NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Evaluating Taint Analysis Tools for JavaScript

Marios G. Papamichalopoulos

Supervisors: **Dimitris Mitropoulos,** Adjunct Faculty
Alexis Delis, Professor

ATHENS

JULY 2019

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αξιολογώντας Εργαλεία Ανάλυσης JavaScript Προγραμμάτων

Μάριος Γ. Παπαμιχαλόπουλος

Επιβλέποντες: **Δημήτρης Μητρόπουλος,** Επισκέπτης Καθηγητής
Αλέξης Δελής, Καθηγητής

ΑΘΗΝΑ

ΙΟΥΛΙΟΣ 2019

BSc THESIS

Evaluating Taint Analysis Tools for JavaScript

Marios G. Papamichalopoulos
S.N.: 1115201400149

SUPERVISORS: **Dimitris Mitropoulos,** Adjunct Faculty
Alexis Delis, Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αξιολογώντας Εργαλεία Ανάλυσης JavaScript Προγραμμάτων

Μάριος Γ. Παπαμιχαλόπουλος
Α.Μ.: 1115201400149

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Δημήτρης Μητρόπουλος,** Επισκέπτης Καθηγητής
Αλέξης Δελής, Καθηγητής

ABSTRACT

In the context of this BSc thesis, we have examined a number of scientific tools that perform *taint analysis* for programs written in the JavaScript programming language.

Taint analysis is defined as a type of analysis which concludes if points of the program that act as entry points for sensitive data are dangerous for the application, by tracking the flow of such data throughout the program. Such points are called *taint sources*.

Specifically, taint analysis marks as *tainted* the variables which have been affected by user input and tracks them until they reach a sensitive method, called *sink*. If a tainted variable reaches such a point, without being properly sanitized first, a vulnerability is reported. *Tainting* is the association of some kind of label or mark to sensitive data that allows the tracking of their flow throughout the program as well as the propagation of *taint* to the variables they come across.

The purpose of this research is the thorough research of scientific tools that perform such kind of analyses for programs written in JavaScript. We hereby present a collection of frameworks and approaches, which developers and enterprises may incorporate to their defense arsenal, for the inspection of the client-side code of their web applications, thus negating possible web attacks.

SUBJECT AREA: Taint Analysis
KEYWORDS: Security, JavaScript, Client-side Vulnerabilities, Static Taint Analysis, Dynamic Taint Analysis

ΠΕΡΙΛΗΨΗ

Η παρούσα μελέτη που διεξήχθη μέσα στα πλαίσια Πτυχιακής Εργασίας περιλαμβάνει την καταγραφή επιστημονικών εργαλείων που πραγματοποιούν *taint analysis* στην προγραμματιστική γλώσσα JavaScript.

Το taint analysis ορίζεται ως ένα είδος ανάλυσης, το οποίο συμπεραίνει αν τα σημεία του προγράμματος που ενεργούν ως σημεία εισαγωγής ευαίσθητων δεδομένων αποτελούν κίνδυνο για την εφαρμογή, παρατηρώντας τη ροή τέτοιων δεδομένων μέσα στο πρόγραμμα. Τέτοια σημεία ονομάζονται *πηγές (taint sources)*.

Συγκεκριμένα, ένα taint analysis χαρακτηρίζει ως *«στιγματισμένες» (tainted)* τις μεταβλητές που έχουν επηρεαστεί από δεδομένα που εισάγει ο χρήστης και τις ιχνηλατεί μέχρι να δει αν φτάνουν σε κάποια ευπαθή μέθοδο, που ονομάζεται *καταβόθρα (sink)*. Αν μία αμαυρωμένη μεταβλητή εισέλθει σε ένα τέτοιο σημείο, χωρίς να έχει *εξαγνιστεί (sanitize)* πρώτα, τότε χαρακτηρίζεται ως ευπαθής. Ο *στιγματισμός (tainting)* είναι η συσχέτιση κάποιου είδους σημαδιού ή ετικέτας στα ευαίσθητα δεδομένα που επιτρέπει την ανίχνευση της ροής τους μέσα στο πρόγραμμα καθώς και την διάδοση της *μόλυνσης (taint)* σε μεταβλητές που συναντούν.

Ο σκοπός αυτής της έρευνας είναι η διεξοδική έρευνα επιστημονικών εργαλείων που εκτελούν τέτοιου είδους αναλύσεις σε προγράμματα γραμμένα σε JavaScript. Παρουσιάζουμε μία συλλογή εργαλείων και προσεγγίσεων, που προγραμματιστές ή οργανισμοί μπορούν να ενσωματώσουν στο αμυντικό οπλοστάσιο τους για την επιθεώρηση του κώδικα της πλευράς πελάτη των διαδικτυακών εφαρμογών τους, αντικρούοντας, έτσι, πιθανές διαδικτυακές επιθέσεις.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Taint Analysis
ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ασφάλεια, JavaScript, Ευπάθειες πλευράς πελάτη, Static Taint Analysis, Dynamic Taint Analysis

Dedicated to my family

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Mr. Dimitris Mitropoulos, for his guidance, cooperation and valuable contribution and assistance for the completion of this thesis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The basis of this research stemmed from my passion for Computer Security. During the spring semester, I took this subject and became thrilled of the fact that from someone's mistake or misuse of programming libraries, an experienced attacker can wreak havoc to the application and to every person that utilizes its services.

As the Web keeps expanding and becomes more embellished with scripts responsible for enhancing the User Experience, there will be a greater need to check for security violations. It is evident to find out frameworks that will help the work of the developers and enterprises. This was the idea leading to the conduction of this research.

I would have never achieved this research had it not been for my professor, Dimitris Mitropoulos. As a teacher, he was able to impart his knowledge and passion for computer security to me and show me new techniques and ways, such as taint analysis. Also, I would like to thank my professor Alexis Delis for helping me and guiding me through all these years.

1. INTRODUCTION

The general increase of web applications has changed the standards regarding user experience (UX). The World Wide Web (WWW) is structured by an abounding number of websites that provide many services and utilities to the users. This gives them the freedom to browse in multiple applications, when trying to perform a task, until they find one that fits their needs. This has led the enterprises and companies to upgrade the UX using JavaScript scripts, in order to appeal the customers and increase their profit.

Why would someone choose JavaScript over another programming language? JavaScript has become prevalent concerning client-side code among web developers because of some of its key characteristics. These are its ease of use, flexibility and power. A Web developer does not need to precompile their code or install a plugin. They have the ability to test their scripts immediately and manipulate the Hypertext Markup Language (HTML) Document Object Model (DOM). In addition to this, there are a lot of extra libraries built on top of JavaScript, providing even more diversity and ease for the code writers.

The upgrade of UX has one major drawback. The extra code, added in the web application to ensure this property, may become responsible for potential security violations. Not only can a malicious user easily map the infrastructure of the application, but they can also find vulnerabilities which originate from inexperienced developers using JavaScript libraries incorrectly. A really experienced attacker can take advantage of these two and perform a series of attacks, with the most common being *Cross-Site Scripting (XSS)* [17] and a variety of *Injections* [18] [19].

According to the *Top 10 Application Security Risks* [1] list, created by Open Web Application Security Project (OWASP) in 2017, Injections are the number 1 threats with XSS attacks following six ranks bellow them.

Injections, as the name suggests, can trick the interpreter into executing unintended commands or accessing data without proper authorization. They can be really harmful to the server since an attacker can steal passwords, destroy databases, and forge new admin accounts to the application.

XSS flaws occur when the application uses untrusted data in a web page without proper sanitization or updates an existing web page with user-supplied data using a browser API that can create HTML or JavaScript. XSS may occur in three different types; Stored XSS, Reflected XSS and DOM-based XSS [17]. Such attacks allow the execution of scripts in the victim’s browser which can hijack user sessions, deface websites or redirect the user to malicious websites. All in all, an XSS attack aims in violating the Same-origin Policy, which is a concept in the web application model stating that a web application cannot execute scripts coming from another source.

Both of these attacks are based on the inexistent sanitization and validation of the user input. Especially, XSS, can be really dangerous, since its subtypes Reflected XSS and DOM based XSS may never reach the server validation code, so it is really hard for developers to realize that there is such a perilous security breach.

As mentioned, a potential security violation may result in destroying the integrity of an enterprise and most importantly harm benign every day users of the platform. That is why it has become a necessity to create automatic tools that expose such Client-Side Vulnerabilities (CSV). Bearing that in mind we have investigated a large number of scientific papers that present tools or approaches that perform a type of analysis, called *taint analysis* on the JavaScript language.

These tools share a common approach. They observe the entry points of sensitive information; they track the flow of data entering from such points throughout the program and report the potential security violations. In more depth, a taint analysis marks the unsanitized variables, which have been influenced by the user input, as "tainted" and tracks their flow to ensure they do not reach a vulnerable method or function that exposes them to the user. If they do, they are dangerous for the integrity of the program. In the analysis, the entry points are called *taint sources* and the exit points are called *sinks*. In most of the papers we have analyzed, a sanitization function between a taint source and a sink is called *downgrader*.

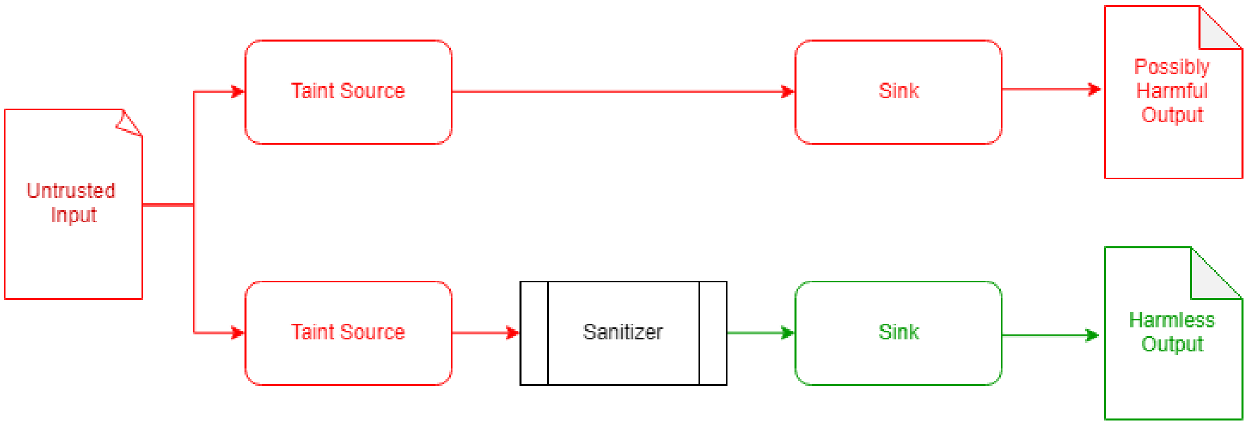


Figure 1: Taint Analysis Visualization

In order to explain the processes done by the frameworks and compare their differences we need to provide an explanation of some key types of analyses and how they operate, for example *data-flow analysis*. Also, we need to explain key data structures like the Abstract Syntax Tree (AST). Aho, *et al* [15] provide a very thorough description for these techniques.

1.1 Static vs Dynamic Analysis

We provide a brief description of static analysis frameworks and their differences with dynamic frameworks.

A *static analysis* or *static code analysis* is an analysis performed to the source code of the program after it has been compiled. A *dynamic* program *analysis* performs the analysis when the program is running by injecting instrumentation code to the subject program.

Respectively, static taint analysis and dynamic taint analysis are analyses focusing on exposing vulnerabilities on the application.

1.2 Data-Flow Analysis

Data-flow analysis refers to a set of techniques used to produce information regarding the flow of data along the execution of a program. It is mostly used for the optimization of a program during its compiling phase, since techniques like dead code elimination rely on the flow of a variable's value to conclude if it affects other variables or not. Likewise, it is used for the understanding of a program, like what type a function has or what is the resource usage in the program analyzed. This type of data-flow analysis is the one tools performing taint analysis use.

In a few words, data-flow analysis is important in the frameworks we are examining because it is the analysis responsible for finding the vulnerabilities. By tracking a dangerous untrusted input throughout the program, a tool can find out if it is a potential threat by reaching a sink.

1.3 Pointer Analysis

Pointer analysis is performed by a set of techniques used to pinpoint the variables or memory addresses, heap references or pointers point to during the execution of a program. It is a very important static code analysis part in the static taint analysis tools, due to the reference-type objects assignments which may propagate taint. Such tricky examples can be seen in the **Section 2** of the thesis. The tools we describe take care of them successfully.

1.4 Interprocedural vs Intraprocedural Analysis

Procedures are what we know as functions. In the terms of Object-Oriented Programming (OOP) procedures are known as methods.

Most compiler optimizations are performed on procedures one at a time. Such analyses are known as *intraprocedural analyses*. They assume that an invocation of a procedure may result in the worst possible result regarding the state of the variables or the stack. They are pessimistic analyses, since they assume the worst possible side effects.

An *interprocedural analysis*, though, performs on the whole program, examining information flow from the caller to the callee and back. It enabled more precise analysis information by using calling relationships among the procedures, through a Call Graph. It is used in the taint analysis process in some frameworks, like for example ANDROMEDA, detecting some tainted cases related to reference-type objects. It makes them the most complex among the two and thus the most accurate.

1.5 Call Graphs

Call Graphs describe the calls between procedures throughout the program. These relations are represented by a graph. Precisely, a call graph is a set of nodes and edges where nodes describe the procedures and edges describe the invocation of them.

