } k \text{ in } e \Downarrow_{pc} \Sigma', V'} \quad \text{[F-LABEL]}$$

$$\frac{\Sigma, e \Downarrow_{pc} \Sigma_1, V \quad \Sigma' = \Sigma_1[k := \Sigma_1(k) \wedge_f \langle\!\langle\, pc \cup \{k\} \;?\; V : \lambda x.true \,\rangle\!\rangle]}{\Sigma, \text{restrict}(k, e) \Downarrow_{pc} \Sigma', V} \quad \text{[F-RESTRICT]}$$

Figure 8: Evaluation semantics for Jeeves labels and policies [24]

program counter contains the label or reverse of the label respectively.

4. `concretize:` This function is used when the faceted value needs to be viewed in an output context. It takes the context object and faceted value as input and resolves the policies for all labels in the program counter of the faceted value recursively till it reaches a raw value with no facets.

5. `partialConcretize:` Partial concretize is similar to the concretize function except it only resolves the policy associated with the first label of a possibly complex faceted value based on the given context object. Figure 9 shows what the concretize and partialConcretize functions look like.

3.2.1 Using Jeeves constructs

Figure 10 shows two test cases of how the Jeeves constructs listed above would be used. Note, `policyEnv` is an instance of the `PolicyEnvironment` prototype.

In `testPolicyComplexFacets`, we are constructing complex faceted values with two principals/labels and have two different policies for each respectively. The call to `concretize` at the end shows what a context object would look like in this case. The faceted value stored in `a` in notation looks like this: $\langle x \;?\; \langle y \;?10 \;:\; 15\rangle \;:\; 0\rangle$.

In `testPartialConcretize`, we are using `partialConcretize` with two different context objects. This type of usage would be ideal for a client-server interaction where

```
function concretize(context, val) {
  if (val instanceof FacetedValue) {
    var label = head(val);
    var policy = this.policyMap[label];
    if (policy(context)) {
      return this.concretize(context, val.high);
    } else {
      return this.concretize(context, val.low);
    }
  } else {
    return val;
  }
}
function partialConcretize(context, val) {
  if (val instanceof FacetedValue) {
    var label = head(val);
    var policy = this.policyMap[label];
    if (policy(context)) {
      val = val.high;
    } else {
      val = val.low;
    }
  }
  return val;
}
```

Figure 9: Concretize and partialConcretize function definitions

you can have different context objects for the server and client-side respectively. Note: in the example, the policies associated with the two labels are expecting different properties in the context object passed to them.

3.3 Policy agnostic programming in dom.js

Web browsers have an implementation of the DOM to allow scripts to access and manipulate content. We add faceted values and policy-agnostic programming constructs to the DOM implementation to show how it can be used to prevent sensitive data from leaking.

We use dom.js [32], which is a DOM implementation written in Javascript. This makes it possible for us to parse it using Narcissus and make DOM components available to scripts just like a web browser would. The advantages of using dom.js

```
 function() testPolicyComplexFacets{
  var x = policyEnv.mkLabel("x");
  policyEnv.restrict(x, function (context) {
    return context.val1 === 22 && context.val2 === 21;
  });

  var y = policyEnv.mkLabel("y");
  policyEnv.restrict(y, function (context) {
    return context.val2 === 22;
  });
  var a = policyEnv.mkSensitive(x, policyEnv.mkSensitive(y, 10, 15),
      0);
  return assertEquals(policyEnv.concretize({val1: 22, val2: 21}, a),
      15);
};

function testPartialConcretize() {
  var x = policyEnv.mkLabel("x");
  policyEnv.restrict(x, function (context) {
    return context.val1 === 22 && context.val2 === 21;
  });

  var y = policyEnv.mkLabel("y");
  policyEnv.restrict(y, function (context) {
    return context.otherVal = 44;
  });
  var a = policyEnv.mkSensitive(x, policyEnv.mkSensitive(y, 10, 15),
      0);

  var result1 = assertEquals(policyEnv.partialConcretize({val1: 22,
     val2: 21}, b).toString(), "{y?10:15}");
  var result2 = assertEquals(policyEnv.partialConcretize({val:22}, b
     ), 0);
  return result2 && result1;
};
```

Figure 10: Example usage of Jeeves constructs

is highlighted by Austin et al. [33, Section 9.3] since it makes it possible to include faceted values in the DOM and track flow of private information on the web browser.

