View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by SJSU ScholarWorks

San Jose State University
SJSU ScholarWorks

Master's Projects

Master's Theses and Graduate Research

Spring 5-16-2016

Taint and Information Flow Analysis Using Sweet.js Macros

Prakasam Kannan
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the Information Security Commons, and the Programming Languages and Compilers Commons

Recommended Citation

Kannan, Prakasam, "Taint and Information Flow Analysis Using Sweet.js Macros" (2016). *Master's Projects*. 468.
DOI: https://doi.org/10.31979/etd.qsyz-fu42
https://scholarworks.sjsu.edu/etd_projects/468

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Taint and Information Flow Analysis Using Sweet.js Macros

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Prakasam Kannan

May 2016

© 2016

Prakasam Kannan

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Taint and Information Flow Analysis Using Sweet.js Macros

by

Prakasam Kannan

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2016

Thomas H. Austin	Department of Computer Science
Mark Stamp	Department of Computer Science
Ronald Mak	Department of Computer Science

ABSTRACT

Taint and Information Flow Analysis Using Sweet.js Macros

by Prakasam Kannan

JavaScript has been the primary language for application development in browsers and with the advent of JIT compilers, it is increasingly becoming popular on server side development as well. However, JavaScript suffers from vulnerabilities like cross site scripting and malicious advertisement code on the the client side and on the server side from SQL injection.

In this paper, we present a dynamic approach to efficiently track information flow and taint detection to aid in mitigation and prevention of such attacks using JavaScript based hygienic macros. We use Sweet.js and object proxies to override built-in JavaScript operators to track information flow and detect tainted values. We also demonstrate taint detection and information flow analysis using our technique in a REST service running on Node.js.

We finally present cross browser compatibility and performance metrics of our solution using the popular SunSpider benchmark on Safari, Chrome and Firefox and suggest some performance improvement techniques.

ACKNOWLEDGMENTS

I want to thank Dr.Austin for guiding me though this project, reviewing and editing this report and offering valuable feedback and setting me back on track when I veered off the course. I want to the thank Dr. Stamp and lecturer Ronald Mak for agreeing to be committee members and reviewing the report. Finally I want to thank Dr.Tim Disney for offering ideas and clarifying doubts while implementing taint and information flow analysis in Sweet.js.

TABLE OF CONTENTS

CHAPTER

1 Introduction 1

2 Background 4

2.1 Web Security Model 4

2.2 Same Origin Policy 4

2.3 Dynamic Taint Analysis 7

2.4 Dynamic Information Flow Analysis 7

2.4.1 Implicit Flows 8

2.4.2 No-Sensitive-Upgrade Check 9

2.5 Macro Systems 9

2.5.1 Sweet.js 10

3 Taint Analysis 11

3.1 Design 11

3.1.1 Express.js 12

3.1.2 MySQL Driver 13

3.2 Examples 13

4 Information Analysis 17

4.1 Design 17

4.2 Examples 19

5 Performance Tests 22

6 Conclusion 27

APPENDIX

Virtual Values 31

A.1 Macros 31

A.2 Harness 32

A.3 Taint Functions 36

LIST OF TABLES

1 Taint Performance Test Resutls 23

2 SunSpider Safari Performance Test Results 24

3 SunSpider Chrome Performance Test Results 25

4 SunSpider Firefox Performance Test Results 26

LIST OF FIGURES

1	Modern Web Site [3]	1
2	Frame Navigation Policies [13]	5
3	Explicit Leak	19
4	Explicit Leak after compilation using Sweet.js	20
5	Implicit Leak	20
6	Implicit after compilation using Sweet.js	21
7	Taint Performance Test Result	23

CHAPTER 1

Introduction

Web application vulnerabilities like cross site scripting (XSS)[1] and SQL injection have devastating effect in the Internet. Though these vulnerabilities are well known and preventive measures are well documented, a lot of applications continue to suffer from such attacks.

