Spring 5-22-2017

Implementing Dynamic Coarse & Fine Grained Taint Analysis for Rhino JavaScript

Tejas Saoji
San Jose State University

Implementing Dynamic Coarse & Fine Grained Taint Analysis for Rhino JavaScript

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Tejas Saoji

May 2017

The Designated Project Committee Approves the Project Titled

Implementing Dynamic Coarse & Fine Grained Taint Analysis for Rhino JavaScript

by

Tejas Saoji

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2017

Thomas Austin	Department of Computer Science
Robert Chun	Department of Computer Science
James Casaletto	Department of Computer Science

ABSTRACT

Implementing Dynamic Coarse & Fine Grained Taint Analysis for Rhino JavaScript

by Tejas Saoji

Web application systems today are at great risk from attackers. They use methods like cross-site scripting, SQL injection, and format string attacks to exploit vulnerabilities in an application. Standard techniques like static analysis, code audits seem to be inadequate in successfully combating attacks like these. Both the techniques point out the vulnerabilities before an application is run. However, static analysis may result in a higher rate of false positives, and code audits are time-consuming and costly. Hence, there is a need for reliable detection mechanisms.

Dynamic taint analysis offers an alternate solution — it marks the incoming data from the untrusted source as 'tainted.'' The flow of tainted data is tracked during the program execution. Whenever tainted data is used in a security-sensitive context, a proper action is taken. The execution may also be suspended depending upon the severity of the operation.

This project implements dynamic taint analysis in Rhino JavaScript. The focus is on adding support for coarse-grained and fine-grained string tainting. Coarse-grained tainting works at the granularity level of a string while fine-grained tainting works at the granularity level of a character in a string. Both approaches are discussed in further detail in the paper. I have also written a SQL library to leverage my implementation of taint analysis in Rhino and conducted performance tests to contrast the overhead of coarse & fine grained taint analysis. My test results show that fine-grained taint analysis in general incurs more overhead than coarse-grained taint analysis.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor Prof. Thomas Austin for his continuos engagement and guidance throughout this project, offering valuable feedback, and motivating me to do better. I would also like to thank Prof. Robert Chun and Prof. James Casaletto for agreeing to be on my project committee, and reviewing the report. Finally, I would like to thank my parents and friends for providing me with constant support and encouragement throughout my years of study, and through this project.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

The Basics

Web application systems today are at great risk from attackers. They use techniques like SQL injection, format string attacks, and cross-site scripting to exploit vulnerabilities in web applications [1]. Attacks like these are caused due to ''bad'' user input. Attacks on web applications caused due to unchecked user input are recognized as being the most common. The above mentioned attacks are a form of code injection. SQL injection is where a user inputs bad SQL statements into the data sent to an application that results in unintended actions being performed on the database. Cross-site scripting occurs when a dynamically generated web page shows user input that has not been properly validated. Format string attacks occur when user input is treated as a command by an application. When unchecked user input is provided as a format string to the `printf` function in C, the attacker can control the behavior of `printf` by inserting format directives [2]. The `eval` is another such example. The eval is a dynamic feature provided by JavaScript that contain a code string that is executed dynamically. When user entered data is passed to eval, it enables the attacker to have his desired code executed when the program is executed.

It is very crucial to prevent the above mentioned attacks. However, it is difficult to do so by imposing standard access control policies [2]. For example, in the case of an SQL injection, a policy is enforced that does not allow special characters (for e.g. semicolon, single-quote) and/or SQL keywords/commands (for e.g. UPDATE, DELETE, DROP, WHERE) in the user input. Though enforcing such a policy may prevent a possible SQL injection attack, it may sometimes clash with valid user inputs. It does in this specific case when the user input is ''O'Reilly.'' An entry like this

would be considered as invalid, and hence a perfectly valid query would be rejected. Such a policy makes the use of database very difficult, and may result in failure of the web application [2].

Other methods like static analysis and code audits also turn out to be not very fruitful in successfully combating the attacks mentioned above. One advantage that static analysis provides is that it finds all the vulnerabilities before the application is run. It avoids runtime taint tracking and hence is more efficient that dynamic analysis. However, it lacks precision. Static analysis may produce a warning whenever untrusted data is used in a security-critical operation, even if there is no attack. This problem can be addressed by the use of the concept of *endorsement*. The developer needs to put validation checks on the untrusted data to ensure that it is safe, and then ''endorse'' it to indicate that that data can be safely used in a security-sensitive operation. The difficult task is to identify all the places in the code that require validation checks [2]. This requires detailed code audits. Code audits are very time consuming and costly affairs. As most developers do not possess expertise in performing security audits, they are often outsourced to security consultants, which adds to the cost [3]. The notion of validity is determined by the source of the input and the manner the input is used in the program. Hence, one needs to go back and forth to determine appropriate validation checks to be put at appropriate places in the code. Moreover, it is hard for the programmers to code validation checks correctly, and the checks themselves frequently have been the source of vulnerabilities [2]. Hence, this altogether is a very error prone process.

Hence, to address such vulnerabilities, the paper suggests implementing taint analysis. Taint analysis is a form of information flow analysis [1]. The incoming data from outside the code is not trustworthy. Hence, taint analysis marks the data coming

from untrusted sources as tainted. It also tracks the flow of taint in the program. Taint information flows from one variable to another. When tainted data affects untainted data, the previously untainted data is also marked as tainted. This is how taint propagates in a program. When tainted data is used in security-critical functions, an error is thrown. As an example, a tainted string used in a print statement. Taint analysis helps in detection and prevention of the aforementioned vulnerabilities in web application systems.

Two ways to implement taint analysis are:

1. Static analysis
2. Dynamic monitors

Static analysis provides one advantage over dynamic analysis; it finds all the potential vulnerabilities before program is executed. However, in comparison to dynamic analysis, static analysis lacks accuracy. The incoming data is tainted while the program is in execution in dynamic taint analysis. The tainted data is tracked throughout the program execution. Whenever tainted data is used in a security-sensitive context, a proper action is taken. The execution may also be suspended depending upon the severity of the operation.

This project implements dynamic taint analysis in JavaScript. JavaScript is one of the most popular languages for client-side as well as server-side web applications. It was recently reported that 98 websites out of the 100 most favored websites use JavaScript [4]. JavaScript is also used by many tablets and smartphones to provide platform independent features. The flexibility and dynamism that the language provides makes it vulnerable to attacks like the ones mentioned above. Given the popularity of the language, it is critical to address these vulnerabilities [4].