We first identify the entry and exit points in the DOM that have the potential to leak sensitive data such as when the `setAttribute` and `getAttribute` functions of an element are called; when a `Text` node is created and appended to a DOM and when the `innerHTML` property of an element is used to access the text within an element;

and finally when an `XMLHttpRequest` is made to load an external script, image, or other media.

Note here, the `setAttribute` function and creation of the `textNode` are entry points into the DOM. Here, we have to be careful not to render sensitive information onto unwanted components like the *src* attribute of an image or script tag. We discuss such a scenario in Section 3.4. On the other hand, once a value is rendered onto the DOM, a script may try to access rendered values using the `getAttribute` function and the `innerHTML` property.

We introduce an instance of the `PolicyEnvironment` prototype to the window object. This gives access to Jeeves constructs within the DOM along with a `policyMap` for each web page. We also introduce a `facetedValueMap` available as a global store that associates textNodes or attributes of elements with corresponding faceted values. The `facetedValueMap` is of type WeakMap that provides a loose mapping from objects to values [34].

Figure 12 shows how creation of a text node for faceted values is handled within the DOM. The function `createTextNode` creates a node that would eventually be rendered onto a web page. At this point, we need to decide which facet of a faceted value should be rendered. If the input to `createTextNode()` is faceted, then we concretize that value to get a raw value to be rendered. Note, in the concretize function we do not specify what the `context` object looks like. We talk about this in detail in Section 3.5. Additionally, we add the faceted value itself to the `facetedValueMap` with the textNode as key. We have added similar code in the `setAttribute` function of an element with one distinction to the object that is used as key for the `facetedValueMap`. Here we cannot use the node as the key since we need to have different keys for different attributes of an element. So, we created an object using the id of the element and the attribute name as follows:

```
1  // Convert the children of a node to an HTML string.
2  // This is used by the innerHTML getter
3  serialize: constant(function() {
4    var s = "";
5    for(var i = 0, n = this.childNodes.length; i < n; i++) {
6      var kid = this.childNodes[i];
7      if (kid in facetedValueMap) {
8        return facetedValueMap[kid];
9      }
10     .
11     .
12     .
13   }
14   .
15   .
16   .
17 }
```

Figure 11: Return faceted value if exists when the innerHTML property is accessed

```
1  createTextNode: function createTextNode(data) {
2    var dataString = data;
3    var dataIsFaceted = isFaceted(data);
4    if (dataIsFaceted) {
5      dataString = window.policyEnv.concretize({...}, data);
6    }
7    var textNode = unwrap(this).createTextNode(String(dataString));
8    if (dataIsFaceted) facetedValueMap[textNode] = data;
9    return wrap(textNode);
10 },
```

Figure 12: Persisting faceted values for createTextNode

`facetedValueMap[{id:this.id, attr:attributeName}] = value;`

Figure 11 shows how access to the `innerHTML` property of an element would return a faceted value instead of the actual content that was rendered on the web page (lines 7-9). Note that `serialize` is a function called by the `innerHTML` getter. We have similar code to return a faceted value in the `getAttribute` function of an element with the key as shown above for the `setAttribute` function.

3.4 A data exfiltration case study

We present a simple data exfiltration attack that succeeds in exfiltrating sensitive data from the a web page to a server that the attacker owns. A direct XMLHttpRequest to do this would be prevented by the same-origin policy of the web browser, but there is a simple workaround. The same-origin policy does not restrict the source of a script or image tag. Although you may use CSP (see Section 1.1.1) to restrict sources, it becomes difficult to track what image sources to allow and so web developers tend to keep the CSP of *img-src* as a wildcard (*), allowing all urls for images.