Similarly, a growing number of web applications as shown in Figure 1 use other third-party libraries and APIs directly on their web pages in mash-ups. It is very common for many websites to embed Google Maps or other mapping gadgets to display directions or URL bookmarking and sharing widgets. Also many publication platforms, news outlets, and ecommerce sites display advertisements but don't control either their origin or content [2].

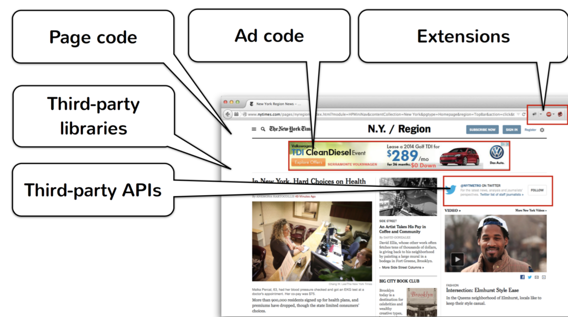

Figure 1: Modern Web Site [3]

It is highly desirable to restrict access to the origin page's resources from these third-party scripts and separate application security from application functionality through language runtime support combined with seamless application framework integration. For example, the majority of modern languages like Java, JavaScript, Go, etc. provide built-in garbage collection, safe memory, and array access guarantees. In

the same vein, we explore the approach of dynamic taint detection to ensure integrity of the data and dynamic information flow to guarantee confidentiality. We lean towards dynamic checking as opposed of static checking [4] [5] to provide a better user experience by avoiding expensive upfront static analysis on each client browser before execution. Dynamic analysis also allows for more flexibility in tailoring policies based on browser capabilities and hot swap policies [7].

We achieve information flow tracking and taint detection using sparse labeling[6], where only tainted values have an explicit label. This approach allows us to easily track sensitive data, while avoiding unnecessary overhead. We leave it up to the frameworks to label any suspect values (in the case of taint detection) and to the programmers to label any sensitive data whose flow they want to track and restrict.

Languages like JavaScript, Java, and SmallTalk, provide facility to proxy operations on any object values either to override the behavior or delegate to the underlying object. This is known as *behavioral intercession.* While all values in SmallTalk are objects, values in JavaScript or Java can be either a primitive or an object, and there is no language level feature to proxy primitive values to achieve behavioral intercession. We wrap JavaScript primitive values inside Sweet.js Virtual Values [9] to make them amenable for behavioral intercession.

Applications perform tasks by executing operations on data. Sweet.js [8] produces virtual values by virtualizing the interface between operations and the data. Each virtual value is a collection of traps corresponding to various legal operations that can be performed on a JavaScript value. These traps are in turn user defined functions that describe how a specific operation should behave. Virtual values also serve as labels identifying tainted or sensitive data.

This paper is organized as follows. This next chapter provides background information on web application vulnerabilities, taint, and information flow analysis, and macro systems. in chapters 3 and 4 we provide implementation details of taint analysis and information flow analysis. Chapter 6 discusses related work and future improvements to the implementation.

CHAPTER 2

Background

Taint analysis is an efficient mechanism for ensuring integrity of data flowing from browsers to a web application server. Information flow analysis guarantees the confidentiality of data within the browser environment. We first introduce the security model of the browser and vulnerabilities it suffers from. We then describe how taint and information flow analysis can be leveraged to mitigate these vulnerabilities.

2.1 Web Security Model

Web application security is enforced by network protocols and the browser's same origin policy (SOP) [11]. Network protocols like TLS and SSL guarantee the integrity and confidentiality of the information flow between the browser and the server. SOP guarantees that resources from the same origin (i.e from same host, port and protocol combination) have unfettered access to each other but resources from other origins do not.

2.2 Same Origin Policy

Web pages are made up of static content comprising HTML markup, style-sheets, fonts, images, videos etc., along with JavaScript. When a page is rendered, static content is converted to an object representation known as the Document Object Model (DOM). Browser components like window, location, history, etc. are also exposed for scripting as a part of the DOM.