Dynamic taint analysis tracks taint propagation when the program is in execution. Two main approaches to dynamic taint analysis are [5] :

1. Coarse-grained taint analysis
2. Fine-grained/Precise-grained taint analysis

Coarse-grained taint analysis tracks taint information at the granularity level of a string. It marks an untrusted string coming in from an outside source as tainted. The taint propagates through the program execution at the string level. Hence, in Figure 1 concatenation of an untainted string in variable `x` and a tainted string in variable `y` results in the resultant string ''Hello O'Reilly'' in variable `z` to be marked as tainted.

```
var x = "Hello ";              //untainted string
var y = taint("O'Reilly");     //tainted string
var z = x + y;                 //tainted string
```

Figure 1: String tainting

The limitation with coarse-grained taint analysis is that, since it tracks taint at the string level, it marks an entire string tainted even if it derives only a part from an untrusted string. Hence, it is hard to sanitize a tainted string, untaint it, and use it in a security-sensitive operation. This results in frequent false positives [5].

Languages like Perl and Ruby also provide taint support to prevent the systems from attacks that are caused due to invalid input. Ruby has functions like `taint` to explicitly mark the object as tainted, `tainted?` to check whether an object is tainted, and `untaint` to remove the tainted mark from the object [6]. In Perl, one can use the `-T` command line flag to explicitly enable the taint mode. In taint mode, Perl keeps a check on every variable to see if it is tainted. Perl taints every data that comes from

outside the code (for e.g., environment variables, command line arguments, all file input, etc.), and does not allow such data to be used in an eval, passed through a shell, or used in any of the Perl commands that modify files, directories, or processes [7]. One can use the `tainted` or `istainted` functions to determine whether a variable contains tainted data. By using the tainted values as keys in a hash the values may be untainted. The other way to do it is by using regular expressions. One can refer to the subpatterns by matching tainted data to the regular expressions. The regular expression match checks that the tainted data has only ''good'' characters, and no ''bad'' characters. Its the developers job to know what is ''safe,'' and design the regular expressions accordingly.

Both Ruby & Perl implement tainting at coarser granularity. Perl is conservative while determining if the data is tainted or not. It follows the principle of ''one tainted value taints the whole expression'' [7]. Consider a subexpression of an expression that contains tainted data. The subexpression is also marked as tainted, even if its value itself does not contain any part of the tainted data.

Fine-grained taint analysis helps overcome the limitations posed by coarse-grained taint analysis. Fine-grained taint analysis tracks taint information at the granularity level of a character in a string. In Figure 2, concatenation of an untainted string in variable `x` and a tainted string in variable `y` results in the resultant string ''Hello O'Reilly'' with tainted characters indexed from 6 to 13 in variable `z` to be marked as tainted.

```
var x = "Hello ";           //untainted string
var y = taint("O'Reilly"); //tainted string
var z = x + y;              //string has tainted characters
```

Figure 2: String tainting

Performance tests were conducted against an unmodified version of Rhino, and my own two implementations of Rhino — Rhino coarse-grained and Rhino fine-grained. SunSpider benchmark test suite was considered for this experiment. The performance tests were conducted to contrast the overhead of coarse-grained & fine-grained taint analysis. My test results show that fine-grained taint analysis in general incurs more overhead than coarse-grained taint analysis.

Organization of the paper is as follows — The next chapter focuses on common vulnerabilities in web application systems, traditional techniques to prevent these attacks, and their limitations, and taint analysis. Chapters 3 talks about the implementation in detail, and provides an insight into how dynamic taint analysis helps us prevent these attacks by giving some nice examples. Chapter 4 presents the performance test results. Chapter 5 concludes the paper and also discusses the scope of future work.

CHAPTER 2

Literature Review

In this chapter, we discuss two common vulnerabilities in web application systems — SQL injection and cross-site scripting. We also discuss a few conventional techniques to counter these attacks and some of their limitations. Finally we explore how taint analysis can be deployed to overcome those limitations to help successfully combat these attacks. Through this survey, we will try to find answers to the questions: *What is taint analysis?* and *Why taint analysis?* The articles that are chosen in this survey focus on both fundamental research and new developments in the application of taint analysis in order to prevent these attacks on web applications.

2.1 Web Application Vulnerabilities

In this section, we talk about two attacks that are frequent in web application systems in detail.

2.1.1 SQL Injection

SQL injection is one of the most common vulnerabilities in web application systems. It is a form of code injection technique that is used to attack applications driven by user data. A user enters ‘‘Bad’’ SQL statements into a textfield on a web form for execution (e.g. to drop a table from the database, to update an entry in the table etc.) [2] [3] [8]. Figure 3 shows a SQL query. The query in the variable `queryString` derives a part of its value from the variable `name`, which contains untrusted or a ‘‘tainted’’ string accepted from a user as input.

```
queryString="select * from employee where fname ='"+name+"'";
```

Figure 3: Query string

An authorized user will enter a valid string. An attacker might enter something like:

''Jane' ; DROP TABLE employee; -- '';

With this value in the variable *name*, the query string in `queryString` looks like:

*select * from employee where fname = 'Jane'; DROP TABLE employee; --'*

The semicolon is used to end the first query. The query string constructed in the variable `queryString` consists of two queries. The first query retrieves all the details of a person named Jane from the `employee` table, and the second query drops the `employee` table from the database. Even though, the attacker does not have permissions to drop the table from the database, he is able to do so by inserting bad SQL statement in an entry field.

Imposing a policy that forbids the query string or a part of it to have tainted special characters (for e.g. semicolon and single-quote) and/or SQL keywords/commands (for e.g. UPDATE, DROP, WHERE, DELETE) will help prevent the attack [2].

2.1.2 Cross-Site Scripting (XSS)

Cross-site scripting attack is caused when dynamically generated web pages display user input on the browser before it is rightly checked and validated [3]. An attacker may insert JavaScript code designed to so something bad into dynamically generated content on a web page . When a user views such a page, the malicious code gets executed on his/her machine. This script may steal the user's account credentials

or cookie information stored on his machine, and dump it to the attacker. Xu et al. [2] gives a good example of a cross-site scripting attack. In this example, the web page contains a form that submits a query to the web site. The query looks like this:

http://www.xyzbank.com/findATM?zip=95126

If a wrong zip code is entered, the website displays an error message like:

<HTML> ZIP code not found: 95126 </HTML>

In the above response, the user entered zip code is displayed back on the browser. A sophisticated cross-site scripting attack may be constructed by an attacker who misuses this. The attacker may send an email that contains a link to an unaware user. The link could be like — click ''here'' to claim a reward. He implants a nasty JavaScript code in the link. When the user clicks on the link ''here,'' a request is sent to the bank with the nasty script as the zip code. The bank returns the same response as above, this time embedding the nasty script in its response. This nasty code is then downloaded and executed on the user's machine. Since the response was sent from a trusted website (i.e. the bank's website), the code has access to all the private information on the user machine.

2.2 Conventional techniques & their limitations

Conventional techniques like static analysis, code reviews, and access control policies are popular to detect vulnerabilities caused because of invalid user input, like the ones discussed above.