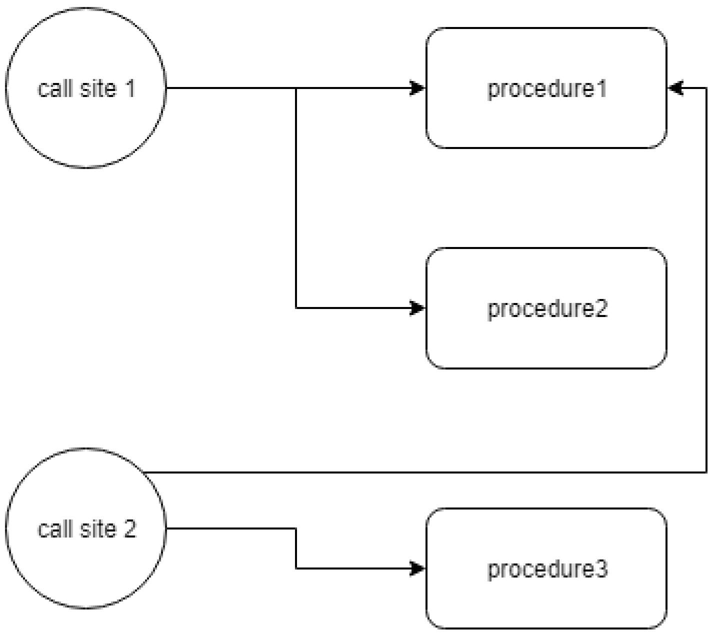


Figure 2: Simple Call Graph Visualization

1.6 Intermediate Representation

The front-end of a compiler constructs an *intermediate representation (IR)* of the source program, which the back-end uses to produce the final program. In a few words, IR is the link between both ends of the compiler. In a taint analysis for JavaScript, the front-end is JavaScript and parts of HTML while the back-end is the one doing the analysis. The two most important IRs are:

- **Trees**, *Abstract Syntax Trees (AST).* An AST is a tree representation of the structure of the source code of a program. During the syntax analysis, there are created nodes in the syntax tree to represent important programming and data structures.
- **Linear representations**, *three-address-code (TAC* or *3AC).* On the other hand, 3AC is a sequence of steps of the program. Compared to the AST there is no hierarchy in the structure. This representation is used for the optimization of the code, by breaking the program into blocks of 3AC to sequence of instructions without branches. No instruction can have more than one operator at its right side. For example, an expression like $x + y + z$ has to be translated into a sequence of two three-address instructions, where t_1 and t_2 are compiler generated names:

$$t_1 = y + z$$
$$t_2 = x + t1$$

Figure 3: 3AC/TAC example of two operators in expression

Other representations worth noting are *Control-Flow Graphs (CFG)* and *Static-Single Assignment form (SSA)*:

- **CFG** as the name states is a graph denoting all the paths that might be traversed through a program execution.
- **SSA** is an intermediate representation that is similar to the TAC IR. However, assignments in SSA are performed with variables with distinct names; hence the term static single assignment. This method of assigning exclusive names for each variable results in a problem when having an *if-then-else* statement. If a variable is part of both branches of an *if-then-else* statement and then that variable is assigned in another part of the program, what is the name it should have so that SSA is accomplished? Let's illustrate this with a simple example:

```
1. if (cond) {
2.     x = 1;
3. } else {
4.     x = 2;
5. }
6.
7. y = x + 1;
```

```
1. if (cond) {
2.     x_1 = 1;
3. } else {
4.     x_2 = 2;
5. }
6.
7. x_3 = o(x_1, x_2);
8. y = x_3 + 1;
```

Figure 4: SSA issue and solution

As we can observe from the left part of **Figure 4**, at the assignment in line 7, since variables in SSA form have different names, variable *y* would not know which to choose for its assignment, between the *x_1* and *x_2*. This is solved by a notational convention combining the two definitions of *x*.

Tools choose different IR for their solutions. They make sure that the ones they choose match their approach. It is common to have more than one IR for an analysis.

1.7 Paper Organization

We have separated our findings in four major categories, based on the type of the analysis the frameworks perform:

- The **Section 1** refers to tools based on static taint analysis frameworks.
- The **Section 2** of this thesis, respectively, presents tools based on dynamic taint analysis frameworks.
- The **Section 4** of the research introduces the combination of both types of analyses into a hybrid taint analysis, including both static and dynamic taint analysis.
- The **Section 5** makes a reference to frameworks that perform taint analysis in a different way than the standards. We believe they are worth mentioning since they provide an alternative approach.
- The **Section 6** of this thesis, at last, presents the conclusions made throughout our research.

Additionally, in the first two sections, **Section 1** and **Section 2**, we have added tools that perform analyses in general and are the foundation for other taint analysis tools. These are *T.J. Watson Libraries for Analysis* [3], *Jalangi* [4] and *Jalangi2* [5]. One can build a custom taint checker on these frameworks, since they provide the necessary mechanisms. We felt it was essential to present them in this research, since the reader can adopt a solid background and understand more clearly how the frameworks that use them operate.

In **Figure 5**, we present a map for the scientific tools and approaches we are going to describe in the thesis:

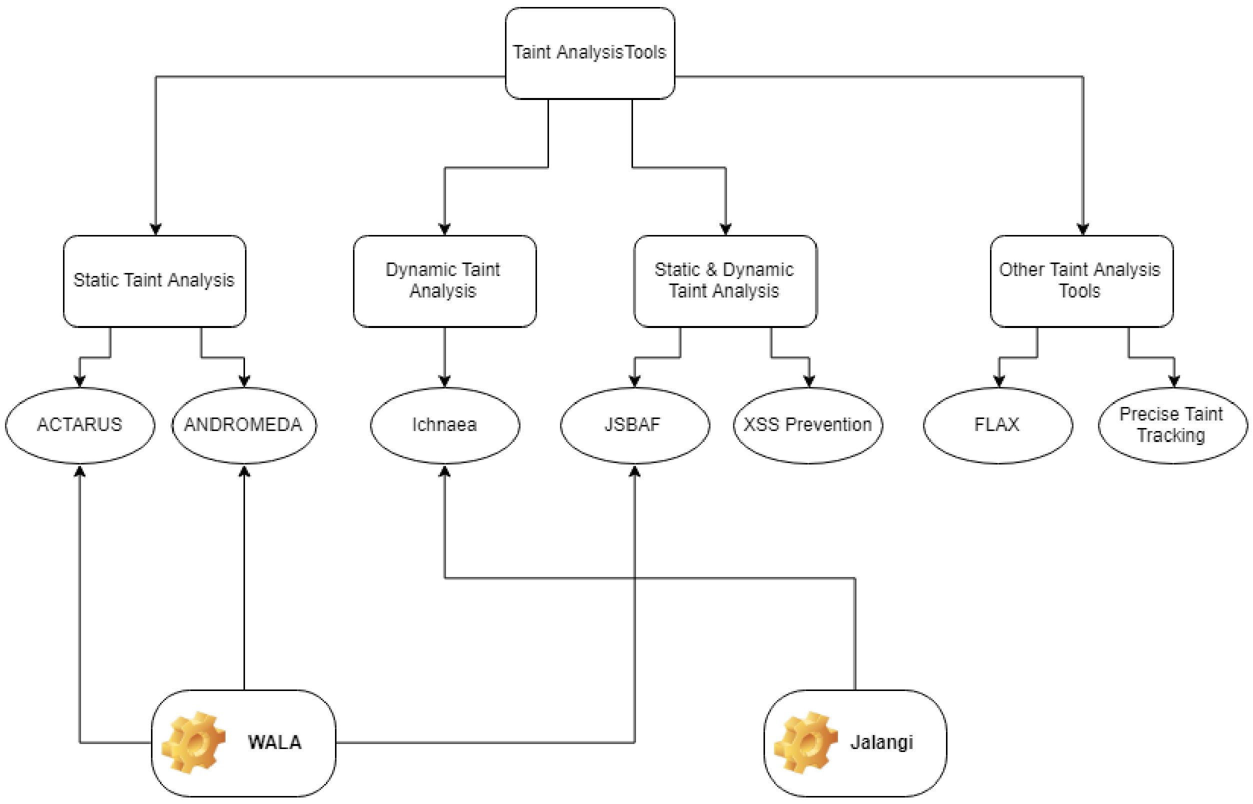


Figure 5: Thesis map

Bear in mind that JavaScript is a really difficult language to model because of its dynamic features (e.g. *eval*, *new operator*). Most of the taint analysis tools successfully confront them but there are cases of false warnings.

2. STATIC TAINT ANALYSIS TOOLS

We have identified three tools matching this category. In general, static analysis is a method of debugging by automatically examining source code before a program is run. Subsequently, static taint analysis is a method of exposing vulnerabilities in the program by evaluating its source code before it is executed.

Static analysis provides three major benefits, speed, depth and accuracy. Manual code reviewing for developers can be time consuming and error prone. By using automated tools, not only can developers find errors in their code, but they can also find their exact location since most static analysis tools pinpoint them. In a similar manner, they can cover every possible code execution path there is, which is something that manually may take absurd amount of time.

On the other hand, static analysis frameworks for JavaScript due to the flexible and dynamic nature of the language are short on number and provide a fair amount of false warnings. As mentioned, it is really difficult to model JavaScript for analysis without having some misleading alarms.

Before describing the frameworks, it would be wise to explain some key terms that influence static analyses and are responsible for the majority of the false warnings formed by a taint analysis:

- **Context insensitivity**: The analysis does not take into account the context when invoking a method. Hence, it merges together different execution contexts of the same method [2].

  In the following example, there is a function called *id* which echoes back its input. Also, there are two variables *y1* and *y2*. It can be observed that the first variable is assigned a simple benign string, while the second one is assigned user's input.

```
function id(x) {
   return x;
}

var y1 = id("hello");
var y2 = id(prompt());
document.write(y1);
```

Figure 6: Context insensitivity example

  Context-insensitive static taint analyses will handle *y1* as untrusted and produce a warning for potential vulnerability coming from the *document.write* sink. That will happen because of the merging of two different contexts to one. As a result, the analysis assumes that function *id* may pose a threat in general, due to the fact that it returns an untrusted value, that being of user input *y2*.

  There is clearly no vulnerability on the above code since the function *document.write* takes as a parameter the variable *y1* which is comprised by a completely safe string.

- **Flow insensitivity:** The analysis does not consider the order of the memory updates and instead calculates all possible memory update orders. The instruction sequence is not taken into account [2].

```
var x;
x.f = prompt();
x.f = " ";
document.write(x.f);
```

Figure 7: Flow insensitivity example

In **Figure 7**, like the previous example, function *document.write* is called with a completely harmful parameter, that being *x.f*. Because of flow insensitivity not tracking the correct order of updates and the fact that at the beginning of the code snippet the property *f* of object *x* was assigned untrusted (user) input, the analysis produces a false warning of a potential security breach.

- **Path insensitivity:** The analysis does not take into consideration the path conditions by neglecting the code based on the result of conditional branches. Instead, it assumes that all the paths in the *Control-Flow Graph (CFG)* are feasible.

  Such an example can be seen in **Figure 8**:

```
var x = "";
var y = 5;

if (y === 5) {
        x = prompt();
} else {
        document.write(x);
}
```

Figure 8: Path Insensitivity Example

In the above *if-then-else* statement depending on the value of variable *y*, variable *x* is either assigned as user input or as a parameter to *document.write.* That means that there is no way variable *x* can be given user input and also act as a parameter for the sink function.

In a path insensitivity analysis, the conditions do not matter so the analysis assumes that function *document.write* is a sink even though only one path from the branch can be traversed at a time depending on variable *y*'s value.

2.1 General Frameworks

Following, we are going to describe the T.J. Watson Libraries for Analysis, a powerful framework that provides the foundation for the static analysis of the static and hybrid taint analysis tools described in the thesis. Its key features and the fact that is publicly available as an open-source framework have made it prevalent on the sector of static analysis.

2.1.1 T. J. Watson Libraries for Analysis – WALA

The *WALA framework* [3] [24] is a static analysis tool developed by IBM. Originally, its front-end was solely designed to analyze Java bytecode, but it expanded for JavaScript as well.

It is an open-source framework under Eclipse Public License available on Github[1] and it is characterized as one of the best tool for static analysis. It was donated to the community by IBM in 2006.

WALA provides a set of libraries for the sake of *interprocedural data-flow analysis*. In order to do so, its operation is comprised by three steps:

1. Construct a class hierarchy, by reading the source code into memory and keep valuable information describing the types. This produces the analysis scope, meaning what is going to be analyzed. This step is used for the analysis of a Java program.
2. *On-the-fly call graph construction* for the execution of a *pointer analysis* using the appropriate data structures (call graph nodes – *`CGNode`* objects) the framework provides. The nodes of the call graph are representing the corresponding methods and its edges are representing the target procedures to be called. In case of a pointer analysis, the tool automatically constructs a call graph before performing a customizable *flow-insensitive Andersen-style pointer analysis*.
3. Performing the subject analysis over the call graph deriving from **Step 2**.

The analysis performed by WALA can be highly customized by the user by modifying the *CallGraph API (`CGNode objects`)* and *PointerAnalysis API (`PointerKey`)*, which are mentioned in the WALA documentation. We feel that describing the way those APIs exactly work does not correspond with the purpose of this subject thesis and thus we do not analyze them.

For the analysis mentioned above, WALA utilizes the open-source JavaScript engine Rhino[2], which is managed by the Mozilla Foundation[3], in order to parse JavaScript and create Abstract Syntax Trees for the analysis. Likewise, for the sake of fetching HTML code it adopts the Jericho HTML parser[4].

For the IR, WALA's structure is composed by three key types of representations:

- TAC
- CFG
- A fully-pruned SSA form.

Following, we are going to describe the way WALA handles JavaScript's complex features to produce its Intermediate Representation without errors so it can successfully construct the static representation of the program. The taint analyses tools which rely on WALA for their static analysis extend some of its approaches. For example, a framework may change the way WALA is treating the JavaScript prototypes using the customization it provides in its documentation. Another framework may leave the default manipulation of prototypes.

2.1.1.1 Prototype Chain

JavaScript is a prototype-based language [20]. All objects in JavaScript are instances of *`Object`*, except for the primitives. That means all the objects in JavaScript inherit the properties and methods from *`Object.prototype`*.

[1] https://github.com/wala/WALA [Accessed: 4 July 2019]
[2] https://github.com/mozilla/rhino [Accessed: 4 July 2019]
[3] https://foundation.mozilla.org/en/ [Accessed: 4 July 2019]
[4] http://jericho.htmlparser.net/docs/index.html [Accessed: 4 July 2019]

For example, in **Figure 9**, *function x* has a property called *prototype* that may include its properties and methods. In case an object is created out of that *function x*, it inherits the methods and properties of the function.

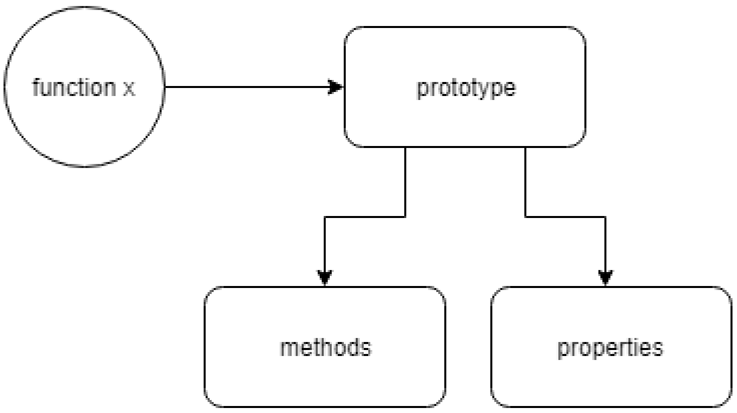


Figure 9: Function's prototype example

Every prototype object may have another prototype object as its prototype. The mechanism of the language allowing an object to traverse from its private prototype object to the object the private prototype has as its prototype until it reaches *null*, in order to collect properties, it is missing, is called *prototype chaining*. An example of the prototype chain process is presented in **Figure 10**.