The browser window object is the top level element in a DOM and contains the document object representing the page being viewed. The window can in turn contain

frames or iframes, which are either made up documents from the same or a different origin. These frames are the primary mechanism of isolation in browsers. Frame navigation policy lays out inter-frame DOM element and JavaScript access rules.

In the past, browsers supported 4 different frame navigation policies. Today, most have converged on Descendant Policy to provide better balance between security and usability [12]. Figure 2 succinctly describes all four navigation policies.

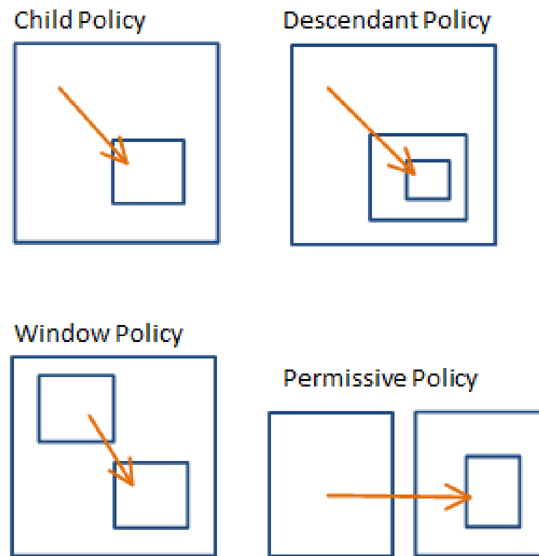

Figure 2: Frame Navigation Policies [13]

1. Permissive policy: Frames and iframes on any window have unfettered access to frames in same or other windows, most browsers deprecated this policy around 1999.

2. Window policy: Frames from same window are allowed to access each other, though this policy is little stricter than permissive policy it still suffers from attacks from malicious adcode or mashup gadget.

3. Child Policy: This is a most strict policy of all, frames are allowed to access

only child frames that they created, due to poor usability concerns never gained much traction.

4. Descendant policy: Stricter than window policy but less so than child policy allows access to child or grandchild.

While scripts loaded on different frames are isolated from each other based on origin, scripts loaded on a single frame from different origins are not isolated from each other. Therefore, third-party libraries have the same privilege as the page. Third-party libraries and ad-code loaded in such a manner are potential vectors for cross site scripting (XSS) and cross site request forgery (CSRF) attacks. While the iframes are restricted from accessing DOM of descendant iframes they can talk to other iframes though the `postMessage` method of `window` and thus can leak data arbitrarily.

Modern mechanisms like iframe sandboxing, content security policy (CSP), sub-resource integrity (SRI), and cross-origin resource sharing are aimed at improving the security and usability of the browser platform. However, developers often work around the restriction of these facilities which can result in dangerous security holes. For example, white-listing privileges like `allow-scripts` and `allow-forms` can lead to XSS attacks but disallowing them can limit functionality to a great extent. Similarly, CSP can be used to white-list sources to which XML HTTP Request (XHR) can be made or from which JavaScript can be loaded. However, these restrictions are specified in HTTP headers and thus cannot sandbox much prevalent inline JavaScript code without the source providing a precomputed hash or nonce (a randomly generated number) for each inline code snippet.

2.3 Dynamic Taint Analysis

Dynamic analysis is the ability to examine the code as it executes. It has become an indispensable tool in performance improvement (just-in-time compilers), automatic memory management (garbage collectors and escape analysis), and not the least in ensuring security analysis and enforcement. Dynamic taint analysis monitors program execution to keep track of tainted data. It either propagates the taint when certain operations are performed on tainted data or prohibits other operations based on the taint policy. Taint analysis generally suffers from two types of errors:

1. Overtainting, occurs when taint analysis marks certain a value as tainted when it is not.

2. Undertainting, occurs when taint analysis marks certain a value as clean when it is actually tainted.

We use Sweet.js virtual values to store and label values from certain sources as tainted and this label is propagated to any new value that is obtained by applying any of the standard JavaScript operators. We also provide functions that can be used to assert if a specific value is tainted before the value is used for some sensitive operation, such as using the value to parameterize a SQL query. In this way, user and library functions can defend against attacks and ensure the integrity of the system.