The main advantage of static analysis is that it finds the vulnerabilities statically, and hence is more efficient than dynamic analysis. However, it lacks precision. Static

analysis may produce a warning whenever untrusted data is used in a security-sensitive operation. This problem can be addressed by the use of the concept of endorsement. Endorsement is indicating that a particular dependency is ‘‘safe’’ to use. The developer needs to put validation checks on the untrusted data to ensure that it is ‘‘safe, ’’ and then endorse it to indicate that it can be used safely. The difficult task is to identify all the places in the code that require validation checks [2]. This requires detailed code reviews. Code reviews are very time consuming and costly affairs. They are often outsourced to security consultants as most developers do not possess the expertise in performing security audits, which adds to the cost. Also, new security errors are often introduced as the old ones are corrected. Hence, this may require a second code review [3]. The source of the input and the manner the input is used in determine the notion of validity. Hence, one needs to go back and forth to determine appropriate validation checks to be put at appropriate places in the code. Moreover, it's hard for programmers to code validation checks appropriately, and they themselves frequently have been identified as the source of vulnerabilities [2]. Hence, this makes the entire process prone to a lot of errors.

Enforcing traditional access control policies also does not provide an effective solution to prevent these attacks [2]. For example, in the case of an SQL injection, a policy is enforced that does not allow special characters (for e.g. semicolon, single-quote) and/or SQL keywords/commands (for e.g. UPDATE, DELETE, DROP, WHERE) in the user input. Though enforcing such a policy may prevent a possible SQL injection attack, it may sometimes clash with valid user inputs. It does in this specific case when the user input is ‘‘O'Reilly.’’ An entry like this would be considered as invalid, and hence a perfectly valid query would be rejected. Such a policy makes the use of database very difficult, and may result in failure of the web application [2].

2.3 Taint Analysis Approaches

Taint analysis is a form of information flow analysis. Information flow analysis ensures integrity of sensitive data by ensuring that information from tainted variables do not flow into untainted variables [9]. Taint analysis tracks the flow of taint information from one variable to another.

Two main approaches to taint analysis are — *static* approach and *dynamic* approach, both of which have their own advantages and disadvantages. Figure 4 shows the different approaches to taint analysis. This section discusses each approach in detail.

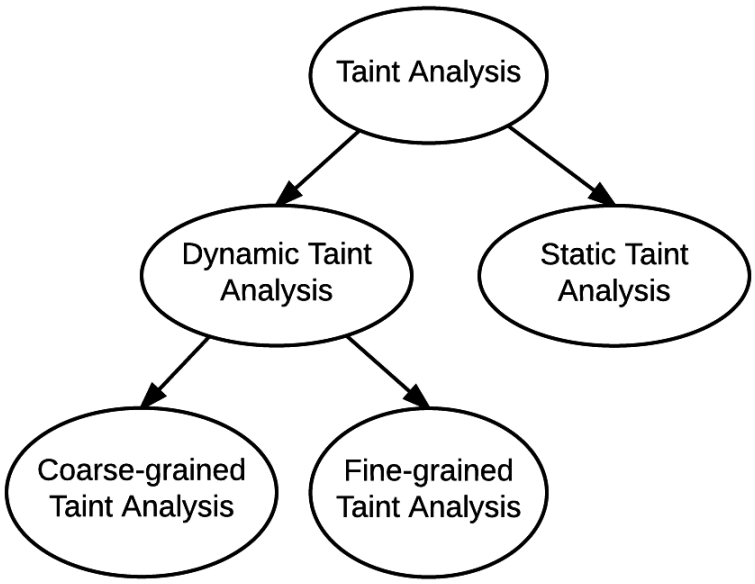


Figure 4: Taint analysis approaches

2.3.1 Static Taint Analysis

Static taint analysis techniques are widely used to detect common vulnerabilities in web application systems like SQL injection attacks, format string bugs, and input validation errors. Static analysis detects dependencies instead of vulnerabilities [2]. In the case of an SQL injection attack, when a query uses unchecked untrusted data in any manner, static analysis may give a warning. This may cause a problem if a dependency like that is an essential part of the system. This problem can be addressed by the use of the concept of *endorsement* [2]. The developer codes some validation checks that are run on that untrusted piece of data to make sure it does not contain any bad input. If it does, the data is ‘‘sanitized,’’ and then ‘‘endorsed’’ to suggest that the data is now safe to be used in sensitive operations.

In W. Chang et al. [10], the system performs static analysis on the program to recognize any violations of the security policies in place. If it finds violations of the security policies, an analysis is performed to identify all the positions in the code that demand dynamic analysis. The code is modified accordingly to enforce those policies dynamically by performing dynamic analysis. Thus, the amount of dynamic tracking needed is minimized by doing static analysis first.

One advantage of static analysis over dynamic analysis is that it finds all the potential vulnerabilities before the program is executed [3]. However, in comparison to dynamic analysis it lacks precision. Also, the programmer needs to understand what is safe before ‘‘sanitizing’’ the untrusted data, and then ‘‘endorse’’ it. However, there is no straightforward way to do this. The source of the input and the manner in which the input is used later in the program determine if the input is ‘‘safe’’ or not. Developers have to trace forward in the program to identify all the possible uses of the untrusted input in security-critical operations, which is a very tedious task.

Validation checks need to be performed at the point where the untrusted data is used and at the source, which requires tracing back from security sensitive operations to determine how its inputs were constructed. This requires manual examination of a large number of programming paths, which can be very difficult and a laborious task. [2].

2.3.2 Dynamic Taint Analysis

Newsome and Song [5] have implemented TaintCheck, a mechanism to detect format string vulnerabilities and buffer overruns by performing *dynamic taint analysis*. Dynamic taint analysis labels the data coming from the untrusted outside sources as *tainted*. The program is observed throughout its execution to track the flow of the tainted data. This help to keep a check on what new data becomes tainted. The usage of tainted data in security-critical operations is also tracked [11], as this may lead to an exploitation of security vulnerabilities. For example, when tainted data is used as a jump address or a format string, it may lead to attacks like buffer overrun or format string vulnerability.

The dynamic taint analysis can be performed in two ways:

1. Coarse-grained taint analysis
2. Fine-grained taint analysis

2.3.2.1 Coarse-Grained Taint Analysis

Coarse-grained taint analysis marks a program variable thats holds the incoming data from an untrusted source as *tainted*. Figure 5 shows an SQL query. The entire query string in the variable `$cmd` is marked as tainted, even if it derives only a portion of its value from the variable `name`, which stores untrusted incoming data [2]. Hence,

differentiating between a permissible query string, when the variable `name` contains a valid string, from an attack, when it contains SQL commands (for e.g. DROP, UPDATE, DELETE) and/or special characters is impossible.

```
$cmd = "SELECT price FROM products WHERE name = '".$name."'"
```

Figure 5: SQL query

Coarse-grained taint analysis rejects the tainted query string in variable `$cmd`. Thus, it successfully prevents a possible SQL injection attack; however, it also makes it impossible for a legitimate query string to be executed.