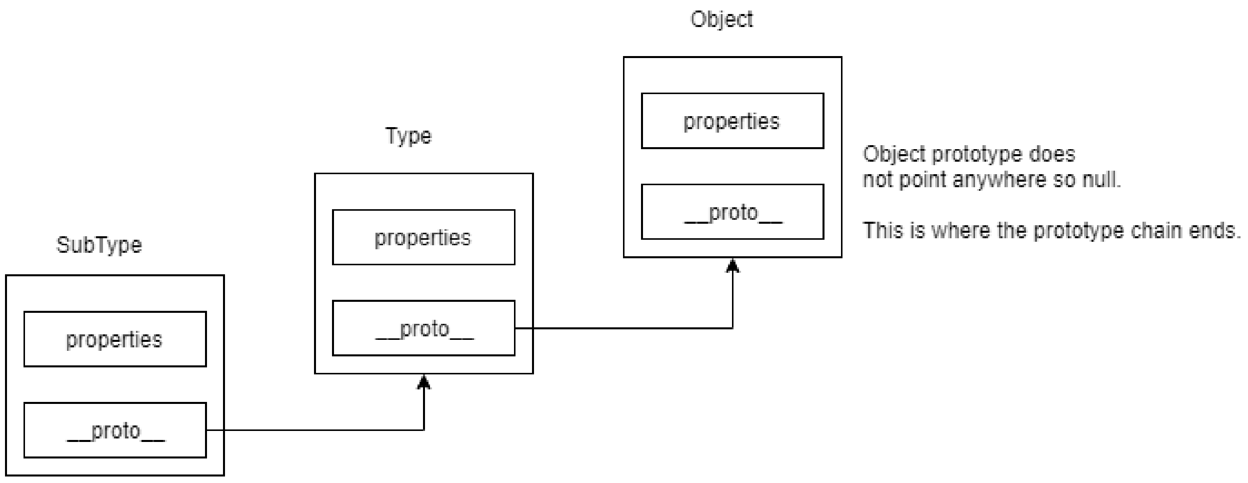

Object
properties
__proto__
Object prototype does not point anywhere so null.
This is where the prototype chain ends.
Type
properties
__proto__
SubType
properties
__proto__

Figure 10: Prototype Chain Process

2.1.1.2 Object Creation

Manipulating the *new* operator in JavaScript is different than in the OOP Languages. The semantics of the *new* operator are dynamic and flexible, since in JavaScript what follows the operator is an expression and **not** a constant. Such a case is seen in **Figure 11** below:

```
var x;

if (cond) {
    x = Object;
} else {
    x = Array;
}

var unknown = new x();
```

Figure 11: New operator issue example

In the example, variable *unknown* can be either an *Object* or an *Array* depending on the conditional *cond* of the *if-then-else* statement.

WALA handles the *new* operator as a dynamic dispatch. The expression following the operator is constructed as a first-class function, hence *new* is translated to a special method call on its argument.

2.1.1.3 Lexical Scoping

JavaScript allows variables which are located inside a method to be accessed and assigned a different value from methods which are inside its body. It has two interesting properties regarding scopes, *Hoisting* and *Closure*.

- *Hoisting* [21]: A variable can be declared after it is used. Hoisting moves all the declarations of the variables to the top of the current scope, which can be a function or a script.
- *Closure* [22]: A feature where an inner function has access to the variables of an outer function.

The code snippet in **Figure 12** refers to a working example where *function bar* can successfully access variable *x* which is in the scope of *function foo* and perform instructions without getting an error.

```
function foo(p) {
    var x = p;
    function bar() {
        var y = x + 2;
        x = y + 1;
    }
}

var z = foo(2);
```

Figure 12: Example of Lexical Scoping

In order to retain this mechanism, the analysis performed is flow insensitive; meaning the instruction sequence in the program is not taken into account.

2.1.1.4 Arguments Array

The *arguments array* [23] is an *Array-like* object that can be accessed within a JavaScript function in order to refer to the value of a parameter given. It is not an *Array* object because it is lacking basic *Array* methods. There are two cases regarding the arguments array, depending on the arbitrary number of parameters the function is called.

The first case refers to a function being called with less number of parameters than its arguments. The rest of the parameters get the value *undefined*, like in **Figure 13**:

```
const foo = function (x, y) {   console:
  console.log(arguments[0]);    1
  console.log(arguments[1]);    undefined
}

foo(1);
```

Figure 13: Less parameters given than declared

In the opposite case, portrayed in **Figure 14**, when the function is called with more arguments than its parameters, the parameters can be accessed through its *arguments.array*.

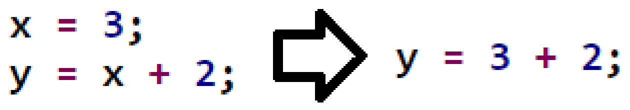
```
const foo = function (x, y) {   console:
  console.log(arguments[0]);    1
  console.log(arguments[1]);    2
  console.log(arguments[2]);    3
}

foo(1,2,3);
```

Figure 14: More parameters given than declared

WALA models the arguments array as an actual array. It creates one with length equal to the parameters given. Accessing an element is a regular array access.

2.1.1.5 Copy Propagation

Copy propagation is an optimization technique used in compilers. It replaces the targets of direct assignments to their values, because such instructions do not have an actual meaning or they may have effect on others. For example, in the following **Figure 15**, the variable *x* is replaced by its value, due to the fact that it does not form a meaningful instruction.



```
x = 3;
y = x + 2;
```

```
y = 3 + 2;
```

Figure 15: Copy Propagation example

This is achieved by constructing the IR with the use of SSA so the analysis is simplified.

2.2 ACTARUS

The IBM's *ACTARUS framework* [6] is a static taint analysis commercial tool based on IBM's WALA static analysis framework. The algorithm used in ACTARUS is available in

IBM's Rational AppScan Source pack[5]. IBM also provides an open-source security test suite[6] that ACTARUS passes with flying colors.

2.2.1 Approach

ACTARUS is relied on an on-demand driven taint analysis approach. It's process consists of two phases:

- Building the static representation of the program using WALA. This stage is composed by the construction of a call graph and the execution of a pointer analysis on the **whole** program, using the *Andersen analysis* library in WALA. It is essential to find the relationship between the variables and the procedures. The fact that object properties in JavaScript can act as taint sources makes a complete call graph representation and point-to analysis essential for the analysis.
- Completing the taint analysis process, by searching for taint sources in the program which may become entry points for the injection of malicious code and tracking the flow of untrusted input data entering from such points. This is done by an interprocedural data-flow analysis. The framework traverses the call graph starting from taint sources and reaching the sinks.

  In more detail, the taint analysis process is based on an expansion of the *Reps-Horwitz-Sagiv (RHS) algorithm* [14], which transforms a data-flow problem in to a graph-reachability problem solved in polynomial time, using the notion of *access paths*.

Building the static representation of a JavaScript program can run into multiple problems and become a difficult procedure. ACTARUS extends some of WALA's techniques eliminating the problems that arise by JavaScript's dynamic features. ACTARUS does **not** model the method `eval`. The remaining of this section presents a thorough description of these techniques.

2.2.1.1 Prototype Chain

ACTARUS models this process in its Intermediate Representation (IR) by rewriting the property accesses into a loop, as seen at the **Figure 16** below. In this specific case the property access is *x* = `y.a`.

```
var p = y;

do {
   x = p.a;
   p = p.__proto__;
} while (!defined(x))
```

Figure 16: ACTARUS property access loop

It can be deducted from the example, that in order for variable *x* to access the property *a* of variable *y*, it checks for variable *y*'s prototypes and goes higher up the chain, until it cannot find any more (reaches `null` pointer).

This is only for property accesses. If a value is assigned to a property that does not exist, a new property is created and there is no need to perform property-chain lookup.

[5] https://www.ibm.com/us-en/marketplace/ibm-appscan-source/details [Accessed: 4 July 2019]
[6] https://researcher.watson.ibm.com/researcher/view_page.php?id=1598 [Accessed: 4 July 2019]

2.2.1.2 Object Creations

The method followed in the framework is similar to the solution WALA uses. However, WALA's modeling of the *new* operator introduces a problem concerning the dynamic nature of the language. For example, depending on the number of arguments given and their type when allocating an *Object*, there can be several cases. Giving the parameter *true* can result in creating a *Boolean* object, giving the parameter *null* can result in creating an empty *Object*.

For this reason, ACTARUS is relied on a custom dispatch, which given for example the *new* operator, it directs these *new* expressions to the appropriate methods, in order to implement the correct semantics based on the parameters given.

2.2.1.3 Lexical Scoping

ACTARUS represents the variables of a program in Static Single Assignment, which means that every variable in the IR is assigned only once. By converting the code on the left of **Figure 17** to its corresponding IR all the variables are replaced by their identifier concatenated with the number of times they have appeared in the program. For example, in line 2 variable *x* is encountered for the first time, so its representation in SSA is *x_1*.

```
1. function foo(p) {
2.    var x = p;
3.    function bar() {
4.         var y = x + 2;
5.         x = y + 1;
6.    }
7. }
8.
9. var z = foo(2);
```

```
1. function foo(p) {
2.  var x_1 = p;
3.  function bar() {
4.        x_2 = LexicalRead(x, foo);
5.        var y_1 = x_2 + 2;
6.        LexicalWrite(x, foo, y_2 + 1);
7.  }
8. }
9.
10.     var z_1 = foo(2);
```

Figure 17: Lexical Scoping SSA form example

In order to preserve the SSA form for the lexical scoped variables, there are two methods used, *LexicalRead* and *LexicalWrite*. Whenever there is a need to access variables from functions inside or outside the local scope of the function, depending on the type of the instruction, if that is a *read* or a *write*, the methods *LexicalWrite* and *LexicalRead* are invoked respectively.

For example, in line 4 of the left part of the **Figure 17**, there is the instruction *var y=x+2*. This instruction is inside *function bar* which is inside *function foo* and is trying to access variable *x* that is declared outside of its local scope. Because of that, in the right part of the **Figure 17** there is a *LexicalRead* statement that precedes the assignment of variable *y*.

Subsequently, the instruction in line 5 at the left part of **Figure 17** is replaced by a *LexicalWrite* invocation.

2.2.2 Taint Analysis

Given a graph *G,* the taint analysis that ACTARUS performs is separated in two phases:

- Traversing G, so the tool can find the sources, sinks and sanitizers. Sanitizers are methods that transform the input data to harmless output data which are ready to be executed by sinks without resulting in any possible harm for the program.

- An interprocedural data-flow analysis that begins from the sources, in order to verify if the data which begin their flow from the sources end up in sinks, without traversing first from a sanitizer.

ACTARUS, instead of explicitly modeling the entire heap which can become really expensive in terms of memory and time, only tracks information relevant for taint propagation. That is tracking local variables and field dereferences which may lead to untrusted data.

$VarId$	Program Variables	v	Values = objects ∪ {null}
$FldId$	Field Identifiers	ρ	Environment = VarId → Val
L	Set of objects	h	Heap = objects x FieldId → Val
		σ	States = 2^{objects} x Environment x Heap

Figure 18: ACTARUS semantics

That is done by performing an interprocedural data-flow analysis relying upon the idea of *access path*, to evaluate the expression in a program state as described in **Figure 18**. An access path $< v, < f1, \dots, fn \gg$ is a pair of a local variable v of the set ρ mapped to its field identifiers $f1, \dots, fn$ from the set h of the program, denoting a heap location. In case a variable v has no identifiers, the access path becomes $< v, \varepsilon >$, *where* ε denotes the empty sequence.

Access paths evaluate to the object o, or $\perp$ if there are intermediate null dereferences or there is no object o in the path. This is done by dereferencing the field identifiers $f1, \dots, fn$ of the object pointed by v. With that way the set of all access paths evaluate to object o and so the flows through the heap can be handled. This is useful for the taint analysis because access paths help it track the taint propagation during the analysis by examining the set of paths that evaluate to untrusted values.

However, due to the unbounded storeless representation of the heap dealing with recursive data structures or heap cycles can lead to access paths being way too long. If their size is not bounded the framework cannot assure termination or it may be time consuming, spending valuable time and resources.

Because of this issue, there is a bound k indicating the length of the tracked access paths for the static analysis. If an access path's length is greater than k it is then approximated/widened by replacing its suffix, beyond the first k field identifiers, by a special symbol, '*'. In practice, the setting $k = 5$ seems to produce the best results.

The on-demand driven analysis performed by ACTARUS lies in the initialization of access paths. They are instantiated only when they are associated with a taint instruction or taint in general. When ACTARUS exposes a taint source, it automatically marks the access paths which refer to this instruction and are associated with the given taint source.

The extra mechanism that ACTARUS has planted in the standard RHS algorithm is related with the alias relations in the heap. Below we will describe two taint analysis examples. The first one presents a simple case. The second one points out a corner case standard RHS fails to detect, while the RHS enhanced ACTARUS successfully points it out.

2.2.2.1 Example 1

```
1. function swapf(y, z) {
2.    z.f = y;
3. }
4.
5. var a = prompt();
6. var b = { };
7. swapf(a, b);
8.
9. console.log(b.f);
```

Figure 19: RHS algorithm example

In this code snippet, there is not a reference-type assignment that can lead to a wrong estimation of the vulnerabilities in the program.

In line 5, variable p is assigned user input through the sink method *prompt*. Immediately, someone can deduct that variable a is a taint source, because adding malicious code to the user input causes an XSS attack. That is how the framework produces the tainted access path $< a, \varepsilon >$, since object a has no field properties.

Continuing the process, the values of the variables a and b become parameters for the *method set*. The relational summary is easily deducted to be $< y, \varepsilon > \rightarrow < z, f >$, since the value of variable y is inserted in the property f of the variable z. That means that it just propagates taint from one parameter to the property of the other parameter.

Returning from the function *set* it is concluded that the path $< b, f >$ is also tainted along with $< a, \varepsilon >$, which leads to an exact result. This can be seen from running the code, since the user input appears in the console.

2.2.2.2 Example 2

Now we are going to look into a different example that the traditional RHS algorithm would miss. ACTARUS's RHS enhanced algorithm, does not though.

```
1. function set(y, z) {
2.     var x = z.c;
3.     x.f = y;
4. }
5.
6. var a = prompt();
7. var b = { c: {} };
8. setf(a, b);
9.
10.    console.log(b.c.f);
```

Figure 20: Enhanced RHS algorithm example

The example is similar to the one distributed before except for one part. In the *setf method* there is a variable x. Variable x is assigned a field identifier of an object, which is a reference type object and not a primitive. That means that it stores the memory location of the property c of object z. As a result at the next instruction in line 3, where in the property f of object x is assigned the value of variable y, a new property f for object b is created holding the user input.

Returning from the $function\ \ setf$ the relational summary deducts that the access paths $< a, \varepsilon >$ and $< b, c.f >$ are tainted. The hypothesis is correct since in the console appears what the user wrote.

The problem that led to the enhanced RHS algorithm is the fact that when a reference type value, an object, is copied to another variable, what gets copied is the memory address and not the value itself. Objects are copied by reference. So when another property is created by a variable pointing there, even by a local method variable as shown below, the property does not get deleted. Thus, whenever a taint flows into an access path, ACTARUS calls a function responsible for tracking alias relations and then proceeds with the basic RHS algorithm.