Figure 13 shows a screenshot of a simple web page we created that shows the salary of the user currently logged-in and salaries of his subordinates. The helpful greeting at the top right corner along with the nice background color is a due to a third-party library that Trudy suggested would be a nice addition to make the otherwise mundane user interface better. It turns out the third-party library also does some malicious activity along with these ‘‘colorful’’ additions. Figure 14 shows the code of the third party library. Here, lines 14-15 would get the message that displays Manny's salary. Lines 17-20 extract Manny's name and salary and line 21 would result in an attempt to asynchronously load an image with the name ‘‘Manny_10000.jpg’’ from ‘‘localhost:8081’’[2] which is not the same as the origin of the web page as shown in the address bar in Figure 13. This would happen anytime Manny clicks anywhere on the web page. Now, although there is no image with the name ‘‘Manny_10000.jpg’’ at ‘‘localhost:8081’’, the attacker can access Manny's salary by checking their http access logs, as shown in Figure 15.

Now let us look at how policy-agnostic programming controls in the DOM would prevent such an attack. Figure 16 shows a code snippet of the code that would display a message similar to the one shown in Figure 13. Note, the string concatenation on

[2]Here, we are using a different port number to stand in for an alternate url such as ‘‘evil.com’’

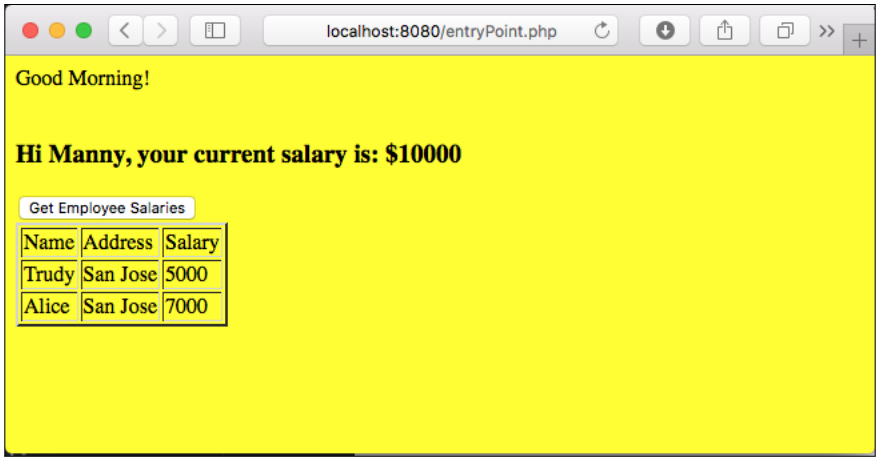


Figure 13: Web page that displays employee salaries

```
1  var d = new Date();
2  var time = d.getHours();
3  var greetingNode = document.getElementById("greeting");
4  var message = "Good day!";
5  document.body.style.fontStyle.color = "black";
6  //This section sets background color and greeting based on the time
7  .
8  .
9  var welcomeText = document.createTextNode(message);
10 greetingNode.appendChild(welcomeText);
11 //Malicious code:
12 document.addEventListener("click", function () {
13   var salaryField = document.getElementById("OwnSalary");
14   var text = salaryField.innerHTML;
15   var malImg = document.createElement("img");
16   var commaPos = text.indexOf(",");
17   var dollarPos = text.indexOf("$");
18   var name = text.slice(3,commaPos);
19   var salary = text.slice(dollarPos+1);
20   var imgName = name + "_" + salary + ".jpg";
21   malImg.setAttribute("src","http://localhost:8081/" + imgName);
22   //The following would violate the same origin policy:
23   //$.post('http://localhost:8081/exfil.php',{message:text});
24 });
```

Figure 14: Third Party library with exfiltration code

line 4 would produce a faceted value of the form:

```
<"n"? <"s"?"Manny's Salary is:10000":"Manny's Salary is:0">:
    <"s"?"JonDoe's Salary is:10000":"JonDoe's Salary is:0"> >
```

```
10.0.2.2 - - [03/May/2017:18:00:50 +0000]
"GET /Manny_10000.jpg HTTP/1.1" 404 213
"http://localhost:8080/entryPoint.php"
"Mozilla/5.0 (Macintosh; Intel Mac OS X
10_12_4) AppleWebKit/603.1.30 (KHTML, like
Gecko) Version/10.1 Safari/603.1.30"
```