Perl tracks taint at a coarser granularity — that of a variable [2] [7]. In Perl, taint mode can be enabled by using the `-T` command line flag. In taint mode, Perl taints every data that comes from outside the code. The data needs to be explicitly untainted before it is used in a security sensitive context. The data is untainted after validation checks are performed to ensure that it is safe to use. Ruby also tracks taint at a coarser granularity. In Ruby, one can explicitly taint and untaint objects [6]. The objects need to be untainted before they are used in security-sensitive operations.

Detectors that implement coarse-grained taint analysis may result in more number of false positives [5], as they lack the ability to differentiate between an actual attack from a ''good'' data marked as tainted. Thus, they fail to provide detailed information about the vulnerability.

2.3.2.2 Fine-Grained Taint Analysis

Tainting is marking incoming data from untrusted sources as ''tainted.'' While coarse-grained taint analysis marks an entire string as tainted even if it derives only a part of it's value from tainted data, *fine-grained taint analysis*, marks individual

characters in the string as tainted. Nguyen-Tuong et al. [12] has modified the implementation of string datatype in a PHP interpreter such that it holds the taint information for string values at the granularity level of a character in the string. The taint information propagates across compositions, assignment operations, and function calls at the granularity level of individual character. Hence, the name *fine-grained* or *precise-grained taint analysis*.

Nguyen-Tuong et al. [12] also give a good example on fine-grained taint analysis. Figure 6 shows a piece of code. In the code, the string in the variable `$x` derives one part of its value from the user input and the other comes from a cookie. The values of `$_GET['name1']` and `$_COOKIE['name2']` are marked as tainted as they come from an untrusted source (let's consider they are Jane and John). After concatenation, the string in the variable `$x` and its taint markings (underlined) are: Hello Jane. I am John. All characters in the two underlined strings are marked as tainted.

Chang et al.'s implementation [10] tracks taint at the granularity of a byte. This enables fine-grained tracking of the data flow properties. This is essential because tracking the taint at the granularity level of a variable is not safe in C, which is a type-unsafe language.

```
$x = "Hello " .$_GET['name1']." . I am " .$_COOKIE['name2'];$
```

Figure 6: PHP string

Fine-grained tainting helps to detect and prevent the most common web application vulnerabilities like SQL & command injection attacks, cross-site scripting attacks, etc. When a security-critical function uses tainted data in a dangerous way, we can either reject the function call or sanitize the tainted data, endorse it (i.e. mark it untainted), and then use it safely.

2.4 Conclusion

Security vulnerabilities pose a great risk to the web application systems. It is important to have a mechanism in place that helps in detection and prevention of these vulnerabilities. Traditional access control policies fall inadequate in preventing these attacks. Taint analysis marks incoming data from an untrusted source as tainted. It tracks the introduction and flow of taint in the program. Static taint analysis involves writing validation checks on the user input to make sure that the input is safe. The programmer needs to code the validation checks, sanitize the input and endorse it to mark it as safe to use. Dynamic taint analysis tracks taint while the program is executing. Coarse-grained and fine-grained taint analysis are the two flavors of dynamic taint analysis. Fine-grained tainting is an attractive option as it tracks taint at the granularity level of individual characters in a string as apposed to coarse-grained tainting that tracks taint at the string level. This precision allows more sophisticated defenses against some of the attacks that we have discussed.

CHAPTER 3

Implementation details

This section discusses two implementations of taint analysis in Rhino 1.7.8 — coarse-grained taint analysis & fine-grained taint analysis. Rhino is a JavaScript engine written in Java. It is an open source software managed by the Mozilla Foundation [13].

3.1 Coarse-grained taint analysis

Coarse-grained taint analysis tracks taint information at the granularity level of a string. The idea is to mark incoming untrusted data as tainted. Taint information is transferred from one variable to another variable through program execution. When tainted data is used in a security critical operation, the program is crashed.

3.1.1 Functionalities

Three basic functionalities that coarse-grained taint analysis provide are -

1. `taint`
2. `untaint`
3. `isTainted`

`taint` marks an untrusted string read in from the outside environment as *tainted*. `isTainted` provides a check to see whether a string is tainted or untainted. `untaint` allows a tainted string to be marked as untainted. A tainted string should be marked untainted only after it has been properly sanitized. This is called ‘‘endorsement.’’ An endorsed variable is considered safe to be used in security-critical operations.

The three functionalities have been realized in my Rhino implementation by the `taint`, `untaint`, and `isTainted` methods respectively. All three methods are implemented in the top-level scope. All three functions accept string variables or string literals as a parameter. `taint` taints an input string and returns the tainted string. `untaint` marks an input string untainted and returns the untainted string which needs to be assigned to a variable. `isTainted` returns a boolean; it returns true if the incoming string is tainted else it returns false.

Figure 7 shows how coarse-grained taint analysis taints and untaints a string variable, and how to check whether a variable is tainted or untainted. In Figure 7 `taint(a)` returns a tainted string ''O'Reilly'', and assigns it to variable `b`. `isTainted(b)` returns true as the string `b` is referring to is marked as tainted. `b.replace(/'/g, "'")` sanitizes the tainted string in variable `b` by replacing a single quote with two single quotes, and `untaint` marks the sanitized string as untainted and returns it, which is assigned to variable `c`. Hence, `isTainted(c)` returns false.

```
var a = "O'Reilly";
var b = taint(a);
isTainted(b);    // returns true
var c = untaint(b.replace(/'/g, "''"));  // c = "O''Reilly"
isTainted(c);    // returns false
```

Figure 7: String tainting

3.1.2 Taint Propagation

Taint analysis tracks the flow of an untrusted string throughout the program execution. If a variable that contains a tainted string is used to initialize some other variable, then the newly initialized variable is also marked as tainted. Figure 8 shows an example of taint propagation through program execution.

```
var a = " Hello ";
var b = taint(a);
var c = b.bold().italics().fontsize(6).fontcolor("yellow");
var d = c;
var e = d.trim();
var f = e.replace("Hello","Hey");
var g = f + "Tejas";
var h = g.substr(0,3);
isTainted(h);          // returns true
```

Figure 8: Taint propagation

The string ‘‘Hello’’ is explicitly tainted using the `taint` function. Support has been added to all the string functions (like bold, italics, trim, replace, concat, substr, etc.) to handle taint propagation. So, if a function is called on a tainted string or a tainted string is passed as a parameter to any of these functions, the resulting string returned is also tainted.

3.1.3 Examples

This section discusses SQL injection attacks, and how coarse-grained taint analysis helps in preventing web application systems from these attacks.

3.1.3.1 SQL Injection Attack

In Figure 9 user entered name is accepted and assigned to the variable `name`. The name string is marked as tainted as it is an untrusted piece of data coming from outside the code. The variable `query` contains a dynamically formed query that uses the user entered id marked as tainted in the variable `tainted_name`.