It is worth noting that when a property of an object is tainted, ACTARUS does not mark the whole object as tainted but only the actual part that is.

2.3 ANDROMEDA

The *ANDROMEDA framework* [7], like ACTARUS, is based upon the IBM's open-source WALA tool and its basic approach is tracking sensitive information, using a demand-driver analysis, without building a representation for the whole program. It calculates the propagation of data flow on-demand. Along with ACTARUS, ANDROMEDA is available as part of the IBM's Security AppScan Source pack[7]. It is able analyze applications written in Java, .NET and JavaScript, in contrast to ACTARUS's JavaScript.

A thorough comparison between ANDROMEDA and ACTARUS is presented at the conclusion of this thesis, in **Section 6.6**.

2.3.1 Approach

The process starts by building a call graph representation of the whole program based on an *intraprocedural type inference*. Also, when there is a need to estimate an alias relationship, which is caused by vulnerable data flow in the heap, the tool performs an on-demand alias resolution. With that way there is no need for a complete pointer analysis.

This has led to:

- A proper, rapid and efficient analysis of large code applications, since only a fraction of them needs to be examined.
- The prospect of *incremental analysis*. Incremental analysis allows the efficient reexamination of the application after a part of its code has changed. This can happen because of a property of the framework to track vulnerable data flows locally or on-demand, separating the parts of the program that have taken part in the analysis from the newly added code. To conclude, when a part of the code changes, the framework does need to redo its data-flow analysis from the beginning, but in the exact specific parts that changed.

In ACTARUS, the framework tracks the sources, downgraders and sinks through the graph using the RHS enhanced algorithm. By using the path relations it deducts the taint propagation between the variables. ANDROMEDA absents from a complete

[7] https://www.ibm.com/us-en/marketplace/ibm-appscan-source/details [Accessed: 4 July 2019]

representation of the program. Instead, it uses lazy data structures for its analysis. There lies their key distinction.

2.3.2 Taint Analysis

ANDROMEDA uses three data structures throughout its analysis. These are a call graph, a type hierarchy and data propagation graph. Instead of building the representation for the whole program, ANDROMEDA employs greedy/lazy methods to construct these data structures so it can perform the analysis.

The analysis starts with the assignment of the web application as input to the algorithm. Then, the framework computes the type hierarchy of the application, by using techniques for caching and demand evaluation. For the construction of the call-graph it utilizes an oracle which deducts if the call graph needs to expand or not, based on the calling methods and the taint source methods, capturing only the data it regards as valuable. This oracle is based on control-flow reachability within the type-hierarchy graph.

ANDROMEDA's taint analysis flow representation is similar to the one ACTARUS uses, that being using the notion of *access paths*. It takes as input a web application, along with the libraries used, and it performs a data-flow analysis based on a set of security rules. The access paths need to abide by them.

The security rules used in ANDROMEDA consist of a triplet $< Src, Dwn, Snk >$, where Src, Dwn and Snk are the sources, downgraders and sinks respectively in the program to be analyzed. A vulnerability is reported when in a security rule data flows from a source to a sink without having a downgrader between them. This is where the tool focuses and thus the turn on-demand driven analysis.

$VarId$	Program Variables	v	Values = objects $\cup$ {null}
$FldId$	Field Identifiers	ρ	Environment = VarId $\rightarrow$ Val
L	Set of objects	h	Heap = objects x FieldId $\rightarrow$ Val

Figure 21: ANDROMEDA's concrete semantics

A state of the program $\sigma = < E, H > \epsilon\ States = Env\ x\ Heap$ points from variables to their values and from object fields to their values respectively. In order to track security facts, there is an instrumentation of the above concrete semantics to maintain access paths. An *access path* is a symbolic representation of a heap location represented as a sequence $x.f1 \ldots fn$ of field identifiers ($FldId$) rooted at a local variable ($VarId$), meaning an element in $VarId\ x\ (FldId)*$. For example, access path $x.g$ denotes the heap location of property g of the object *x.*

The access paths are needed in order to extend the state of the program to an instrumented concrete state which is now a triple, $\sigma = < E, H, T >$ where T is a set of *tainted access paths*. The semantic rules for updating T in case of an assignment or field-read statement are similar to ACTARUS. Whenever a tainted value is assigned to another variable, the later one is marked as tainted. In case of object properties, only the property is regarded as tainted and not the object with all its property fields.

Using access paths can lead to the same issues ACTARUS had regarding the storeless representation of the heap. ANDROMEDA solves these issues as ACTARUS, by using a technique called *access path widening*. We have already described that technique in **Section 2.2.2**.

2.3.2.1 On-Demand Aliasing

The analysis is performing an on-demand alias analysis whenever untrusted data flows into a property of an object. It is a lot similar to the issue ACTARUS describes regarding the RHS algorithm, which would not take into account reference type object assignments like the one explained in **Example 2**.

On-demand aliasing occurs whenever there is a corner case. In particular, whenever there is an instruction assigning a variable to a property of an object, ANDROMEDA traverses the control flow graph backwards searching for aliases of the object. A simple example of this feature will be explained in the section below.

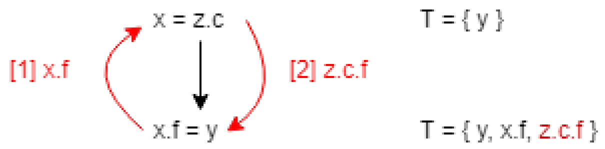


Figure 22: On-demand aliasing example

In the above example, set T has an already tainted access path. Variable x is assigned a property of another object that means it is pointing to the memory address of the property c of object z.

Reaching the second instruction triggers the on-demand analysis. Its steps are the ones written on red font. Because of this assignment and because of object x pointing to the memory address of $z.c$, the property f of the property c of object z has become tainted. ANDROMEDA successfully returns backwards to the graph to pinpoint any alias relations of x finding the first instruction and deducting that $z.c.f$ is correctly tainted.

2.3.2.2 Change Impact Analysis (CIA)

The CIA algorithm is the one responsible for the feature of incremental analysis. In more detail, it examines all the layers of the data structures of ANDROMEDA. In case of a code change in a compilation unit, it resolves the differences between the two versions. It does that by exposing all the points of the program that have been affected and modifies the data structures using them appropriately. For example, a code change may result in a modification of the call graph, while the type hierarchy stays intact.

3. DYNAMIC TAINT ANALYSIS TOOLS

Both types of analyses, either that is a static or a dynamic taint analysis, detect vulnerabilities in the subject code. The big difference is where they find problems in the development lifecycle. As mentioned, static taint analysis identifies vulnerabilities before you run a program by examining the source code. On the other hand, dynamic taint analysis adds instrumentation to the program so it can reach conclusions regarding the taint propagation during the execution.

Even though their goal is the same, they tend to find errors in code that the other cannot. There are defects that a dynamic taint analysis can miss, the static one can detect successfully and vice-versa.

This taint analysis category comprises of a sole framework.

3.1 General Frameworks

In the following we are going to describe the two versions of the Jalangi framework and its key aspects. It is used for the basis instrumentation of the dynamic taint analysis tool presented in the ending of the section, due to its unique mechanic which allows the proper dynamic analysis of a program.

3.1.1 Jalangi

The *Jalangi framework* [4] is an open-source dynamic analysis framework for the JavaScript language available in Github[8]. It is a platform-independent framework, meaning that its design is not relied on browsers or JavaScript engines. This design does not require the continuous maintenance of the framework in case of a browser update and it is not tied to a particular engine. Had it been tied to a browser, whenever there was an update it would need to be updated as well. In contrast to this, the analysis can be performed in any machine, desktop or cloud.

The tool is no longer supported but it is still available. We discuss it to explain some key features there are used to its updated version *Jalangi2* and also mention the fact that there is a simple taint checker implemented within. It combines two basic approaches:

- **Selective record-replay:** This mechanism allows the user to *record* a specific part of the program and execute it again (*replay*). This can be really useful to a person performing an analysis since he can thoroughly examine an unexpected behavior.
- **Shadow values:** This mechanism allows the storage of useful information about an actual value in the program. Every value in the program can be associated with a shadow one.

3.1.1.1 Selective record-replay

As the name indicated this technique is divided into two phases, the record and the replay. During the record phase, the whole application is being executed along with the instrumented parts of the program the user has added for his analysis. During the replay phase, only the parts of the program that the user has added instrumented code get replayed. The native functions and the uninstrumented parts do **not**.

This division in two phases allows their distinct execution. One can run the record process on one platform and then the replay process on another platform. For example, someone whose computer is not that powerful and the application they want to process

[8] https://github.com/SRA-SiliconValley/jalangi [Accessed: 4 July 2019]

is demanding, can run the record process through a cloud machine with higher capabilities and then complete the record-replay process with his desktop machine.

As mentioned, the replay phase does not track uninstrumented code or native functions executing. Instrumented functions that get called inside that code pose a demanding problem for the framework. Jalangi records each object and function with a unique numerical identifier. Also, it records the explicit calls to the instrumented functions which are invoked by functions whose code is not taking place in the replay process.

Since the replay phase only executes a fraction of the program, it needs to load the correct values so the user can successfully replay their instrumented code. Jalangi makes use of the shadow memory in order to address this problem. During the record phase, the framework keeps track of the memory loads taking place only inside the instrumented code. If during that process Jalangi finds out that the values it has recorded in the shadow memory for its variables are different than the ones at the end of the process, due to uninstrumented code or native functions in between changing the values of variables, it records them for the next phase.

In order to achieve this, the instrumentation taking place during the replay phase uses the function *sync,* which makes sure that in the end of the record process the actual value of a variable and the value of its shadow variable are not mismatched. With this way, the values loaded for the replay process are sure to be correct.

3.1.1.2 Shadow Memory

The shadow values and shadow execution play a major role in the vector of a dynamic taint analysis. They are the ones responsible for propagating the taint and holding taint information. For example, Jalangi when noticing a taint source it saves the fact that it is vulnerable input and that it can propagate taint in its corresponding shadow value. Hence, whenever it comes into contact with other variables, it makes sure they are marked as tainted too.

As mentioned, the instrumentation of the application is performed during the record phase of the analysis. The same applies to the shadow execution, since it is a form of instrumentation defined by the user.

The core of the shadow execution is associated with the object *AnnotatedValue*. This object has two fields, one for storing the actual value of a variable and another one for storing the shadow value of a variable, meaning extra information for it.

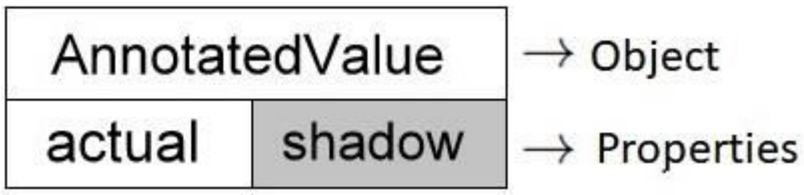


Figure 23: The `AnnotatedValue` object

For example, if during a taint analysis and user input method is encountered, the actual value of the method can be replaced by a user defined annotated value. This can be done by creating a new *AnnotatedValue* object which has two methods, one returning the actual value field of the object and the other one returning the shadow value field of the object.

Jalangi makes sure built-in JavaScript methods are performed on the actual values even if they are replaced by user defined annotated values, by injected the appropriate instrumentation code.

3.1.2 Jalangi2

Jalangi's advanced version is *Jalangi2* [5]. Like its predecessor, it is an open-source dynamic analysis tools that can also be used in a hybrid analysis. Its code can be found on Github[9]. It can handle all the dynamic features of the JavaScript language. The key difference with Jalangi is the removal of the record-replay mechanism.

On the other hand, it still retains the shadow memory mechanism but in a more advanced manner. There is a new shadow memory API that can be found on the documentation of Jalangi2. Expanding further into this API does not match the goals and targets of this thesis. The basic idea is the same as Jalangi with the exception of the way the shadow memory is manipulated by the user.

The structure of Jalangi2 is seen below:

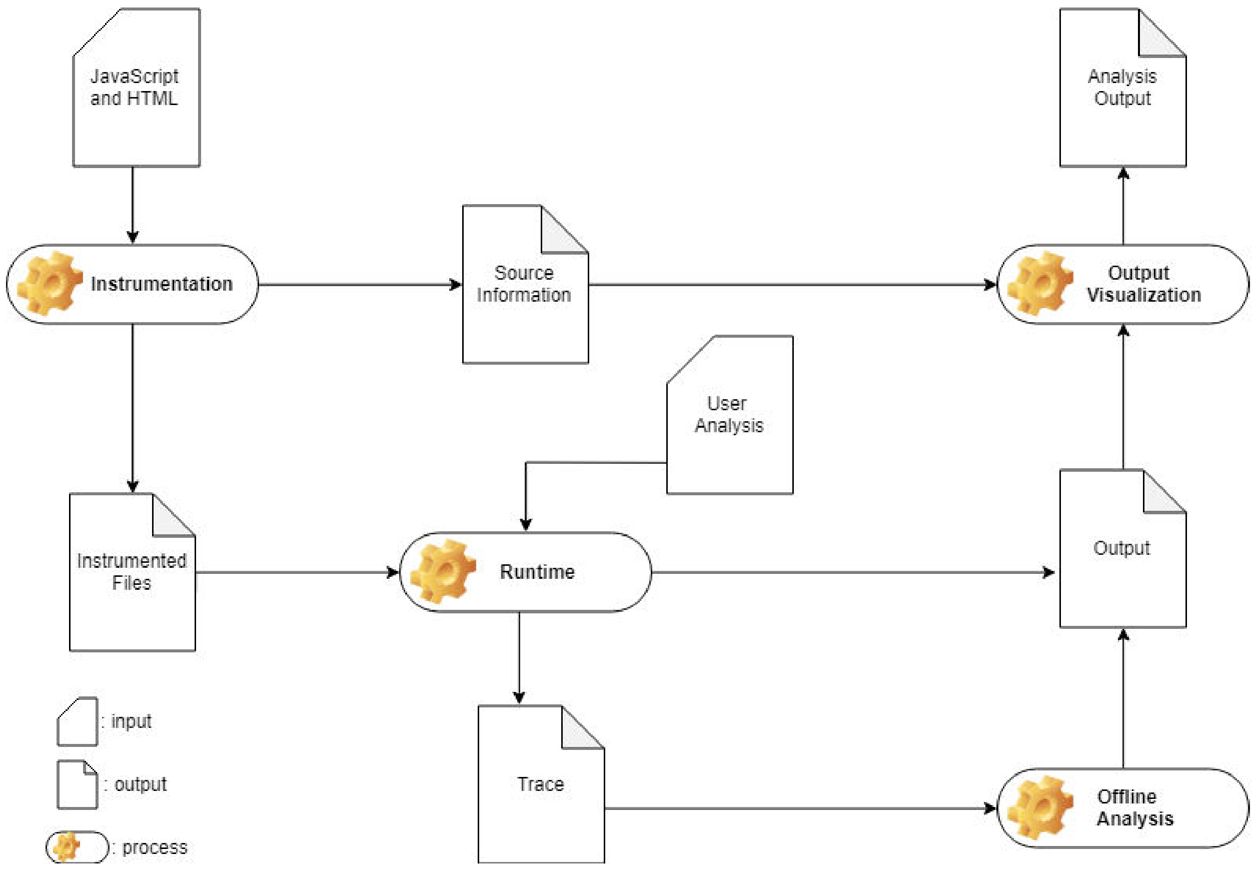


Figure 24: Jalangi2 process

3.1.2.1 Jalangi Instrumentation

Takes as input the HTML and JavaScript code and produces the Source Information with the Instrumented Files.