Figure 15: Access log entry giving Manny's salary information to the attacker

```
1 var domPolicyEnv = window.policyEnv;
2 var fName = domPolicyEnv.mkSensitive("n", "Manny", "JonDoe");
3 var fSalary = domPolicyEnv.mkSensitive("s",10000, 0);
4 document.body.appendChild(document.createTextNode(fName + "'s Salary
      is:" + fSalary));
```

Figure 16: Code that would set the display message on the web page

Now, when the code from Figure 14 is run, the concatenation on line 21 would produce a faceted value of the form:

```
<"n"? <"s"?"Manny_10000.jpg":"Manny_0.jpg">:
    <"s"?"JonDoe_10000.jpg":"JonDoe_0.jpg"> >
```

So, with the correct policies in place (see Section3.5), the message displayed on the web page would be ‘‘Manny's Salary is:10000’’ and the image request would be for ‘‘JonDoe_0.jpg’’.

3.5 The context object and defining policies

In Section 2.2 we briefly touched upon the definition of a context object and what it might look like in a health database application. Here, we define what a context object would look like for our case study above and how it would be used by policy functions. A context object contains all information that is relevant to define the ‘‘context’’ of the output channel and varies based on the output channel. For instance, the context object when an image load request is made by the browser could be defined as:

```
{time:  new Date(), elementType:  "img"}
```

When defining a policy for a sensitive value, the designer/developer needs to be aware of where it is expected to flow to. A good approach to creating a policy is identifying the two kinds of output channels: one where we expect our data to flow to, and the other where we definitely do not want the data to leak to. For the channels where we expect the data to flow to, we identify conditions under which we would allow the private facet to be sent out. For all other cases, we allow only the public facet to be seen. How you define a policy completely depends on your application and the data you are trying to protect. In our example, we do not expect salaries to be used in the image tag among other conditions so we need to define the policy accordingly. Figure 17 shows what that policy might look like. Here, we specify three conditions: one which defines we do not want salary data to flow to the image or script tag; the second one specifies that any attempt to render or use salary data past 6:00 pm would not be allowed; the third specifies the only condition in which salary data is allowed to be rendered. Notice, in the third condition we are looking for an attribute that is not part of the context object we have specified above. When `createTextNode` is called the context object would look like the following:

```
{time:  new Date(), URL:mycorp.org/salaryManager.php}
```

With this context, assuming time is less than 6:00 pm, and the policy in Figure 17, the private facet will be rendered to the Text node.

3.6 Client-server interaction with policy-agnostic programming

Since we have seen the uses of policy-agnostic programming on both the server (see Section 2.2.3) and client-side (see Section 3.3) we should talk about how we imagine they would interact with each other. First thing to note is that policies on the server-side would not be relevant to the client-side and vice-versa. This is mostly

```
function (ctxt) {
  if (ctxt.elementType && ctxt.elementType == "img" || ctxt.
      elementType == "script") {
    return false;
  } else if (ctxt.time && ctxt.time.getHours() > 18) {
    return false;
  } else if (ctxt.URL && ctxt.URL == "mycorp.org/salaryManager.php")
      {
    return true;
  }
  return false;
}
```

Figure 17: Example of a policy function for the salary faceted value

due to the fact the output context in both cases will be different and the kind of information leaks they are trying to prevent will also be different. To visualize this notion, we present an example that demonstrates the interaction of faceted values between the server and client-side. Suppose, the server of an application stores location information of a user as a faceted value of the form:

```
<"serverlabel"?
<"clientlabel"?"Psychiatric center, 4th St.":"Bermuda triangle">:
<"clientlabel"?"Doctor's office":"Bermuda triangle">
```

Note, the policy function for "serverlabel" would be in the `policyEnvironment` of the server, while the policy function for the "clientlabel" would be in the `policyEnvironment` of the client. When the location data is to be sent to a client, the context object here would be the currently logged in user and the policy function could be defined such that only the Doctor of the user is able to access the private facet while other users would get access to the public facet. Here, the `partialConcretize` function from Section 3.2 would be useful since we do not want to concretize to a raw value. We will have different policies on the client side that define what facet is rendered on the browser.