2.4 Dynamic Information Flow Analysis

Modern web pages mix together content, ad-code, and third-party libraries. Thus the ability to protect sensitive information against untrusted code is important to guarantee the confidentiality of the system.

2.4.1 Implicit Flows

Information flow analysis classifies values into different security levels. Values that are public (or low security) are labeled as L, values that are secret (or high security) are labeled as H. The primary goal of information flow analysis is to guarantee that the information flows only upwards, assuming that $L \leq H$.

Based on this definition, taint analysis can be viewed as a special case of information flow analysis where tainted values are H values and untainted values are L values. Thus $Untainted \leq Tainted$ and tainted values are prohibited from leaking into untainted values.

Information flow is either an explicit or implicit. Let's see some examples in each category of these flow types assuming that `secret:` H and `leak:` L.

```
leak = secret;
```

However, implicit flows are subtler [16]. The following, example leaks `secret`; it uses `secret` variable in a conditional statement to infer the value contained in it and returns inferred result.

```
y = true;
z = true;

if (secret)
  y = false;

if (y)
  z = false;

leak = z;
```

Arrays can lead to very subtle implicit flows, and thus leak information [15]. In

the following example, H value is used as a subscript of an array whose elements are initialized to 0, to leak `secret`.

```
a[secret] = 1;

for (int i = 0; i < a.length; i++) {
  if (a[i] == 1) leak = i;
}
```

2.4.2 No-Sensitive-Upgrade Check

It is clear from the above examples that the key challenge to dynamic information flow lies in handling implicit flows. One solution to handling implicit flows is to stop the program execution on unsafe updates; when a conditional branch is taken based on a secret value flags a public value, it is unsafe to update public variables in this context. Stopping these leaks can be achieved by pushing a confidential flag on to a stack when a branch is taken based on a secret value and popping it on the branch exit. When this stack flag is present, any update to public variables halts program execution.

2.5 Macro Systems

Macro programming systems have been around since the days of assembly programming [17]. Macro systems were originally used for text/token substitution [18] to ease coding in assembly language and provide for conditional compilation in `c`. However, they have evolved into systems of syntactic abstraction [19] that enable extending the programming language itself.

Macro systems are very popular in Lisp and Scheme [8] as programs in these languages are written as s-expression [20]. Symbol expressions or s-expressions are simply an atom or list of atoms that can represent both data and code, in LISP an

atom is the basic syntactic unit. Symbol expressions can be easily parsed in to abstract syntax tree thus a macro implementation is able to easily expand a s-expression into a complex s-expression. Lack of such delimited s-expression in modern programming languages rendered the ecosystem unsuitable for evolution of macro languages.

2.5.1 Sweet.js

In JavaScript, the compiler, parser, and lexer are intertwined due to its syntactic ambiguity. For example, the token `'/'` is used to delimit both regular expression literals and operands of division operation. Sweet.js overcomes these limitations by introducing a reader sandwiched between the lexer and parser, similar to the one found in Scheme.

The Sweet.js reader converts the tokens into token trees, which provides enough information to disambiguate the syntax without any help from the parser. Sweet.js macros are also hygienic; hygiene prevents variable names defined inside macros from colliding with variable names in the surrounding JavaScript code. Sweet.js achieves hygiene by keeping track of lexical context and rewriting variable names during the macro expansion phase.

CHAPTER 3

Taint Analysis

The goal of our taint analysis implementation is to provide a means to facilitate taint introduction, taint propagation, and taint checking. To demonstrate the application of taint introduction and propagation facilities, we have provided an Express.js middleware that automatically taints HTTP query parameters and `application/json` payloads, and a web application leveraging the middleware. Additionally, we have also provided a modified MySQL database driver that perform taint check on SQL queries before executing them to prevent SQL Injection attacks.

3.1 Design

The Sweet.js virtual values, made up of a set of macros, listed in section A.1, that rewrites standard operators in JavaScript in to one of the following function calls and an harness, listed in section A.2, that implement the same.