In more detail, during this process the framework takes the various instructions in the JavaScript program and adds instrumented code so it can process them in the Jalangi Runtime process happening later.

3.1.2.2 Source Information

During the instrumentation, Jalangi2 associates a unique instruction identifier iid to each instruction. Also, during the runtime it associates a unique script id to each script.

[9] https://github.com/Samsung/jalangi2 [Accessed: 4 July 2019]

The user can take advantage of such information while processing his Jalangi Callback functions by using the methods *J$.getGlobalIID(iid)* which returns the unique id of the expression and *J$.iidToLocation(iid)* which returns the location of the specified *iid*.

3.1.2.3 Jalangi Runtime

Takes as input the Instrumented Files and the User written Analysis and produces the Output of the analysis and a Trace which can be used for Offline Analysis.

The User Analysis file notes the Jalangi Callback[10] functions that take part in the analysis. The implementation of an analysis requires the implementation of several callback functions. Specifically, an analysis is declared when there are objects assigned to *J$.analysis* object, where each object defines the instrumentation code for a callback function. Whenever a condition or instruction in the program is executed the corresponding property of the analysis object is called. The user can define these properties as they prefer to deduct conclusions concerning his analysis. An analysis is always terminated by calling the *endExecution* callback function.

Below we are going to present a simple analysis that shows how many times a for-loop was executed:

```
for (var i=0; i<5; i++) {}
console.log("Done doing absolutely nothing.");
```

Figure 25: Dummy program

In **Figure 25** there is a dummy *for-Loop* that does absolutely nothing. After its execution there is a message logged in the console. The analysis is customized by the implementation of the *conditional* callback function:

[10] Jalangi Callback functions can be found in the jalangi2/docs/MyAnalysis.html in the Github page

```
(function(){
var id;
J$.analysis = {

     conditional : function (iid, result) {
          id = J$.getGlobalIID(iid);

          if (result) {
               console.log("true");
          } else {
               console.log("false");
          }
     },

     endExecution : function () {
          var location = J$.iidToLocation(id);
          console.log("The boolean values were printed by the "
               + "conditional callback function invoked at "
               + "location: " + location + ".");
     }
};
}());
```

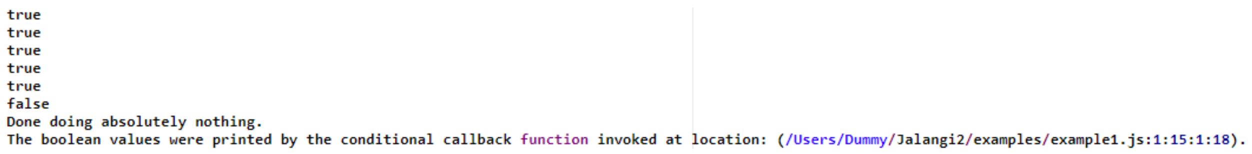
Figure 26: Jalangi analysis example

The analysis starts by assigning two properties to the *J$.analysis* object. Each property is represented by a callback function.

The *conditional* takes as arguments the instruction identifier of the conditional examined. It records the id of the conditional and prints to the console the boolean values *true* or *false* based on the result evaluation of the condition.

The *endExecution* does not take any arguments. It is called at the ending of the analysis in order to print the location of the conditional that took place. Normally, it is used to conclude some results about the analysis.

The output printed when executing the above analysis is the following:



```
true
true
true
true
true
false
Done doing absolutely nothing.
The boolean values were printed by the conditional callback function invoked at location: (/Users/Dummy/Jalangi2/examples/example1.js:1:15:1:18).
```

Figure 27: Jalangi2 analysis example output

3.1.2.4 Offline Analysis

A useful mechanism in the framework that allows the users to analyze a previously captured *Trace* without requiring the analysis to start from the beginning and performing it offline.

3.1.2.5 Output Visualization

Takes the Source Information file and associates the output file with the exact locations in the program to produce a more precise analysis deduction.

3.2 Ichnaea

Ichnaea [8] is a platform-independent dynamic taint analysis tool based on the Jalangi2 instrumentation framework. It handles the ECMAScript 5 language. Its platform-independent feature means there is no need to modify the JavaScript interpreter in order to keep track of the information flow analysis, just like *Jalangi* and *Jalangi2*. It can be used with any existing JavaScript engine. It was designed in order for developers to have in their possession a tool that they can use during the development of an application.

As with the other tools described, the Ichnaea framework is not an open-source tool. It incarnates the approach presented on the remaining section.

3.2.1 Approach

The JavaScript source code to be analyzed is given as input to Ichnaea along with a configuration file called *taint specification*. This configuration file specifies the taint sources, tainted sinks and extra configurations.

Specifically, the configuration file may specify the taint sources and sinks through a configuration parameter called *taintSpec*, which has two properties called *sink* and *source* that are arrays of properties to values. If someone for example wants to assign the method *eval* as a sink they will need to assign into the *sink* array of properties a property called *name* with the corresponding value *eval*.

```
Spec.taintSpec = {
   "source" : [ {"name" : "prompt"} ],
   "sink"   : [ {"name" : "eval"} ]
};
```

Figure 28: Ichnaea taint specification example

In addition to this, there can be other configuration parameters that enhance the analysis and provide useful features, such as:

- *taintNodeCommandLineInput*. If set to true all the command line arguments given to the program are assumed they are tainted.
- *taintAllUserDefinedString*. If set to true any user defined string will be treated as tainted data.
- *reportFlowLocation*. If set to true the analysis will be able to report the location of the first tainted flow found.
- *reportAllFlows.* If set to true the analysis will be able to report the location of all the flows found.

After specifying the configuration file for the taint sources and sinks and with the help of Jalangi's instrumentation, Ichnaea generates a sequence of instructions for an abstract stack machine. The abstract machine is implemented as a Domain Specific Language (DSL), meaning it is developed to meet the needs of the particular framework.

The abstract machine includes operations among abstract values. Abstract values are strings representing locations, where each location is represented by the file's name and the line's number. Every abstract value is mapped to a local variable or object property.

The abstract machine defines a set of operations described below:

- *push( )*: pushes *true* or *false* to the stack indicating if the value is tainted or not. True means it is tainted, false it is not.

- *pop( )*: pops off previously pushed taint of an expression and discards its value.
- *unaryop(op)*: pops off the stack, applies the unary operator *op* for the evaluation of an expression and then pushes the result back into the stack.
- *binaryop(op)*: pops off the two top elements of the stack, applies an operator *op* for the evaluation of the expression and then pushes the result back into the stack.
- *initvar(var)*: pops the stack and creates a new map entry for the local variable *var* according to the tainted value popped.
- *readvar(var)*: loads taint for the variable *var* and push it into the stack.
- *writevar(var)*: stores taint value on top of the stack into the variable *var*.
- *setvar(var, taint)*: stores tainted value *taint* into local variable *var*.
- *initproperty(obj, prop)*: pops the stack and initializes the property *prop* of an object *obj* according to the tainted value popped.
- *readproperty(obj, prop)*: loads taint for the property *pr*op of an object *obj* and pushes it into the stack.
- *writeproperty(obj, prop)*: stores taint value on top of the stack into the property *prop* of an object *obj*.

Instructions *writevar* and *writeproperty* are always followed by a *pop* method, since they do not pop a taint value from the top of the stack. Literals, functions and objects are never tainted. Hence, the generation of abstract stack machine instructions equal to a boolean *false* push in the stack.

The instructions we described above provide a solid structure for the algorithm. Nonetheless, they do not cover all of JavaScript's dynamic features, such for example *eval.* Let us see how such features are handled:

- *Arrays*: The abstract stack machine handles arrays and objects as they were completely the same type.
- *Getters and setters*: These read and writes are not handled as property read and writes as they are used. Instead, they are modeled as a function call.
- *Apply and Call*: These functions are used to assign the *this* pointer to a function. Their difference is that *apply* gets an array of arguments, whereas *call* gets the arguments separated by comma. These functions are represented as a call to the function they are used.
- *eval*: *eval* operations are regarded as sinks during the analysis. However, because of the taint specification extra configuration argument *reportAllFlows* described before; there is a need to examine the code inside an *eval*. The code *eval* needs to evaluate is treated as an additional script in the program and it is instrumented as if it was not inside it.
- *Exceptions*: They are modeled with the declaration of the special variable *_throw_*. In more detail, when there is need to use a *throw* statement, the framework models it as *writevar('_throw_').* Respectively, a *catch* statement is modeled by a *readvar('_throw_')* in order to process the *throw* statement.
- *Arguments*: Each access to this array object, which is available to all methods, is handled as an object access.

- *Arguments in function calls*: Calling a JavaScript function with less parameters than the arguments declared results in the missing parameters assigned the *undefined* value. In the analysis there are pushed false taint values in the stack.
- *Native functions*: Native functions in JavaScript take callback functions as parameters. For that reason, in the taint analysis each native function is associated with its callback.

Executing these instructions generated with the abstract machine produces a report for the taint flows of the application. The execution is performed by a JavaScript engine or browser. A portrayal of the process taking place in Ichnaea can be seen in the **Figure 29**.

What makes Ichnaea platform-independent is its ability to execute the generated abstract machine instructions in another platform than the one the application was developed, since it does not need a specially customized JavaScript engine or instrumented browser.

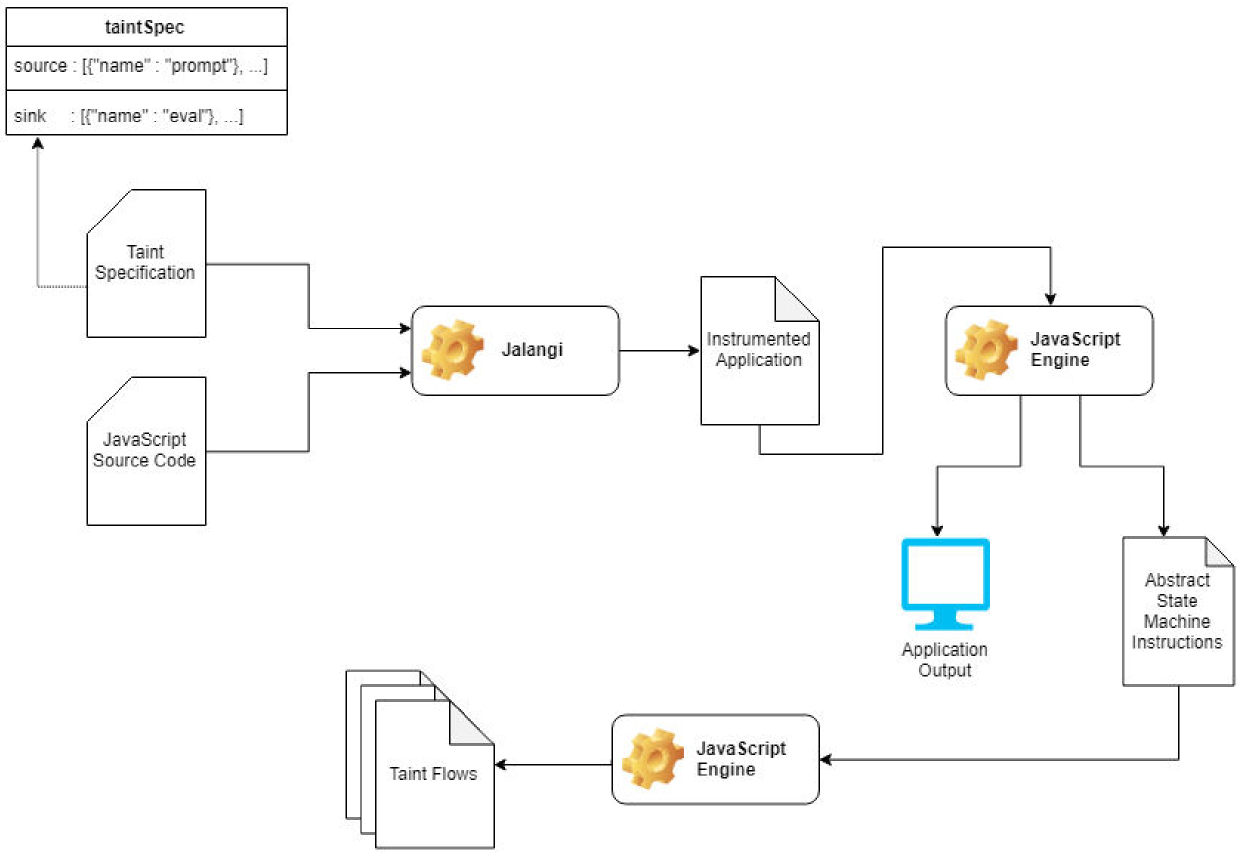


Figure 29: Ichnaea's Structure

3.2.2 Taint Analysis

The instructions for the abstract stack machine are issued with the help of Jalangi's shadow memory feature, enabling the association between object identifiers with objects and arrays. In the current implementation Jalangi2 is used.

To describe the way the instructions for the abstract machine are generated and how the framework understands a variable from a taint source reaches a sink we are going to examine an example.

```
var child_process = require('child_process');
var array = ['ping'];
array[1] = process.argv[2];
var command = array[0] + array[1];
child_process.exec( command,
                   function (err, stdout, stderr)
                   {
                        console.log('stdout: ' + stdout);
                        console.log('stderr: ' + stderr);
                        if (error !== null) {
                             console.log('exec error: ' + error);
                        }
                   });
```

Figure 30: Ichnaea sample program

In the above example, a user gives as input an Internet Protocol (IP) address to a platform and then platform informs him if the host of the IP he gave is alive or not. This could be used to check if a website is available or not. Someone can observe that the above code snippet is vulnerable to a code injection, since it accepts user input without validation. If a user gave as input the following statement *'127.0.0.1 && sudo rm –rf /'* they would get a message responding to his ping request and succeed in the deletion of the whole server's files.

Bear in mind that literals, functions and objects are never considered taint, after every *writevar* or *writeproperty* follows a *pop* and the instructions for the abstract machine are issued from the right to the left, meaning if there is an assignment, the instructions generated are first for its right part and then for the its left part. Also, there is a taint specification file setting the sink to be the command *exec*.