CHAPTER 4

Future Work and Conclusion

We explored the use of policy-agnostic programming in the DOM. Another interesting area to explore on the client-side would be frameworks like Angular.js (angular). Since, angular is an MVC framework, the policy-agnostic programming solution for it should be similar to what Yang et al. [2] have demonstrated with Jacqueline.

We saw how robust information flow controls can help prevent leaking of sensitive data and how it has seen very slow adoption because of the programmer burden to write and maintain policy code. Policy-agnostic programming is a promising approach to implement information flow controls in your system with very limited programmer burden.

We explored how policy agnostic programming would look like on the client-side. Through our Javascript implementation in Narcissus and dom.js we were successful in demonstrating how it helped prevent a known exfiltration attack to which most modern browsers are vulnerable.

LIST OF REFERENCES

[1] J. Yang, K. Yessenov, and A. Solar-Lezama, ''A language for automatically enforcing privacy policies,'' 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. 85--96. [Online]. Available: http://projects.csail.mit.edu/jeeves/papers/popl088-yang.pdf

[2] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, ''Precise, dynamic information flow for database-backed applications,'' in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16. New York, NY, USA: ACM, 2016, pp. 631--647. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908098

[3] T. Christie, ''Django rest framework,'' Framework Documentation, 2017, [Online]; accessed April, 2017. [Online]. Available: http://www.django-rest-framework.org

[4] T. B. Lee, ''Worldwideweb,'' W3C, 1990, [Online]; accessed April, 2017. [Online]. Available: https://www.w3.org/People/Berners-Lee/WorldWideWeb.html

[5] D. Robinson and K. Coar, ''The common gateway interface (cgi) version 1.1,'' Internet Request for Comments, The Internet Engineering Task Force, Tech. Rep., 2004 (accessed April, 2017). [Online]. Available: https://tools.ietf.org/html/rfc3875

[6] C. Severance, ''Javascript: Designing a language in 10 days,'' *Computer*, vol. 45, no. 2, pp. 7--8, Feb 2012.

[7] D. Flanagan, *JavaScript: The Definitive Guide: Activate Your Web Pages*. O'Reilly Media, 2011.

[8] J. Ruderman, ''Same-origin policy,'' MDN Docs, 2016, accessed April, 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy

[9] A. van Kesteren, ''Cross-origin resource sharing,'' W3C Recommendation, W3C, Tech. Rep., 2014, [Online]; accessed April, 2017. [Online]. Available: https://www.w3.org/TR/cors/

[10] ''Content security policy (csp),'' MDN Docs, 2017, [Online]; accessed April, 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

[11] E. Y. Chen, S. Gorbaty, A. Singhal, and C. Jackson, ‘‘Self-exfiltration: The dangers of browser-enforced information flow control,’’ in *Proceedings of the Workshop of Web 2.0 Security and Privacy*, vol. 2. Citeseer, 2012. [Online]. Available: http://www.w2spconf.com/2012/papers/w2sp12-final11.pdf

[12] S. Van Acker, D. Hausknecht, and A. Sabelfeld, ‘‘Data exfiltration in the face of csp,’’ in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS ’16. New York, NY, USA: ACM, 2016, pp. 853--864. [Online]. Available: http://doi.acm.org/10.1145/2897845.2897899

[13] P. L. H. Arnaud Le Hors, ‘‘Document object model core,’’ W3C Recommendation, W3C, Tech. Rep., 2004, [Online]; accessed April, 2017. [Online]. Available: https://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/core.html

[14] G. Smith, *Malware Detection*. Boston, MA: Springer US, 2007, ch. Principles of Secure Information Flow Analysis, pp. 291--307. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-44599-1_13

[15] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands, ‘‘Termination-insensitive noninterference leaks more than just a bit,’’ in *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ser. ESORICS ’08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 333--348. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88313-5_22