1. unary - for unary operations
2. binary - for binary operations

This harness invokes either a trap if an operand is a virtual value proxy or performs the standard JavaScript operation when the value is a primitive. Our taint introduction function, listed in section A.3, wraps a tainted value inside a JavaScript proxy that implement these traps. These traps implemented in a virtual value proxy enable us to propagate taint as a tainted value is modified using any of the overridden operators.

1. unary - for unary operations
2. left - for binary operations where the proxy is on the left
3. right - for binary operations where the proxy is on the right

Compiling any JavaScript file using the virtual values compiler injects these macros, harness, and proxy code into the source and rewrites all JavaScript operations to enable taint analysis. While this approach is fine for small snippets of code, a large code base is rendered unreadable and results in a lot of code duplication. One popular approach that makes such mangled code readable and friendly to debug is Source Map [21]. An alternate strategy is to separate the macros, harness, and proxy code into Commons.js [22] module and import it into the source file by reference. This modular approach eliminates code duplication and minimizes the code mangling with less effort compared to Source Map, so we choose the Common.js approach.

3.1.1 Express.js

Express.js is a popular framework for building web applications and REST services on Node.JS. In the heart of Express.js is a router that maps HTTP URL paths to a JavaScript module and a customizable pipeline that surrounds the module servicing the HTTP requests known as middleware. To demonstrate the ability of taint analysis to prevent XSS and SQL injection attacks we have developed a middleware that taints HTTP request query parameters and `application/json`. This middleware can be enabled in any application built using Express.js just by configuration.

3.1.2 MySQL Driver

While the Express.js middleware automatically taints the user inputs there is no guarantee that the application code cleanses it before appending it to a SQL query and passing it on to a database driver. We have customized a Node.js MySQL driver to check for tainted SQL queries and reject them before execution. Any web application built using this driver in conjunction with the Express.js middleware described in the previous section can effectively prevent SQL injection attacks.

3.2 Examples

The following is a REST service that returns students' information from a MySQL database written in JavaScript leveraging our tainting library, our Express.js middleware and our taint aware MySQL driver. This application reads the query parameter `lastname` and uses it to construct a SQL query, which it passes on to the database driver. When the driver detects that a SQL query is tainted it raises an exception.

```
"use strict";
var mysql = require("mysql");
function search(req, res) {
    query(req.query.lastname, function(err, rows) {
        if (err) {
            return res.send(JSON.stringify(err));
        }
        res.send(JSON.stringify((rows)));
    });
}
```

```
function query(lastname, callback) {
    var connection = mysql.createConnection({
        host      : "localhost",
        user      : "webappuser",
        password  : "WSXwer43@",
        database  : "test",
        multipleStatements: true
    });

    var searchSQL =  "select * from student"
        + " where lastname = \"" + lastname + "\";";
    connection.connect();
    connection.query(searchSQL, function(err, rows, fields) {
        if (err) {
            return callback(err, null);
        }
        callback(null, rows);
    });
    connection.end();
}
module.exports = search;
```

The following is the same code after compilation using, the Sweet.js virtual values extension to propagate taint along as the `lastname` being appended to the SQL query.

```
"use strict";
```

```
var mysql$1381 = require("mysql");
function search$1382(req$1384, res$1385) {
    query$1383(req$1384.query.lastname,
        function (err$1386, rows$1387) {
            if (err$1386) {
                return res$1385.send(JSON.stringify(err$1386));
            }
            res$1385.send(JSON.stringify(rows$1387));
    });
}
function query$1383(lastname$1388, callback$1389) {
    var connection$1390 = mysql$1381.createConnection({
        host: "localhost",
        user: "webappuser",
        password: "WSXwer43@",
        database: "test",
        multipleStatements: true
    });
    var searchSQL$1393 =
        vvalues.binary("+", vvalues.binary(
        "+", "select * from student where lastname = \"",
        lastname$1388), "\";");

    connection$1390.connect();
    connection$1390.query(searchSQL$1393,
```

```
    function (err$1394, rows$1395, fields$1396) {
      if (err$1394) {
        return callback$1389(err$1394, null);
      }
      callback$1389(null, rows$1395);
    });
    connection$1390.end();
}
module.exports = search$1382;
```

CHAPTER 4

Information Analysis

Information flow analysis is a more general case of taint analysis. While taint analysis guarantees integrity, information flow guarantees confidentiality. Information flow analysis enable us build applications using third-party libraries and gadgets, shortening application development time without sacrificing the confidentiality of the application's data.