In the case of coarse-grained taint analysis, the entire query string in the variable `query` is marked as tainted as it derives a portion from the tainted data. Hence, it

```
var name = document.getElementById("name").value;
var tainted_name = taint(name);    //taints string in id
query="select * from employee where ename ='"+tainted_name
   +"'";
```

Figure 9: Query string

gets difficult to distinguish an attack from a legitimate query string. Thus, it makes it all the more difficult to sanitize the query string and then execute it. Coarse-grained taint analysis thus rejects the query string if its tainted, and throws an error and crashes the system; if not the query is executed. Thus, this prevents the system from a possible SQL injection attack, but at the cost of crashing on many valid queries. Figure 10 shows how coarse-grained analysis may be used to handle the tainted query string.

```
if(isTainted(queryStr)){
  throw new Error("Can't execute a tainted query!");
}
else{
  exec_query(queryStr); //execute the query
}
```

Figure 10: Handling a tainted query

3.1.4 Limitations of coarse-grained taint analysis

Coarse-grained taint analysis tracks taint at the granularity level of a string. In Figure 8, two string are concatenated and the resulting string is stored in the variable `g`. The entire string in the variable `g` is marked as tainted as it derives a part of its value from the variable `f` that contains tainted data.

Thus, coarse-grained tainting makes it difficult to sanitize a tainted string, as the entire string is marked as tainted even if only a small portion of the string is derived

from tainted data. Hence, in the case of coarse-grained taint analysis the best way to prevent an attack is to suspend the execution, or crash the system whenever a tainted string is passed to a security critical operation.

Coarse-grained taint analysis also results in high false positives, as it is hard to sanitize the tainted string, even in the case of a legitimate user input. In contrast, fine-grained taint analysis works at the granularity level of a character in a string. It thus helps us keep track of the exact characters in the string are actually tainted.

3.2 Fine-grained taint analysis

Unlike coarse-grained taint analysis that tracks taint at the granularity level of a string, fine/precise grained taint analysis tracks taint at the granularity level of a character in a string. Hence, it gets easier to sanitize a tainted string and mark it as untainted to indicate that it can be safely used in security-sensitive operations.

3.2.1 Functionalities

The fine grained taint analysis API extends the coarse-grained taint analysis API with `taintedRegions` and `sanitize` functions.

In fine-grained taint analysis the function `taint` works a little differently than it does in coarse-grained taint analysis. Fine-grained tainting taints individual characters in a string, and keeps track of which characters are tainted. The `untaint` function works exactly like it does in coarse-grained taint analysis. It is used to untaint a tainted string after it is properly sanitized. This is called ‘‘endorsement.’’ An endorsed variable is considered safe enough to be used in security-critical operations. The `isTainted` function provides a check to see if either the entire string or region(s)/subset(s) of the string are tainted. The `taintedRegions` function returns an array containing

left & right boundaries of tainted region(s)/subset(s) in a string. One can iterate over the tainted regions, sanitize them, and construct an untainted string that is safe to use. If called on an untainted string, the function returns an empty array. The `sanitize` function is introduced later, which allows string sanitization for different attacks in an easier and a cleaner way, thus eliminating the need for duplicate code at multiple places. It takes two arguments — a tainted string, and a callback function to be applied on the untainted string to sanitize it. The function internally uses the `taintedRegions` function to get all the tainted subsets in the string argument, sanitizes them, and constructs a well-formed, sanitized, and safe-to-use string that is untainted. It then returns the string. The callback function is specific to the attack at hand, thus allowing the `sanitize` function to be used to prevent the system from multiple possible attacks.

Figure 11 shows how fine-grained taint analysis taints and untaints a string. It also shows how to check whether a string is tainted or not, and how to sanitize a string that is marked as tainted.

```
var a = taint("Hi") + " Jane " + taint("Doe!")
isTainted(a.substring(3,7)); // returns false
taintedRegions(a);       // returns [[0, 1], [8, 11]]
//returns a sanitized string
var b = sanitize(a,function(s){return s.replace(/'/g,"''");})
isTainted(b);            // returns false
```

Figure 11: String tainting

In Figure 11, variable `a` contains a string ‘‘Hi Jane Doe’’ with character ‘H’, ‘i’, ‘D’, ‘o’, ‘e’, and ‘!’ marked as tainted. `isTainted(a.substring(3,7))` returns false as the substring ‘‘Jane’’ does not contain tainted characters. `taintedRegions(a)` returns [[0, 1], [8, 11]], essentially an array of left & right boundaries of all the tainted regions in the

string in variable `a`. `sanitize(a,function(s){return s.replace(/'/g,"''");})` sanitizes all the taintedRegions in the string ''Hi Jane Doe'', and returns a reconstructed well-formed sanitized untainted string back, which is assigned to variable `b`. Hence, `isTainted(b)` returns false. The callback function is specific to the context the string is going to be used in. The callback function that is passed as a parameter to the `sanitize` function in the figure sanitizes the string for use in a SQL query.

3.2.2 Taint Propagation

Taint propagation in fine-grained taint analysis works similarly to how taint propagates in coarse-grained taint analysis. Figure 12 shows an example of taint propagation through program execution in fine-grained taint analysis.

```
var a = taint("  Hi") + " John " + taint("Doe!  ");
var b = a.trimLeft();
var c = b;
var d = c.trimRight();
var e = "Hello! " + d;

var f = e.substr(0,6);              // f = "Hello!"
isTainted(f);                       // returns false
taintedRegions(f);                  // returns an empty array

var g = e.substr(10,e.length);      // g = "John Doe!"
isTainted(g);                       // returns true
taintedRegions(g);                  // returns [[5, 8]]
```

Figure 12: Taint propagation

The variable `a` contains a string ''Hi John Doe!'' with characters 'H', 'i', 'D', 'o', 'e', and '!' explicitly marked as tainted. Taint support has been added to all the string functions (like trimLeft, trimRight, replace, concat, substr, etc.). So, in the case of either a function being called on a tainted string or a tainted string being passed as a

parameter to a function, the characters in the resulting string are marked as tainted. The variable `f` contains the string ‘‘Hello!’’, a substring of the tainted string in variable `e`. However, none of the characters in the string ‘‘Hello!’’ are marked as tainted. Hence, `isTainted(f)` returns false and `taintedRegions(f)` returns an empty array. The variable `g` contains a substring ‘‘John Doe!’’ with characters ‘D’, ‘o’, ‘e’, and ‘!’ marked as tainted. Hence, `isTainted(g)` returns true and `taintedRegions(g)` returns [[5,8]], an array containing the left and right boundary of the only tainted region in the string.

3.2.3 Examples

This section discusses two attacks — SQL injection and cross-site scripting, and how fine-grained taint analysis helps in preventing web application systems from these attacks.