[16] A. Sabelfeld and A. C. Myers, ‘‘Language-based information-flow security,’’ *IEEE J.Sel.A.Commun.*, vol. 21, no. 1, pp. 5--19, sep 2006. [Online]. Available: http://dx.doi.org/10.1109/JSAC.2002.806121

[17] J. Yang, ‘‘Five research ideas instagram could have used to protect comey’s secret twitter,’’ Blog Post, 2017, [Online]; accessed April, 2017. [Online]. Available: http://jxyzabc.blogspot.com/2017/03/five-research-ideas-instagram-could.html

[18] A. Feinberg, ‘‘This is almost certainly james comey’s twitter account,’’ Gizmodo article, 2017, accessed April, 2017. [Online]. Available: https://gizmodo.com/this-is-almost-certainly-james-comey-s-twitter-account-1793843641

[19] A. Russo and A. Sabelfeld, ‘‘Dynamic vs. static flow-sensitive security analysis,’’ in *2010 23rd IEEE Computer Security Foundations Symposium*, 2010, pp. 186--199. [Online]. Available: http://www.cse.chalmers.se/~andrei/csf10.pdf

[20] A. Chlipala, ‘‘Static checking of dynamically-varying security policies in database-backed applications,’’ in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 105--118. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924951

[21] L. Z. A. C. Myers, N. Nystrom and S. Zdancewic, ‘‘Jif: Java information flow,’’ http://www.cs.cornell.edu/jif, 2015, [Online]; accessed April, 2017.

[22] A. C. Myers, ‘‘Jflow: Practical mostly-static information flow control,’’ in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’99. New York, NY, USA: ACM, 1999, pp. 228--241. [Online]. Available: http://doi.acm.org/10.1145/292540.292561

[23] J. Yang, ‘‘Preventing information leaks with policy-agnostic programming,’’ dissertation, MIT, 2015. [Online]. Available: http://www.cs.cmu.edu/~jyang2/papers/jeanyang_phd_thesis.pdf

[24] T. H. Austin, J. Yang, C. Flanagan, and A. Solar-Lezama, ‘‘Faceted execution of policy-agnostic programs,’’ in *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, ser. PLAS ’13. New York, NY, USA: ACM, 2013, pp. 15--26. [Online]. Available: http://projects.csail.mit.edu/jeeves/papers/plas07-austin.pdf

[25] 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

[26] V. Rajani, A. Bichhawat, D. Garg, and C. Hammer, ‘‘Information flow control for event handling and the dom in web browsers,’’ in *2015 IEEE 28th Computer Security Foundations Symposium*, July 2015, pp. 366--379.

[27] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, *Information Flow Control in WebKit’s JavaScript Bytecode*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 159--178. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-54792-8_9

[28] J. Koskela, N. Weaver, A. Gurtov, and M. Allman, ‘‘Securing web content,’’ in *Proceedings of the 2009 Workshop on Re-architecting the Internet*, ser. ReArch ’09. New York, NY, USA: ACM, 2009, pp. 7--12. [Online]. Available: http://doi.acm.org/10.1145/1658978.1658981

[29] ‘‘Narcissus,’’ GitHub, 2012, accessed April, 2017. [Online]. Available: https://github.com/mozilla/narcissus

[30] F. Marchand de Kerchove, J. Noyé, and M. Südholt, ‘‘Towards modular instrumentation of interpreters in javascript,’’ in *Companion Proceedings of the 14th International Conference on Modularity*, ser. MODULARITY Companion 2015. New York, NY, USA: ACM, 2015, pp. 64--69. [Online]. Available: http://doi.acm.org/10.1145/2735386.2736753

[31] T. Austin, ‘‘Zaphodfacets,’’ GitHub, 2011, accessed April, 2017. [Online]. Available: https://github.com/taustin/ZaphodFacets

[32] A. Gal, ‘‘dom.js,’’ GitHub, 2012, accessed April, 2017. [Online]. Available: https://github.com/andreasgal/dom.js

[33] 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, April 2017, forthcoming.

[34] ‘‘Weakmap,’’ MDN Docs, 2017, [Online]; accessed April, 2017. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/WeakMap