4.1 Design

In addition to the changes in the virtual value proxy traps, with minimal changes to our original macro and harness we were able achieve information flow analysis functionality. To enable virtual values to detect leakage of confidential information, we enabled the compiler to rewrite the standard assignment operator to function calls with the help of the following macro.

```
operator = 3 left { $left, $right } => #{ (function() {
    $left = vvalues.binary("=", $left, $right) })() }
```

However, detecting implicit flows requires an additional macro to override conditional statements as shown below. This macro rewrites conditional statements so that the result of conditional operation is pushed onto a stack. If the conditional operation involved any confidential information, the result of the operation is tagged as an H value.

```
let if = macro {
    case { _ ($expr) { $body ...} } => {
```

```
            return #{
                vvalues.push($expr);
                if (vvalues.peek()) { $body ... }
                vvalues.pop();
            }
        }
    }
```

On assignment of a L value to another L valued variable, the harness checks the stack for an H value, signifying assignment being made on a conditional branch involving confidential data. If there is an H value on the stack, the harness raises an exception.

```
vvalues.binary = function (a0, a1, a2) {
    ...
    if (a0 === "=") {
        if (vvalues.peek() && isSecret(vvalues.peek())) {
            throw new Error("Implicitly leaks secret");
        }

        var left = a1;
        var right = a2;
        return left = right;
    }
    ...
}
```

Finally, the virtual value proxy trap is also modified to raise an exception on assignment of confidential data to a public variable. Recursive definition of the assignment operation caused the compilation to expand the macro recursively until running out stack space. We overcome this problem by substituting ':=' symbol for '=' symbol though it is not the ideal solution.

4.2 Examples

The example in Figure 3 shows a JavaScript code snippet that leaks confidential information explicitly. Figure 4 shows the same code after compilation using the Sweet.js that would track and raise exception. We create confidential data using, the function `secret` which wraps the plain JavaScript value inside a virtual value proxy. Rewritten JavaScript operations delegate to the proxy, which raises an exception upon leakage of information.

```
"use strict";
require("./lib/secret");

var x = secret("x");
var y = secret("");
var z = false;

y = x;
z = 1;
z = x;
```

Figure 3: Explicit Leak

This example in Figure 5 shows an implicit flow inside a conditional branch taken based on a confidential value. Figure 6 shows the same code after compilation using the Sweet.js in which first the result of conditional operation is pushed on to a stack, then the branch is taken if the conditional operation evaluated to be true. Finally,

```
"use strict";
require("./lib/secret");

var x$1431 = secret("x");
var y$1432 = secret("");
var z$1433 = false;

(function () {
    y$1432 = vvalues.binary("=", y$1432, x$1431);
}());
(function () {
    z$1433 = vvalues.binary("=", z$1433, 1);
}());
(function () {
    z$1433 = vvalues.binary("=", z$1433, x$1431);
}());
```

Figure 4: Explicit Leak after compilation using Sweet.js

the result of the conditional operation is popped out of the stack just outside the conditional branch. Since the assignment in the body of the conditional branch is from a public variable to another public variable, it is intercepted by the harness instead of the proxy. The harness in turn peeks at the stack and raises an error if the peeked value is confidential.

```
"use strict";
require("./lib/secret");

var x = secret("x");
var z = false;

if (x) {
    z = true;
}
```

Figure 5: Implicit Leak

```
"use strict";
require("./lib/secret");

var x$1431 = secret("x");
var z$1433 = false;

vvalues.push(x$1431);
if (vvalues.peek()) {
    vvalues.binary("=", z$1433, true);
}
vvalues.pop();
```