3.2.3.1 SQL Injection Attack

Let’s consider the same example as discussed in 3.1.3.1. In the case of fine-grained taint analysis, only the characters in the query string in the variable `query` that come from outside the code are marked as tainted. A user enters something like ‘‘O’Reilly’’ for name. With ‘‘O’Reilly’’ as the user entered name, the query string appears like:

*select * from employee where ename = "O’Reilly";*

with characters ‘O’, ‘’’, ‘R’, ‘e’, ‘i’, ‘l’, ‘l’, and ‘y’ marked as tainted. The `taintedRegions` function is used to sanitize the tainted query. Figure 13 shows how fine-grained analysis handles a tainted query. We iterate over the tainted regions in the query string. The `idxInQuery` keeps tracks of the current index in the query.

```
if(isTainted(query)){
  let idxInQuery = 0;
  let taintedregions = taintedRegions(query);
  for (i = 0; i < taintedregions.length; i++) {
    let taintLBound = taintedregions[i][0];
    let taintRBound = taintedregions[i][1];

    untaintedQuery += untaintQuery(query,idxInQuery,
       taintLBound,taintRBound);
    idxInQuery = taintRBound + 1;
  }
  if(idxInQuery != queryLen)
    untaintedQuery += query.substring(idxInQuery,queryLen);

  exec_query(untaintedQuery); //execute the untainted query
}
else{ exec_query(query); } //execute the query

function untaintQuery(){
  if(taintLBound===0)
    untaintedStr = "";
  else
    untaintedStr = query.substring(indexInQuery,taintLBound);

  taintedSubStr = query.substring(taintLBound,taintRBound+1);
  sanitizedStr = taintedSubStr.replace(/'/g, "''");

  return untaintedStr + sanitizedStr;
}
```

Figure 13: Untainting a query string

The `taintLBound` & `taintRBound` correspond to the left and right boundaries of a tainted region respectively. The `untaintQuery` function takes the `query`, `idxInQuery`, `taintLBound`, and `taintRBound` as parameters. It sanitizes the tainted subquery indexed from `idxInQuery` to `taintLBound` by replacing single quotes with two single quotes, and returns the untainted substring. At the end of the `for` loop an untainted query is constructed in the variable `untaintedQuery`, which looks like:

*select * from employee where ename = "O''Reilly";*

The untainted query string can be safely executed. Thus, fine-grained taint analysis prevents the system from a possible SQL injection attack without having to crash the program execution.

3.2.3.2 Cross-Site Scripting

Flanagan [14, Section 13.6.4] gives a fine example of a cross-site scripting attack. Consider the web-page in Figure 14. It uses JavaScript to greet the user by name:

```
<script>
var name = decodeURIComponent(window.location.search.
   substring(1)) || "";
document.write("Hello " + name);
</script>
```

Figure 14: Sample web-page

`window.location.search` return the contents of the page URL after the question mark. The dynamically generated content is then written to the document using `document.write()`. The URL used to access the page looks like:

`http://www.example.com/greet.html?John`

The user is greeted as ''Hello John''. However, an attacker can inject a script after the question mark which can lead to an unexpected behavior. For example, when the URL is invoked with this (%3C is code for < & %3E is code for >):

`http://www.example.com/greet.html?%3Cscript%3Ealert('!')%3C/script%3E`

It displays a dialogue box. This is a very simple example of cross-site scripting.

In the case of fine-grained taint analysis, all the characters in the string in the variable `name` are marked as tainted. The `taintedRegions` function is used to sanitize the tainted string. Figure 15 shows how fine-grained analysis handles the tainted string.

```
if(isTainted(name)){
  let idxInStr = 0;
  let taintedregions = taintedRegions(name);
  for (i = 0; i < taintedregions.length; i++) {
    let taintLBound = taintedregions[i][0];
    let taintRBound = taintedregions[i][1];

    untaintedName += untaintString(name,idxInStr,taintLBound,
       taintRBound);
    idxInStr = taintRBound + 1;
  }
  if(idxInStr != nameLen)
    untaintedName += name.substring(idxInStr,nameLen);

  document.write("Hello " + untaintedName);
}
else{ document.write("Hello " + name); }

function untaintString(){
  if(taintLBound===0)
    untaintedStr = "";
  else
    untaintedStr = name.substring(idxInStr,taintLBound);

  taintedSubStr = name.substring(taintLBound,taintRBound+1);
  sanitizedStr =
        taintedSubStr.replace(/</g,"&lt;").replace(/>/g,"&gt;");

  return untaintedStr + sanitizedStr;
}
```

Figure 15: Untainting a user input string

As one can see, the code to sanitize a string in the cross-site scripting attack is very similar to the code to sanitize the query string in an SQL injection attack 3.2.3.1. The major difference is the actual method of sanitization, highlighted in bold in figures 13 and 15. In case of SQL injection it is done as `replace(/'/g, "''")`, while in the case of cross-site scripting it is done as `replace(/</g, "<").replace(/>/g, ">")`. Hence, the better way would be to generalize the code, and write a single `sanitize` function that can sanitize the strings in case of both types of attacks.

3.2.4 Sanitize Function

The `sanitize` function aims to reduce code duplication by generalizing the code to sanitize the tainted strings. It takes 2 arguments — a tainted string, and a callback function to sanitize the tainted string. Figure 16 shows the `sanitize` function.

```
function sanitize(str, callback){
  let idxInStr = 0;
  let taintedregions = taintedRegions(str);
  for (i = 0; i < taintedregions.length; i++) {
    let taintLBound = taintedregions[i][0];
    let taintRBound = taintedregions[i][1];

    if(taintLBound===0) untaintedStr = "";
    else untaintedStr = str.substring(idxInStr,taintLBound);

    taintedStr = str.substring(taintLBound,taintRBound+1);
    sanitizedStr = callback.call(this, taintedStr);
    untaintedStr = untaintedStr + sanitizedStr;
    idxInStr = taintRBound + 1;
  }
  if(idxInStr!=strLen) untaintedStr+=str.substring(idxInStr);
  return untaintedStr;
}
```

Figure 16: Sanitize function

Figure 17 shows how the `sanitize` function can be used to sanitize a tainted query in the case of an SQL injection attack, and a tainted string in the case of a cross-site scripting attack. In the case of an SQL injection attack, the callback function replaces all the single quotes with two single quotes. The `sanitize` function returns a sanitized and untainted query string, which is safe to use. The untainted query string in the variable `untaintedQuery` could then be executed safely.

```
//Example - SQL injection
if(isTainted(query)){
  untaintedQuery = sanitize(query,function(s){ return s.
     replace(/'/g, "''");});
  exec_query(untaintedQuery);
}
else{
  exec_query(query);
}

//Example - Cross-site scripting
if(isTainted(name)){
  sanitizedName = sanitize(name,function(s)\{ return s.
     replace(/</g, "\&lt;").replace(/>/g, "\&gt;");\});
  document.write("Hello " + sanitizedName);
}
else{
  document.write("Hello " + name);
}
```

Figure 17: Sanitize function call

In the case of a cross-site scripting attack, the callback function replaces the angular brackets in the tainted string in the variable `name` with their corresponding HTML entities. This helps to escape and deactivate any HTML tags in the tainted string. The sanitized and untainted string in the variable `sanitizedName` could then be used safely in `document.write()`.