The abstract machine instructions after the code instrumentation with the help of Jalangi would be the following:

Line 2:

```
1. push(false);
2. initproperty('obj7', '0');
3. push(false);
4. writevar('frame3:array');
5. pop();
```

Line 3:

```
6. readvar('frame3:array');
7. push(false);
8. readvar('frame5:process');
9. push(false);
10.   readproperty('obj9', 'argv');
11.   push(false);
12.   readproperty('obj11', '2');
13.   pop();
14.   push('(example.js:3:12)');
15.   writeproperty('obj7', '1');
16.   pop();
```

Line 4:

```
17.   readvar('frame3:array');
18.   push(false);
19.   readproperty('obj7', 0);
20.   push(false);
21.   readproperty('obj7', 1);
22.   binaryop('+');
23.   writevar('frame3:command');
24.   pop();
```

Figure 31: Abstract stack instructions generated for the example

As seen, the line 2 of the example generates five instructions. At first, a taint value for the literal *'ping'* is pushed. Then there is an initialization for the element *0* of the array *obj7* and a push of a taint value for the array literal. Finishing up, the array object initialized is written to the variable array following up by a pop.

The line 3 of the example is where the taint propagates. The element of the array *array[1]* is assigned with the user input argument given. It is handled as a taint source. That can be seen in line 14, where the exact location of the taint source is pushed to the property *1* of the object *obj7* meaning the *array[1]* variable.

In line 4 is where the variable *command* is assigned the string concatenation of the array elements *array[0]* and *array[1]*. Since *array[1]* has been tainted the taint needs to traverse to the command as well. *Binaryop* instruction by popping the stack twice and assigning the result back to it is responsible for that and succeeds, since in line 23 in the variable *command* is assigned the tainted source location which was pushed back in line 14 to the property *1* of the object *array*.

Finally, the instructions reach the sink *exec* which was specified to be one in the taint specification file. The taint report issues the exact location of the tainted flow.

In case in our example line 4 was substituted with the native function reduce the instructions would be the following, since Ichnaea handles the native functions with nested callback functions successfully, given there is only one callback function inside it.

Line 2:

```
1. push(false);
2. initproperty('obj7', '0');
3. push(false);
4. writevar('frame3:array');
5. pop();
```

Line 3:

```
6. readvar('frame3:array');
7. push(false);
8. readvar('frame5:process');
9. push(false);
10.  readproperty('obj9', 'argv');
11.  push(false);
12.  readproperty('obj11', '2');
13.  pop();
14.  push('(example.js:3:12)');
15.  writeproperty('obj7', '1');
16.  pop();
```

Line 5:

```
17.  readvar('frame3:array');
18.  push(false);
19.  readproperty('obj13', 'reduce');
20.  push(false);
21.  push(false);
22.  push(false);
```

from the second argument of reduce to the accumulator

```
23.  pop();
24.  initvar('_accum_');
25.  pop();
```

from the accumulator to the first argument of the callback function

```
26. readvar('_accum_');
27. push(false);
28. push(false);
29. readproperty('obj7', '0');
```

body of the callback function

```
30. push(false);
31. push(false);
32. readproperty('obj17', '0');
33. push(false);
34. push(false);
35. readproperty('obj17', '1');
36. binaryop('+');
37. writevar('_ret_');
38. pop();
```

from the return value of the callback function to the accumunlator

```
39. readvar('_ret_');
40. writevar('_accum_');
41. pop();
```

from the accumulator to the return value of reduce

```
42. readvar('_accum_');
43. writevar('_ret_');
44. pop();
```

Line 4:

```
45. pop();
46. readvar('_ret_');
47. writevar('frame:command');
48. pop();
```

Figure 32: Abstract stack machine instruction using the native function reduce

4. STATIC & DYNAMIC TAINT ANALYSIS TOOLS

As we have presented, both types of tools have the same goal but operate in a different way. Hence, it is a good practice to integrate both analyses to form a hybrid taint analysis framework that uses static analysis for the thorough examination of the program and a dynamic analysis for the manipulation of the dynamic features of JavaScript.

Following, we present a commercial framework and an approach we have found, that cover these thoughts.

4.1 JavaScript Blended Analysis Framework

4.1.1 Approach

The *JavaScript Blended Analysis framework (JSBAF)* [9] was designed in order to combine both types of analyses, static and dynamic. Its static infrastructure was built on the IBM WALA open-source static analysis framework. The reasoning behind this approach is derived from the fact that dynamic analysis can detect flaws in the program that the static analysis misses, like for example a dangerous *eval* operation. The process taking place for this tool is composed by two phases, the dynamic and static ones.

The first one is separated in two sub-phases:

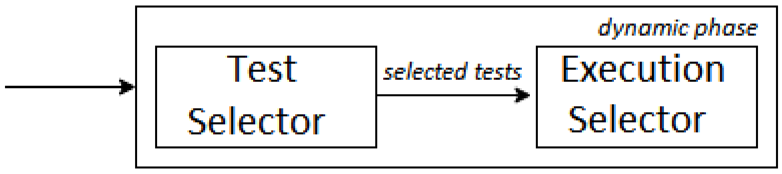


Figure 33: JSBAF's Dynamic Phase.

- **Test Selector:** Chooses a subset from a set of tests that provide good coverage of the program given for processing, to achieve good analysis and in lower cost that using all the set.
- **Execution Selector:** Collects run-time information for every test executed and produces a dynamic trace for each one of them. For example, function calls.

Respectively the second one is separated in the phases described below:

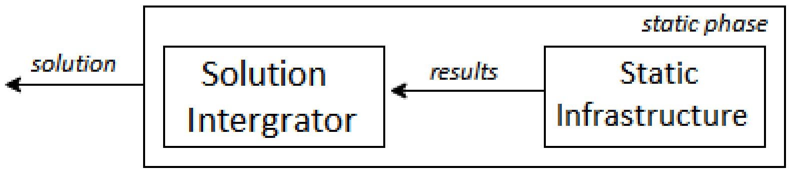


Figure 34: JSBAF's Static Phase.

- **Static Infrastructure:** Analyzes the program.
- **Solution Integrator:** Merges the data-flow solutions from different test traces into a program solution. Then decides if there should be more traces to analyze.

The structure of the framework is portrayed by the **Figure 35** below:

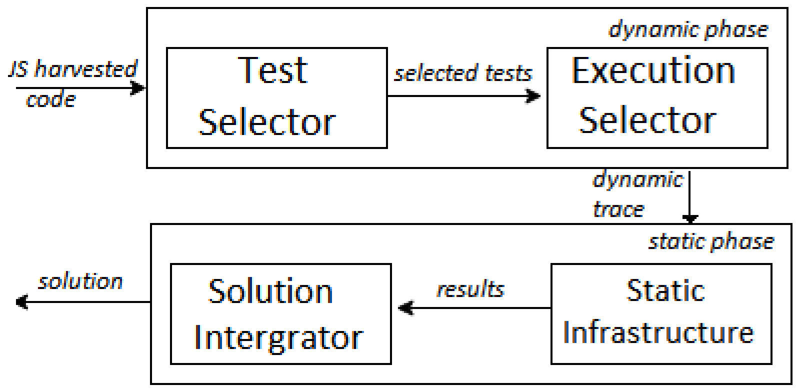


Figure 35: Structure of JSBAF

Regarding its design, JSBAF performs dynamic analysis in each dynamic trace separately and it merges the results. Even though it would be faster to have a single uniform dynamic trace instead of multiple small ones, the designers chose this approach, because of its accuracy.

As with ACTARUS, JSBAF handles some of the dynamic issues of the language that arise during static analysis. The rest, such as object creations, prototype chaining are modeled by default by IBM's WALA framework. There can be solid improvements regarding their accuracy though.

4.1.1.1 Eval

A pure static analysis may miss *eval* expressions or approximate them in the worst case, due to the fact that they produce code at runtime. This dynamic generated code makes static analysis unsafe when analyzing JavaScript programs.

In this specific framework, *eval* calls are being monitored by the Execution Collector during the dynamic phase of the analysis. The Execution Collector gathers them along with any code they produced and transfers them to the Static Infrastructure, which analyzes the program including the aftermath of the *eval* calls.

4.1.1.2 Function Variadicity/Arguments Array

Function Variadicity occurs when a function is called with a number of parameters that differ from its declaration. Static analyses ignore this property of the language, because the number of the parameters in which a function is called cannot be known before runtime. This problem is again assigned to the Execution Collector since it is able to detect the exact number of parameters for every function call.

With this way, instructions that are based on the number of parameters given and don't match with that number are pruned, with the pruning mechanism. Instead, static analysis would assume that the data flow could take any branch resulting in a false result.

JSBAF's pruning mechanism, knowing the arguments given at runtime presents a form of context sensitivity. A static analysis would assume that every branch is feasible.

```
function variadicity_example() {
    var input;

    if(arguments.length === 0) {
        input = null;
    } else {
        input = arguments[0];
    }

        other_function(input);
    }
```

Figure 36: Function variadicity example

Instead as shown in **Figure 36** the variable `input` can take two values depending on the length of the `arguments.array.` So examining the number of arguments and finding the correct one depending on occasion can result in pruning the unnecessary code.

4.1.2 Taint Analysis

Continuing we are going to describe JSBAF's phases in detail and see how they differ from pure static analyses, like ACTARUS.

4.1.2.1 Dynamic Phase

As mentioned, this phase of the framework is consisted of two processes. The *Execution Selector*, which is responsible for extracting the *page traces*, through the Trace Extractor and the *Trace Selector*, which is responsible for choosing the best page traces for analysis.

The Execution collector relies on a specialized version of TracingSafari, an instrumented version of the open-source web browser engine WebKit[11], developed for characterizing behavior of JavaScript programs.

TracingSafari records the operations performed by the JavaScript's interpreter of the Safari web browser including `reads, writes, deletes, field adds`. JSBAF's dynamic phase requires human interaction. That is why the representation should affect the browser performance slightly. TracingSafari collects only the information that is essential for the taint analysis.

To ensure the security of a website, web tester checks all the pages of the same domain. A web application's operation may have code from different pages. The instructions gathered get separated in page traces, where each trace includes a series of instructions from the same page of the website. There is at least one trace for each page.

A page trace consists of a dynamic call tree, recorded object creations, compile-time visible JavaScript source code and dynamically generated code including any executed library code.

[11] https://webkit.org/ [Accessed: 4 July 2019]

The Execution Collector records the exact number of parameters for each function called. The Trace Extractor is the one responsible for constructing the page traces for each web page.

However, in a web application there might be similar pages. That results in similar page traces as well. JSBAF framework avoids checking such cases, since the results taken from examining them compared to the time spent are not satisfactory.

This is where Trace Selector interferes by choosing the traces which differ the most and cover a large extent of the application. The traces chosen satisfy the following requirements by including:

- Dynamically generated code.
- Methods.
- Object creations.

Choosing which page traces prevail over others is assigned to different metrics developed by the framework.

4.1.2.2 Static Phase

The Static Phase is where the page traces are statically analyzed by the WALA framework. JSBAF's *Call Graph Builder* is responsible for building a call graph for every page trace as a WALA data structure with pruned source code for each node. Since WALA cannot detect dynamically generated code, JSBAF has created the *Code Collector* to obtain that code from the page trace. Also, the Call Graph Builder applies pruning to the code of all functions as mentioned in the Function Variadicity handling.

Below we are going to describe the static taint algorithm which detects the program for any integrity violation. It is divided in four steps:

1. A pointer analysis for JavaScript performed in order to obtain aliases of objects in the program. This technique helps establish which pointers or heap references can point to which variables. There are a lot of corner cases that can be missed without examining the heap of the program, like the ones referenced in the examples given in ACTARUS and ANDROMEDA.
2. *Sources* and *sinks* are automatically identified in the program:
    - A data source is called tainted when the user of an untrusted third party has control of its value. That is for example a user. The same for JavaScript functions from an untrusted third party. Taint sources like that can be JavaScript's event handlers.
    - Every method that writes sensitive information is referred as a sink. Variables holding browser/user information are regarded as sensitive.
3. A call-graph reachability analysis is performed to filter out any node that is not a direct call path from a method containing taint sources to a method containing sinks. The remaining nodes are called *candidates*.
4. A performance of an interprocedural traversal of the call graph from each taint source to each sink, from the candidate nodes left out of the call-graph reachability analysis. At each encountered candidate method, an intraprocedural data dependence analysis is applied to track the tainted variables into candidate calls. If one argument of the call is tainted, we assume all the arguments are tainted as an optimization to avoid analysis of these methods. A taint propagation process to conclude the analysis.

4.2 An approach on XSS Prevention

Due to the severity of XSS attacks, this paper [10] proposes a technique that tracks the information flow of sensitive information on the client-side. Monitoring the client-side of an application can eliminate all the types of XSS attacks.

4.2.1 Approach

In more detail, it uses the JavaScript engine of the browser to track the flowing of sensitive information, by using a *dynamic taint analysis*. However, since a dynamic taint analysis may miss taint propagation on some occasions, because of the fact that the attacker may be able to execute code that it does not cover, there are also some fractions of the code that are analyzed by a *static taint analysis*. The approach presented uses in its majority dynamic taint analysis and on-demand static taint analysis.

The description of this taint analysis technique was incorporated into the Mozilla Firefox[12] 1.0 pre web browser. For the verification of the mechanism was used a web crawler based on Firefox and able to traverse the application and perform user actions.

4.2.2 Taint Analysis

As mentioned the taint analysis used in this approach is a hybrid variation combining a dynamic and static taint analysis.

4.2.2.1 Dynamic Taint Analysis

The dynamic taint analysis taking place in the hybrid analysis has the ability to track data-dependencies and control-dependencies to ensure correct taint propagation. That means that when there are assignment instructions where a tainted value is assigned to another value, that value becomes tainted too. The same happens, in case of operations like *switch, if-then-else* when they are based upon a tainted variable. The result is marked as tainted.

A really experience attacker can try to sanitize tainted variables so he overpasses the dynamic taint analysis by using complex structures. For example, by creating a structure that decomposes a string to characters and then merges them back together.

In the example in **Figure 37** the attacker is trying to overcome both the dependencies mentioned. At first, they try to assign parts from a taint source to the properties of an array and then from that array they try to assign the characters of the cookie using a *switch*.

By using data dependency, when the *cookie* string is assigned to the array, since on the right of the assignment is a tainted variable the array becomes tainted too. By using direct control dependencies, when the technique notices that both conditions are manufactured by a tainted variable all the results they produce are masked as tainted. That is why it is essential to track both types of dependencies accurately.