CHAPTER 4

Performance Tests

In order to quantify the performance overhead caused because of coarse-grained and fine-grained taint analysis, I modified the popular JavaScript performance benchmark SunSpider [15]. The JavaScript files in the SunSpider benchmark test suite were interpreted and executed on both the variants of Rhino — one with support added for coarse-grained taint analysis and the other with support added for fine-grained taint analysis. The results were compared against the baseline. These tests were performed on a Mac Book Pro with one 2.7 GHz dual-core Intel Core i5 processor, 8 GB 1867 MHz DDR3 RAM, and an Intel Iris Graphics 6100 graphics processor with 1536 MB of memory.

The SunSpider benchmark test suite contains tests that are balanced between different areas of the language and different types of code. It consists of tests on String & RegExp manipulation, mathematical & bit operations, and other tests like cryptography and code decompression. It runs each test multiple times and determines an error range with a confidence interval of 95% [16]. Hence, claims to be statistically sound.

I have commented out 5 tests from the test suite due to errors in code interpretation on both my variants of Rhino. Two experiments were conducted as a part of the performance tests. The first experiment was to run all the tests in the test suite on all three variants of Rhino, with no variable explicitly marked as tainted. This gave a comparison of the performance overhead caused by the implementation of fine-grained taint analysis and coarse-grained taint analysis in my two variants of

Rhino respectively against the unmodified version of Rhino. Table 1 shows the results of the experiment. It compares the time taken in milliseconds for each test to run on all three variants of Rhino. As observed from the table, in comparison to the Rhino base version, the version with coarse-grained taint analysis adds an overhead of $\approx 4\%$, while the fine-grained taint analysis version adds an overhead of $\approx 22\%$.

Table 1: Performance Test Results With No Tainted Variables

Test	**Rhino Base**		**Rhino Coarse**		**Rhino Fine**	
	Mean	**95% CI**	**Mean**	**95% CI**	**Mean**	**95% CI**
3d	551.6ms	±88.6%	573.7ms	±122.7%	573.1ms	±83.2%
cube	189.6ms	±114.8%	172.7ms	±76.9%	159.1ms	±92.3%
morph	211.9ms	±69.9%	240.4ms	±85.2%	245.3ms	±48.1%
raytrace	150.1ms	±82.8%	160.6ms	±231.6%	168.7ms	±140.6%
access	890.1ms	±41.1%	804.7ms	±67.0%	838.7ms	±48.5%
binary-trees	69.7ms	±64.3%	71.7ms	±142.5%	81.6ms	±74.6%
fannkuch	465.1ms	±23.5%	406.3ms	±52.6%	421.4ms	±40.9%
nbody	178.6ms	±40.6%	177.0ms	±82.1%	196.6ms	±65.2%
nsieve	176.7ms	±86.7%	149.7ms	±55.5%	139.1ms	±61.4%
bitops	904.9ms	±24.1%	1015.6ms	±12.7%	1061.9ms	±62.0%
3bit-bits-in-byte	152.9ms	±93.6%	154.1ms	±50.2%	145.9ms	±35.2%
bits-in-byte	263.6ms	±14.1%	234.4ms	±28.1%	241.3ms	±130.3%
bitwise-and	244.0ms	±9.7%	405.6ms	±2.5%	432.9ms	±78.9%
nsieve-bits	244.4ms	±10.0%	221.4ms	±4.6%	241.9ms	±46.2%
controlflow	93.3ms	±119.4%	84.4ms	±13.7%	101.1ms	±58.4%
recursive	93.3ms	±119.4%	84.4ms	±13.7%	101.1ms	±58.4%
crypto	266.3ms	±35.2%	267.9ms	±58.43%	766.7ms	±50.0%
md5	172.7ms	±35.8%	174.9ms	±80.6%	594.9ms	±50.7%
sha1	93.6ms	±34.1%	93.0ms	±17.3%	171.9ms	±49.8%
date	108.7ms	±124.0%	166.3ms	±109.8%	192.1ms	±106.3%
format-tofte	108.7ms	±124.0%	166.3ms	±109.8%	192.1ms	±106.3%
math	524.4ms	±12.9%	550.0ms	±35.4%	585.7ms	±49.7%
cordic	254.9ms	±16.8%	249.3ms	±37.7%	267.6ms	±41.9%
partial-sums	173.3ms	±8.1%	206.9ms	±33.8%	223.4ms	±46.6%
spectral-norm	96.3ms	±14.5%	93.9ms	±33.3%	94.7ms	±84.0%
string	249.6ms	±73.2%	263.7ms	±66.7%	264.7ms	±85.5%
fasta	145.0ms	±55.9%	196.0ms	±60.5%	198.9ms	±61.3%
unpack-code	104.6ms	±98.0%	67.7ms	±88.1%	65.9ms	±167.0%
total	3588.9ms	±44.7%	3726.3ms	±55.4%	4384.1ms	±56.0%

In the second experiment in the *crypto-md5* test was modified to introduce taint in an incremental fashion. The first test kicked off with ten calls to the *crypto-md5* test with no taint explicitly introduced. The subsequent tests are conducted with +1 tainted calls than the previous test, until all the ten calls are tainted. The Figure 18a shows Test0 with no tainted calls, and Figure 18b shows Test1 with one tainted call out of all the ten calls. Similarly, Test2 with two tainted calls, and so on until Test10 with all the calls tainted.

```
// Test 0
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
```

(a) Test 0

```
// Test 1
var md5Output = hex_md5(
    taint(plainText));
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
var md5Output = hex_md5(
    plainText);
```

(b) Test 1

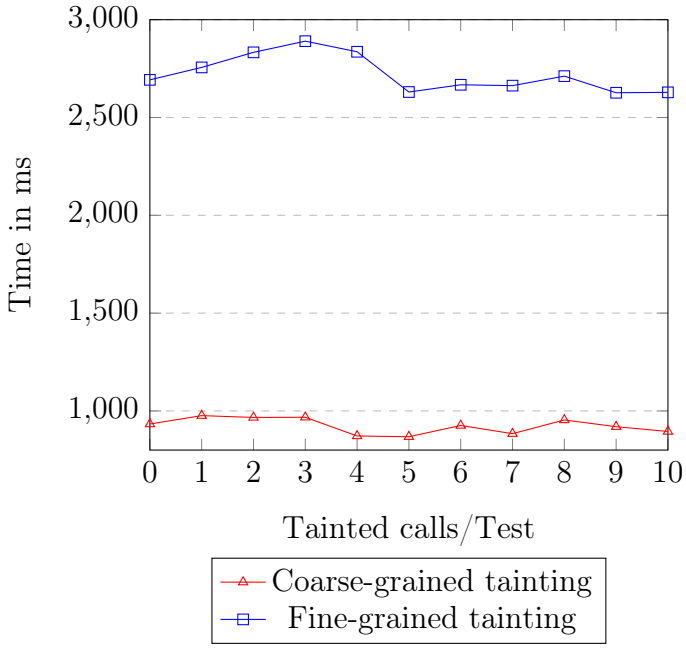
Figure 18: Incremental taint introduction

The tests were run on my two implementations of Rhino to compare the performance overhead caused on both variants when taint is introduced in the program.