[12] https://www.mozilla.org/el/firefox/new [Accessed: 4 July 2019]

```
var cookie = document.cookie;
var cookie_array = [];

for(i=0; i<cookie.length; i++) {
 cookie_array[i] = cookie[i];
}

var duplicate = '';
for(i=0; i<cookie_array.length; i++) {
     switch(cookie_array[i]) {
         case 'a': duplicate += 'a';
              break;
         case 'b': duplicate += 'b';
              break;
         .
         .
         .
     }
}

document.location =
     'http://www.malicious.com/cookiestealer.php?cookie='
     + duplicate;
```

Figure 37: XSS attack trying to override data and control dependencies

The mechanism responsible for propagating the taint is locating on the JavaScript engine of the web browser. Specifically, a JavaScript program is parsed and compiled into bytecode which is then executed by the JavaScript engine's interpreter. The engine is extended to understand when the result of a bytecode instruction could be tainted or not, by separating the instructions in four major categories where each category has its own rules regarding taint propagation:

- *Assignments*: Whenever a tainted value is assigned to another variable's value, the later one is marked as tainted. In case of object properties, when a field identifier of an object or an element of an array is tainted, the whole structure is marked tainted. That is so control-dependencies and function calls can propagate taint correctly.
- *Arithmetic and logic operations*: The result of such operations is tainted when one variable taking part in the operation is tainted.
- *Control structures and loops*: As mentioned before when a variable in the condition of such structures is tainted, every variable assigned value inside them is marked as tainted. In more detail, a *tainted scope* is wrapped around the control structure checking all the operations inside it marking the tainted variables according to the assignments taking place.
- *Function calls and `eval`*: Functions called inside tainted scope are regarded as tainted. If one of the parameters is tainted then the associated arguments are tainted to. Everything that is assigned inside a tainted function or returned by it is marked as tainted.

The *eval* method is treated in a special manner. The arguments given to it are executed as a JavaScript program. If a parameter of it is tainted or the *eval* function itself is tainted, due to being called in a tainted scope, everything inside it is regarded as tainted.

At last, another useful mechanic that the extended JavaScript engine has is that it manages another trick an attacker may use from his arsenal, that of DOM laundering. They can store a tainted variable into a DOM element and access it later, so the analysis will not be able to see through it and handle it as tainted. However, taint information is not lost when a variable leaves the JavaScript engine. So the attacker will not be able to trick the analysis.

4.2.2.2 Static Taint Analysis

As mentioned static taint analysis confronts issues that the dynamic taint analysis cannot track. Dynamic taint analysis cannot figure *indirect control dependencies*. Taking for example an *if-then-else* structure which is based on a tainted condition, operations taking place on the branch executed are the only ones marked as tainted, despite the tainted scope generated around the control structure.

This can be abused strategically by an attacker by evaluating always a branch as false in order to not mark a critical value inside the branch as tainted and use it later in the program to update him on his operations.

```
var cookie = document.cookie;

var correct = false;
var incorrect = false;

if (cookie[0] == 'a') {
      correct = true;
} else {
      incorrect = true;
}

if (correct === false) {
      document.location =
        'http://malicious.com/cookiedecryption.php?retry=true';
}

if (incorrect === false) {
      document.location =
        'http://malicious.com/cookiedecryption.php?cookiechar=a';
}
```

Figure 38: Cookie decryption by exploiting indirect control dependencies

This kind of an attack is what produces the need of a static taint analysis to thoroughly check indirect control dependencies by examining all the possible paths in the condition. The attacker by taking advantage of the correlation between the variables *correct* and *incorrect*, which a dynamic taint analysis cannot trace, he is able to decrypt the cookie of the user. Instead of checking a character one by one he can also try and guess the cookie value or perform a binary search. The point is he can leak information out of the application to his cause.

To counter such an attack the technique performs a linear static analysis in every branch of a control structure whose condition is related to a tainted variable. Precisely, if the static linear analysis passes an instruction responsible for assigning values to another variable inside the bytecode tainted scope of the tainted control structure it taints the variable.

Variables assigned inside such control structures, despite being executed or not, are considered vulnerable. In the previous example described such an analysis would characterize both variables as tainted and the attacker would not be able to leak information back to them.

However, instructions responsible for assigning the properties of an object use the stack to designate their target object, due to JavaScript being a stack-based language. That has led to the making of a stack analysis which sole purpose is to estimate the elements inside the stack for every instruction in the program. This stack analysis is achieved using an intraprocedural data flow analysis. As explained in the Introduction Section an intraprocedural analysis performs the instructions one by one and simulates its potential results and assumes that the procedures invoked may alter all the visible variables. In this paper, the simulations of the instructions are performed on an abstract stack.

In the implementation presented in the paper not all the bytecode instructions are correctly implemented to fit the process. Instructions like *throw* and *safe* have not been modeled. When the analysis tracks such instructions it automatically taints them.

5. OTHER TOOLS

Besides the frameworks we have analyzed that perform taint analysis, we felt it would be delicate to describe some other tools we came across that have the exact same goal but operate in a different way than the taint analysis standards presented before.

The first tool presented, employs an interesting test technique through *fuzzing* to the subject program while the second uses an API able to detect and sanitize the tainted input while the program is executing, so an attack is averted.

5.1 FLAX

The basic philosophy behind *FLAX framework* [11] is a type of dynamic analysis called *tainted enhanced blackbox fuzzing*. It is a hybrid technique combining the features of dynamic taint analysis and random fuzzing. Fuzzing or random black-box testing is a popular mechanism for testing applications. However, it doesn't perform too well when it has a large number of inputs.

In this approach it is used for creating tests which represent a Client-Side Vulnerability (CSV), confronting with this way the constraints of pure dynamic taint analysis. Some of the CSV it takes care are *XSS, Command Injection, Cookie-sink vulnerabilities* and *Origin Mis-attribution.*

It is also worth mentioning, that this mechanism eliminates the false warnings which would result from a taint analysis tool.

5.1.1 Taint Analysis

The taint analysis the tool performs is divided in five steps:

1. The application to be analyzed is executed. It is given a harmless input. The execution of it is operated in an instrumented browser resulting in the generation of a trace in JavaScript Simplified Instruction Language (JASIL). In a few words, the application is transformed to a JASIL IR trace appropriate for analysis.

   JASIL is an IR used to simplify the taint analysis, since it gets rid of the dynamic semantics of JavaScript by having a small set of operations that represent a subset of JavaScript's most used semantics in applications.

   In order to downgrade the semantics of JavaScript and generate a JASIL trace, Webkit's[13] open-source JavaScript interpreter, which is the core of the Safari web browser[14], got instrumented appropriately to translate the bytecode executing to JASIL form. Webkit's interpreter was also used in the JSBAF framework.

   Complex functions and mechanisms in the language get downgraded to simpler ones along with accesses like for example property look ups, creating and destroying objects, which are handled by map of the JASIL form. This simplicity that JASIL provides is key for the dynamic taint analysis of a complex language such as JavaScript.

   FLAX is not platform-independent, since in order to produce the JASIL trace there is need of a special instrumented browser.

2. Perform character-level dynamic taint analysis on the JASIL trace generated from step 1. The analysis tracks data flows from the input data to the critical sinks of the application and finds out which part of the initial benign data given to the application as input can be exploited.

[13] https://webkit.org [Accessed: 4 July 2019]
[14] https://safari.en.softonic.com [Accessed: 4 July 2019]

FLAX categorizes sinks according to their resulting exploit in case someone injects malicious code. Its taint analysis is the like the ones have described throughout the thesis. It carefully tracks the flow of the variables that are affected from a taint source until they reach a sink. For such flows that do not get sanitized while traversing paths to sinks, the tool groups them to subject them to fuzzing.

3. The statements dynamic taint analysis characterizes as potentially harmful are extracted into a program slice, called *acceptor slice* A_s. Such statements are the ones that effect data, which are arguments of a sink operation or are associated with the range of input characters that can be exploited according to the analysis.

   When the framework tracks such a sink, it goes backwards from the sink to the taint source and extracts the mentioned acceptor slice A_s. In order to continue to the next step, that of fuzzing, the acceptor slice is converted back to JavaScript from JASIL form.

4. This step describes the other half of the process, the *sink-aware random fuzzing*. Each A_s is fuzzed to find a set of inputs that may exploit a vulnerability in the application. Acceptor slices A_s are created and analyzed instead of the whole program because of their reduced size. Fuzzing can focus only on a fraction of the program input.

   Specifically, the tool applies random inputs/tests on each acceptor slice A_s based on the kind of sink they are based on. As mentioned before, FLAX categorizes all the sinks based on the attack someone can perform by exploiting them. Hence the term sink-aware random fuzzing, since the tool generates the appropriate tests for each sink.

5. Verify the inputs that the fuzzer produced for each acceptor slice. This process is done by performing the attacks on the web application. An oracle observing the attacks deduces the final results on whether the attacks were successful or not. If an attack was successful it notes it to the vulnerability report.

A detailed snapshot of the framework and its analysis can be seen below:

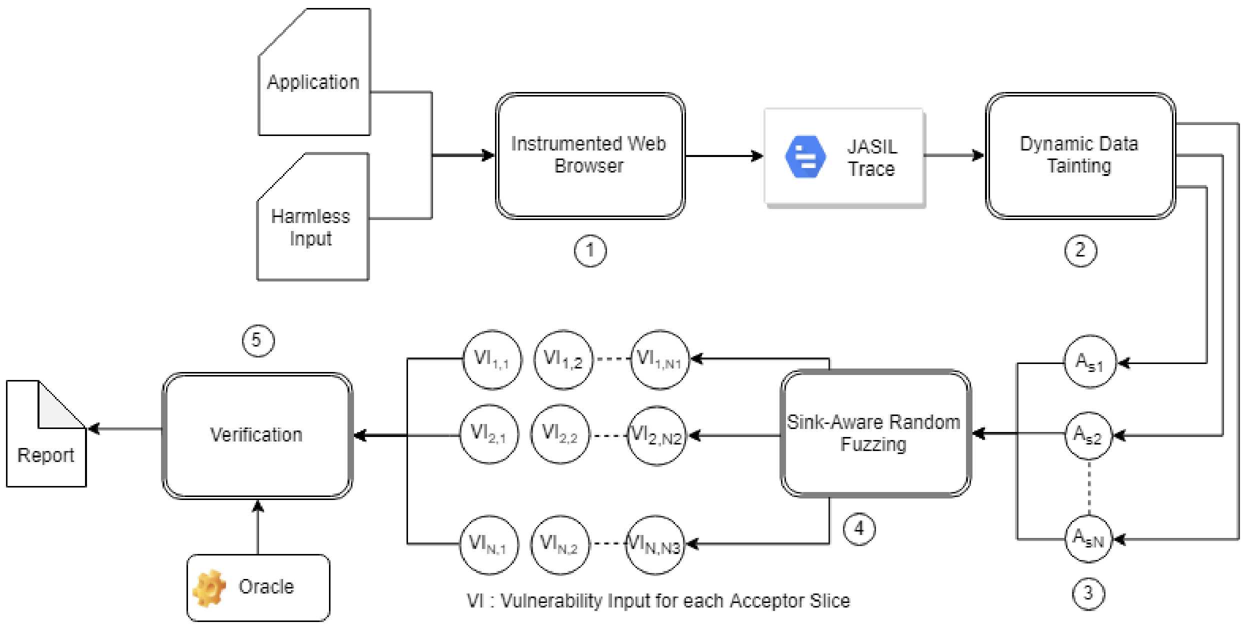


Figure 39: FLAX Analysis

5.2 Precise Taint Tracking

The approach presented in *Precise Taint Tracking* [12] refers to a method that expands *coarse-grained taint tracking*. It is based on another approach proposed by Nguyen-Tuong[13] called *precise tainting*, a framework that replaces the standard PHP interpreter with a modified interpreter that precisely tracks taintedness and checks for dangerous content in uses of tainted data.

Precise Taint Tracking tracks tainted input at the character level and gives the ability to the developer to sanitize the tainted strings and continue the execution of the application without the need to crash it. In more detail, it gives the opportunity to the library writers to sanitize data when developers misuse the libraries by performing bad programming techniques and on top of that warn them of their mistakes.

For example, in case of an SQL Injection Attack, the programmer is alarmed of his bad use of JavaScript libraries concerning the manipulation of the SQL queries and at the same time the dangerous SQL input is sanitized at its tainted substrings by adding escapes.

Firstly, there is presented an implementation of a coarse-grained taint tracking, the first technique implemented. Then using precise tainting the writers reached their final framework, precise taint tracking.

5.2.1 Coarse-Grained Taint Tracking

Coarse-grained taint tracking forms a robust approach for the taint analysis of script languages. Apart from tracking when one value is inserted in the program from outer untrusted sources it also aims in sanitizing such values on the runtime and producing warnings.

The implementation presented is based on the open-source JavaScript engine written in Java, Rhino[15] which is managed by the Mozilla Foundation[16]. For the tainting the technique appends a special character token at the end of the string which is considered dangerous. The Application Programming Interface for the analysis consists of three main functions:

- *`taint(input)`*: Takes a parameter input string *`input`* and applies a special token to the end of it to mark it as tainted. Returns a copy of the initial string.
- *`isTainted(str)`*: Returns *`true`* or *`false`* depending if the parameter *`str`* is tainted or not.
- *`untaint(tainted)`*: Untaints the parameter tainted string *`tainted`* and returns a copy of the string without the special token indicating that it is tainted.

An example:

```
var cookie = taint(document.cookie);
console.log(isTainted(cookie));
```

Figure 40: Coarse-grained taint tracking example

The cookie taken by the user's session is marked as tainted. Thus the console when calling the function *`isTainted`* on the variable *`cookie`* prints *`true`*.

[15] https://github.com/mozilla/rhino [Accessed: 4 July 2019]
[16] https://foundation.mozilla.org/en/ [Accessed: 4 July 2019]

5.2.2 Extending Coarse-Grained Taint Tracking

What led to the precise taint tracking was the inadequacy of the coarse-grained taint tracking technique to detect which portions of the string are tainted. Instead, the whole string was tainted. Hence, in case a small benign part of the string was assigned to another variable that variable was regarded as tainted. In the example below, the value *true* is printed to the console, which is a false positive.

```
var cookie = taint(document.cookie);
var msg = "Your cookie is: " + cookie;
var sliced = msg.slice(0, 4);
console.log(isTainted(sliced));
```

Figure 41: Issue Regarding Coarse-Grained Taint Tracking

For that cause, that implementation was extended to taint the values at a character level. Also, the API was extended to include two more functions:

- *taintedRegions(tainted)*: Returns an array of two numbers indicating the start and the end indexes of the tainted parts of the tainted input parameter *tainted*.
- *sanitize(tainted, callback)*: Allows the modification of the tainted string *tainted* by performing the function *callback* on it to each tainted region of the string.

and extend the already implemented functions to be more delicate:

- *taint(input)*: Also, appends a list of the ranges of the tainted parts of the tainted input string *input*.
- *isTainted(str)*: Now, correctly returns true only if there are portions of the string *str* tainted. In contrast to the example presented before in the coarse-grained taint tracking that presented a perfectly safe string as tainted.
- *untaint(tainted)*: Removes taint from the tainted string *tainted* by removing the special token at the end of the string along with the lists indicating the tainted regions.

Let us see how the precise taint tracking may be used to track down an attempt on SQL Injection and sanitize it at the same time, without crashing the actual program:

```
1. var name = "Bobby Drop Tables'; DROP TABLE Users; -- ";
2. var tname = taint(name);
3. var query = "SELECT * FROM Users WHERE UserName='" + tname + "';";
4.
5. console.log(isTainted(query));
6. console.log(isTainted(query.substring(0, 36));
7. console.log(taintedRegions(query));
8.
9. var safe = sanitize(query, function (str) {
10.                               return str.replace(/'/g, "''");
11.                          });
12. console.log(isTainted(safe));
```

Figure 42: Precise Taint Tracking managing an SQL Injection attempt

In line 1, the variable *name* which is given by the user and put inside the query includes malicious code. The attacker is aiming to delete the whole table *Users* inside the database. Thus, in line 2 it is marked as tainted and at line 9 it is sanitized by replacing the single quotes with two single quotes, as show at the two figures below, **Figure 43** and **Figure 44**. Hence, a possible SQL injection attack would result in an SQL syntax error.

```
Bobby Drop Tables'; DROP TABLE Users; --
                    ↓
Bobby Drop Tables''; DROP TABLE Users; --
```

Figure 43: Escaping the UserName given

```
SELECT * FROM Users WHERE UserName = 'Bobby Drop Tables'; DROP TABLE Users; -- ';
                                        ↓
SELECT * FROM Users WHERE UserName = 'Bobby Drop Tables''; DROP TABLE Users; -- ';
```

Figure 44: Result query after escaping the UserName given

Executing the example produces the following output in the console which shows the correct tainting of parts of the string in contrast to the whole tainting the coarse-grained taint tracking would perform:

```
true
false
[36, 76]
false
```

Figure 45: Console output executing the example

6. CONCLUSIONS

The purpose of the thesis was executing multiple tests on the pool of taint analysis frameworks collected to measure their Accuracy in terms of simple and complicated attacks. However, the majority of the tools we have accumulated are commercial without option for trial, apart from ACTARUS and ANDROMEDA. Both of these are in the IBM'S Security AppScan Source for analysis pack. Unfortunately, we were not given access to the trial version and thus we could not perform test cases on any framework.

Mitropoulos, *et al* [16] present a thorough classification of tools used to defend against web application attacks by analyzing each of them based on their:

- Accuracy
- Availability
- Ease of use
- Performance Overhead
- Security
- Detection Point

Since we did not manage to get our hands on the tools we examined, we are unable to analyze them in terms of *Accuracy, Ease of Use* and *Performance Overhead*. In addition to that, the *Detection Point* is the same for every one of them, since taint analysis tracks vulnerable input on the client-side.

Instead, we are going to emphasize on the differences on these frameworks. What does one do that the other fails to cover in the analysis, like for example handling the *eval* call? Which of them are platform-independent? Which of them are based on browsers?

Thus, we group and analyze our frameworks in a similar manner to provide some conclusions out of our research.

In our classification tables presented below, we only mention the taint analysis frameworks we have analyzed during the thesis and not techniques which have not been implemented in a tool, like for example the one presented in **Section 4.2**. Also, we have included the frameworks described in **Section 5** to compare them with the pure taint analysis tools. At last, we do not include general frameworks like *WALA* and *Jalangi*, even though it is able to build a taint checker on them.

6.1 General Conclusions

In **Table 1**, we present the frameworks and in which categories they belong based on their taint analysis:

Table 1: Frameworks' categories

Frameworks	Static Taint Analysis	Dynamic Taint Analysis
ACTARUS	✓	
ANDROMEDA	✓	
Ichnaea		✓
JSBAF	✓	✓
FLAX		✓
Precise Taint Tracking		✓

In **Table 2** we present the taint analysis tools along with the framework they base some of their key analyses taking place:

Table 2: Base analysis frameworks for the taint analysis

Frameworks	Base Framework
ACTARUS	WALA
ANDROMEDA	WALA
Ichnaea	Jalangi
JSBAF	WALA
FLAX	-
Precise Taint Tracking	-

As we can see, the majority of the frameworks use the IBM's WALA static analysis tool. We believe that happens because of the fact that it is open-source, it is highly customizable and provides a sound and novel approach to the dynamic features of JavaScript.

6.2 Attacks

Each framework handles different kinds of attacks. All of them, though, are able to handle the most prevalent attacks performed on web applications, those being *Cross-Site Scripting* and *Injection* attacks. In the *Injections* category we assume any type of injection with the most common being *SQL Injection* and *Command Injection.*

Table 3 presents a detailed series of attacks each framework is able to negate:

Table 3: Attacks successfully handled by each framework

Frameworks	Cross-Site Scripting	Injections	Unvalidated Redirects and Forwards	Log Forging	Origin Mis-attribution
ACTARUS	✓	✓	✓		
ANDROMEDA	✓	✓		✓	
Ichnaea	✓	✓			
JSBAF	✓	✓			
FLAX	✓	✓			✓
Precise Taint Tracking	✓	✓			

6.3 JavaScript feature extension

As mentioned on the **Section 2.1.1*,*** WALA takes care of the dynamic features of the JavaScript language, like prototype chains, object creations. Some of the tools using it for their static analysis decide to extend WALA's approaches, whereas others leave it as is. In **Table 4**, we present the tools that extend WALA's techniques. We are not sure how ANDROMEDA takes care of this matter, so we abstain from giving a false conclusion.

Table 4: Extension of the JavaScript features for the WALA-based frameworks

Frameworks	Eval Call	Arguments Array	Prototype Chain	Object Creation	Lexical Scoping
ACTARUS		✓	✓	✓	✓
ANDROMEDA	-	-	-	-	-
JSBAF	✓	✓			

6.4 Platform-independency

Tools that are platform independent do not modify the JavaScript interpreter in order to keep track of the taint analysis. There is need of a special instrumented browser that helps the framework in its execution. Such frameworks are Jalangi and Ichnaea. These specifically do not use a modified engine. Because of this platform-independency feature, Jalangi has the ability to perform its two phase analysis in two different machines.

The only taint analysis platform independent tool is Ichnaea due to the fact that its analysis relies on Jalangi.

ACTARUS, JSBAF and ANDROMEDA are based on WALA, which uses the Rhino parser by Mozilla Foundation to parse JavaScript and other data.

Platform independency is described in **Table 5**:

Table 5: Platform-independency

Frameworks	Platform-independent
ACTARUS	
ANDROMEDA	
Ichnaea	✓
JSBAF	
FLAX	
Precise Taint Tracking	

The tools that are not platform-independent, in most cases, base their analysis in a modified browser/engine or they are tied to a particular engine. Specific engines are used for the dynamic phases of the frameworks performing dynamic taint analysis. A mapping for each framework and the engine they use/modify is presented in **Table 6**:

Table 6: JavaScript engines that each platform-dependent framework uses

Frameworks	JavaScript Engine	
ACTARUS	Rhino	
ANDROMEDA	Rhino	
JSBAF	**Static:** Rhino	**Dynamic:** WebKit
FLAX	WebKit	
Precise Taint Tracking	Rhino	

6.5 Availability

All the taint analysis tools we have analyzed are only for commercial use. The only form of availability is that of a trial version for the IBM's *Security AppScan Source for analysis* package that includes the algorithms used for ACTARUS and ANDROMEDA.

Tools that are publicly available are described in **Table 7**:

Table 7: Available frameworks to the public

Frameworks	Available
ACTARUS	✓
ANDROMEDA	✓
Ichnaea	
JSBAF	
FLAX	
Precise Taint Tracking	

6.6 ACTARUS vs ANDROMEDA

ACTARUS and ANDROMEDA are two frameworks both produced by IBM and integrated in the same security package. For that purpose, we would like to compare them and present the pros and cons for each one of them. They are both based on an on-demand driver taint analysis using the notion of access paths. However, they have key differences. A detailed comparison is presented in **Table 8**.

Table 8: ACTARUS vs ANDROMEDA

ACTARUS	ANDROMEDA
Constructing a call-graph representing the whole program	Constructing a call-graph using lazy methods and **not** representing the whole program
Pointer analysis of the whole program	On-demand aliasing when needed
JavaScript front-end	JavaScript, .Net, Java front-end
-	Incremental Analysis

Despite their differences, the fact that they have been developed by the same company makes them have some major similarities. In addition to this, it should be noted that a group of people that contributed to the making of ACTARUS have also contributed to the making of the ANDROMEDA framework. In **Table 9** we present their correlation.

Table 9: Similarities between ACTARUS and ANDROMEDA

Similarities
On-demand driven taint analysis using access paths
Context sensitive taint analysis
Based on the IBM's WALA static analysis framework
Commercial products under the IBM's Security AppScan Source for analysis

6.7 Conclusion

XSS and Injections vulnerabilities have become increasingly popular due to the extensive use of JavaScript in web applications. Even though such attacks can be easily eliminated, the developers still use wrong programming techniques for the developing of a web application. Since, to err is human we felt it was a necessity to present automated tools in the form of taint analysis to protect both benign users from their leak of private information and enterprises from harming their reputation beyond repair.

Despite the fact that we were not able to measure their precision, we are optimistic that we have presented a thorough research and conclusions based on the resources we were given. In addition to this, we believe that we have demonstrated a comprehensive introduction to people who were not familiar to this technique until today. We hope sometime in the future; more researchers pick up taint analysis for analyzing JavaScript and produce sound techniques publicly available for a more secure Internet.

TABLE OF TERMINOLOGY

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Client-side	Πλευρά Πελάτη
Tainted	Στιγματισμένος
Source	Πηγή
Sanitize	Εξαγνίζω
Sink	Καταβόθρα
Tainting	Στιγματισμός
Taint	Μόλυνση

ABBREVIATIONS – ACRONYMS

3AC	Three Address Code
API	Application Programming Interface
AST	Abstract Syntax Tree
CFA	Control Flow Analysis
CFG	Control Flow Graph
CIA	Change Impact Analysis
CSV	Client-side Vulnerabilities
DOM	Document Object Model
DSL	Domain Specific Language
HTML	Hypertext Markup Language
IP	Internet Protocol
IR	Intermediate Representation
JASIL	JavaScript Simplified Instruction Language
JS	JavaScript
JSBAF	JavaScript Blended Analysis Framework
OOP	Object Oriented Programming
OWASP	Open Web Application Security Project
RHS	Reps-Horwitz-Sagiv
SSA	Static Single Assignment form
SQL	Structured Query Language
TAC	Three Address Code
URL	Uniform Resource Locator
WALA	T.J. Watson Libraries for Analysis
WWW	World Wide Web

REFERENCES

[1] Owasp.org, “OWASP Top 10 Application Security Risks”, 2017 [Online]. Available: https://www.owasp.org/index.php/Top_10-2017_Top_10 [Accessed: 25 May 2019].

[2] O. Tripp, S. Guarnieri, M. Pistoia, A. Aravkin, “ALETHEIA: Improving the Usability of Static Security Analysis”, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. (CCS '14),* ACM, 2014, pp. 764.

[3] J. Dolby, M. Sridharan, “Static and Dynamic Program Analysis Using WALA (T.J. Watson Libraries for Analysis)”, PLDI 2010.

[4] K. Sen, S. Kalasapur, T. Brutch, S. Gibbs, “Jalangi: a selective record-replay and dynamic analysis framework for JavaScript”, *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013),* ACM, 2013, pp. 488-498.

[5] M. Sridharan, K. Sen, L. Gong, “Jalangi: A Dynamic Analysis Framework for JavaScript”. [Online]. Available: https://manu.sridharan.net/files/JalangiTutorial.pdf, pp. 1-44. [Accessed: 25 Jun. 2019]

[6] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, R. Berg, “Saving the world wide web from vulnerable JavaScript”, *Proceedings of the 2011 International Symposium on Software Testing and Analysis. (ISSTA ’11),* ACM, 2011, pp. 177-187.

[7] O. Tripp, M. Pistoia, P. Cousot, S. Guarnieri, “Andromeda: Accurate and Scalable Security Analysis of Web Applications”, *Fundamental Approaches to Software Engineer (FASE 2013)*, Springer, Berlin, Heidelberg, 2013, pp. 210-225.

[8] R. Karim, F. Tip, A. Sochurkova, K. Sen, “Platform-Independent Dynamic Taint Analysis for JavaScript”, *IEEE Transactions of Software Engineering (IEEE),* Early Access.

[9] S. Wei, B.G. Ryder, “Practical blended taint analysis for JavaScript”, *Proceedings of the 2013 International Symposium on Software Testing and Analysis. (ISSTA 2013),* ACM, 2013, pp. 336-346.

[10] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, G. Vigna, “Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis”, *Proceedings of the Network and Distributed System Security Symposium (NDSS 2007)*, San Diego, California, USA, 2007.

[11] P. Saxena, S. Hanna, P. Poosankam, D. Song, “FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications”, *Proceedings of the Network and Distributed System Security Symposium (NDSS 2010)*, San Diego, California, USA, 2010.

[12] T. Saoji, T. H. Austin, C. Flanagan, “Using Precise Taint Tracking for Auto-sanitization”, *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Securit. (PLAS ’17),* ACM, 2017, pp. 15-24.

[13] D. Greene, D. Evans, A. Nguyen-Tuong, J. Shirley, S. Guarnieri, “Automatically Hardening Web Applications Using Precise Tainting”, *IFIP International Information Security Conference*, Springer, 2005, pp. 295-307.

[14] T. Reps, S. Horwitz, M. Sagiv, “Precise interprocedural dataflow analysis via graph reachability”, *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL ’95)*, ACM, 1195, pp. 49-61.

[15] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques and Tools*, 2006, pp. 9, 10, 25, 91-105, 357-370, 399-408, 597-618, 903-917.

[16] D. Mitropoulos, P. Louridas, M. Polychronakis, A. D. Keromytis, “Defending Against Web Application Attacks: Approaches, Challenges, Implications”, *IEEE Transactions on Dependable and Secure Computing,* vol. 16, issue 2, pp. 188-203, March 2017.

[17] Owasp.org, “Cross-Site Scripting (XSS)”, 2018. [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS) [Accessed: 5 Jun. 2019].

[18] Owasp.org, “Command Injection”, 2018. [Online]. Available: https://www.owasp.org/index.php/Command_Injection [Accessed: 5 Jun. 2019].

[19] Owasp.org “SQL Injection”, 2018. [Online]. Available: https://www.owasp.org/index.php/SQL_Injection [Accessed: 5 Jun. 2019].

[20] Mozilla.org, “Object.prototype”, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/prototype [Accessed: 10 Jun. 2019].

[21] Mozilla.org, “Hoisting”, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Glossary/Hoisting [Accessed: 10 Jun. 2019].

[22] Mozilla.org, “Closure”, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Closures [Accessed: 10 Jun. 2019].

[23] Mozilla.org, “The arguments object”, 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments [Accessed: 10 Jun. 2019].

[24] IBM, “WALA: T.J. Watson Libraries for Analysis”, 2015. [Online]. Available: http://wala.sourceforge.net [Accessed: 20 Jun. 2019].