Table 2 shows the results of the second experiment.

Table 2: Taint Performance Test Results

Tainted variables	**Rhino Coarse**		**Rhino Fine**	
	Mean	**95% CI**	**Mean**	**95% CI**
0	933.2ms	±83.5%	2692.4ms	±81.6%
1	976.0ms	±69.7%	2756.0ms	±80.0%
2	966.8ms	±82.5%	2833.0ms	±44.0%
3	967.6ms	±107.0%	2890.2ms	±40.3%
4	872.0ms	±162.1%	2836.0ms	±79.7%
5	868.2ms	±164.8%	2630.2ms	±61.1%
6	925.6ms	±150.9%	2667.2ms	±56.7%
7	883.8ms	±133.7%	2662.8ms	±51.4%
8	953.8ms	±129.9%	2711.4ms	±53.4%
9	919.4ms	±109.5%	2626.4ms	±43.2%
10	895.0ms	±94.2%	2628.6ms	±50.1%



As seen in Table 2, the execution time of the tests on both the versions of Rhino go up & down, and does not increase in proportion with the increase in the number of tainted calls. However, all the tests on fine-grained taint analysis version of Rhino take $\approx 3X$ time to execute as compared to the coarse-grained taint analysis version.

CHAPTER 5

Conclusion & Future Work

In the paper, I have provided an insight into how dynamic taint analysis could be deployed to tackle prevalent attacks on web application systems. Most common attacks on the web application systems like SQL injection, cross-site scripting, and format string attacks are caused due to unchecked user input. The paper discusses conventional techniques like static analysis, access control policies, and code reviews and how they are inadequate to prevent these attacks.

Dynamic taint analysis provides a mechanism to track taint while the program is in execution. The data coming from outside the code is not trustworthy. Hence, it is marked as tainted, and tracked throughout the program execution. In this project, I implemented dynamic taint analysis in Rhino, a JavaScript engine. JavaScript being one of the most favored languages for web application is prone to the aforementioned attacks. Hence, it is crucial to address these vulnerabilities.

I have shown two flavors of dynamic taint analysis namely coarse-grained and fine/precise-grained taint analysis, implemented on two variants of Rhino respectively. Coarse-grained taint analysis tracks taint information at the granularity level of a string. Fine-grained taint analysis tracks taint at the granularity level of the characters in the string. Fine-grained taint analysis provides more control in terms of being able to sanitize a tainted string than coarse-grained taint analysis. Although both the approaches are equally good at preventing the system from a possible attack, fine-grained taint analysis edges out coarse-grained taint analysis in terms of number of false positives, and its ability to help library writers sanitize strings.

The paper showed that fine-grained taint analysis in general incurs more overhead as compared to coarse-grained taint analysis, as expected. It causes an overhead of $\approx 22\%$, and coarse-grained taint analysis causes an overhead of $\approx 4\%$ in comparison to the Rhino base version, when no variable is tainted. The paper also made an observation that fine-grained taint analysis is $\approx 3X$ slower than coarse-grained taint analysis, when taint is introduced in the program in an incremental fashion.

With the ever increasing popularity of web applications, and the prevalent security vulnerabilities that pose a threat to them, implementing language level features to prevent these security vulnerabilities is essential. The paper shows that implementing dynamic taint analysis as a language feature proves more effective in preventing these attacks as compared to the conventional techniques. While coarse-grained taint analysis stops the attack by crashing the system after an attack is detected, fine-grained taint analysis provides more control in preventing the attack. It enables a tainted string to be sanitized and used safely later on, thus preventing the system from a possible attack without having to crash it. While the paper mainly talks about preventing the web applications from SQL injection and cross-site scripting, both fine-grained and coarse-grained dynamic taint analysis approaches could be easily extended to Eval and other constructs in the language to prevent the systems from eval injection and format string attacks. In the future, I would also like to dig deeper into the code to identify optimizations that can be done to the code to improve the performance.

LIST OF REFERENCES

[1] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, ''Taj: Effective taint analysis of web applications,'' in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 87--97.
Available: http://doi.acm.org/10.1145/1542476.1542486

[2] W. Xu, S. Bhatkar, and R. Sekar, ''Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks,'' in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006.
Available: http://dl.acm.org/citation.cfm?id=1267336.1267345

[3] V. B. Livshits and M. S. Lam, ''Finding security vulnerabilities in java applications with static analysis,'' in *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, ser. SSYM'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 18--18.
Available: http://dl.acm.org/citation.cfm?id=1251398.1251416

[4] S. Wei and B. G. Ryder, ''Practical blended taint analysis for javascript,'' in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 336--346.
Available:http://doi.acm.org/10.1145/2483760.2483788

[5] J. Newsome and D. Song, ''Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software,'' 2005. [Online].
Available: http://bitblaze.cs.berkeley.edu/papers/taintcheck-full.pdf

[6] ''Help and documentation for the ruby programming language,'' http://ruby-doc.org/core-2.4.1/Object.html, [Online; accessed 17-April-2017].

[7] ''Perl programming documentation,'' https://perldoc.perl.org/perlsec.html, [Online; accessed 17-April-2017].

[8] T. Pietraszek and C. V. Berghe, ''Defending against injection attacks through context-sensitive string evaluation,'' in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2005, pp. 124--145.
Available: http://tadek.pietraszek.org/publications/pietraszek05_defending.pdf

[9] G. Smith, ''Principles of secure information flow analysis,'' in *Malware Detection*. Springer, 2007, pp. 291--307.
Available: http://users.cis.fiu.edu/~smithg/papers/sif06.pdf

[10] W. Chang, B. Streiff, and C. Lin, ''Efficient and extensible security enforcement using dynamic data flow analysis,'' in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 39--50.
Available: http://doi.acm.org/10.1145/1455770.1455778

[11] J. Clause, W. Li, and A. Orso, ''Dytan: A generic dynamic taint analysis framework,'' in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 196--206.
Available: http://doi.acm.org/10.1145/1273463.1273490

[12] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, ''Automatically hardening web applications using precise tainting,'' in *IFIP International Information Security Conference*. Springer, 2005, pp. 295--307.
Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.135.1565&rep=rep1&type=pdf

[13] Wikipedia, ''Rhino (javascript engine) --- wikipedia, the free encyclopedia,'' 2016, [Online; accessed 19-November-2016]. [Online].
Available: https://en.wikipedia.org/w/index.php?title=Rhino_(JavaScript_engine)&oldid=750360385

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

[15] F. Pizlo, ''Announcing SunSpider 1.0,'' 2013, [Online; accessed 01-May-2017].
Available: https://webkit.org/blog/2364/announcing-sunspider-1-0/

[16] ''SunSpider 1.0.2 JavaScript Benchmark,'' https://webkit.org/perf/sunspider/sunspider.html, [Online; accessed 01-May-2017].