Figure 6: Implicit after compilation using Sweet.js

CHAPTER 5

Performance Tests

In order to quantify the performance overhead of virtual value proxies that enable taint and information flow analysis and to demonstrate cross browser compatibility of the solution, We modified and compiled the popular SunSpider JavaScript performance benchmark [23] using Sweet.js compiler and tested it on Safari, Chrome, and Firefox and compared the results with the baseline. These tests were run on a Mac Book Pro with one 2.6 GHz Intel Core i7 processor containing 4 cores, and 16 GB of RAM, and Intel Iris Pro graphics processor with 1536 MB of memory.

We also took one of the test (`validate-input`) in the performance test suite that generates about 4000 email addresses and zip codes and validates them using Regular Expression and tainted a portion of these random email addresses and zip codes incrementally to measure the performance overhead of taint analysis and tabulated the results as well.

We chose the SunSpider benchmark, as it is focuses on a wide range of JavaScript features from Date, String, and Regexp manipulation to a wide variety of numerical, array-oriented, object-oriented, and functional idioms. The SunSpider test suite also focuses more on the client side, where JavaScript engines do not have much time to optimize the code, as most of the event handlers run for a short time. This benchmark also claims to be statistically sound, as it runs the tests several times and determines the error range with a 95% confidence interval.

Rewriting JavaScript operations into function calls, comes with a certain performance penalty, but progress in software development like, higher level languages,

automatic memory management, etc., often starts out with very high performance penalties and improves rapidly. We are hopeful such performance improvements can be made to our solution as well.

We can augment the Sweet.js virtual value compiler to identify expressions that do not involve proxies during the parse phase and leave them intact instead of rewriting operations into function calls. We believe that due to label locality [6] combined with the small proportion of code involving tainted and confidential data, we can improve the performance.

We excluded 3 tests cases, since they contain minified JavaScript, which cannot be reliably modified before compilation using Sweet.js.

Table 1: Taint Performance Test Resutls

Number of Variables	Tainted Variables	Mean	95% CI
4000	40	18.6ms	+/- 3.7%
4000	80	19.1ms	+/- 2.1%
4000	160	19.1ms	+/- 2.1%
4000	320	19.2ms	+/- 3.4%
4000	640	19.3ms	+/- 3.0%
4000	1280	19.5ms	+/- 3.6%

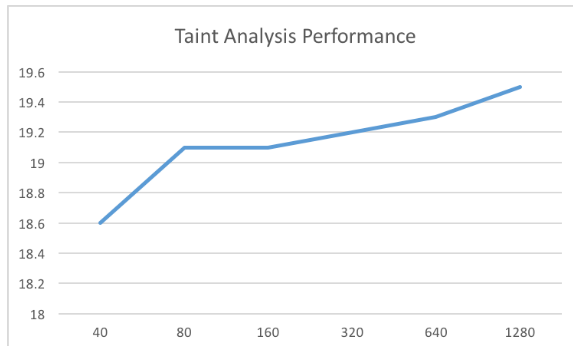

Figure 7: Taint Performance Test Result

Table 2: SunSpider Safari Performance Test Results

Test	Safari Baseline		Safari with Virtual Values	
	Mean	95% CI	Mean	95% CI
3d	10.0ms	+/- 3.4%	73.3ms	+/- 1.1%
cube	5.0ms	+/- 6.7%	31.8ms	+/- 1.8%
morph	5.0ms	+/- 0.0%	41.5ms	+/- 0.9%
access	13.0ms	+/- 5.2%	122.0ms	+/- 0.7%
binary-trees	2.2ms	+/- 20.5%	8.7ms	+/- 8.7%
fannkuch	5.2ms	+/- 5.8%	72.6ms	+/- 0.5%
nbody	2.6ms	+/- 14.2%	23.0ms	+/- 1.5%
nsieve	3.0ms	+/- 0.0%	17.7ms	+/- 2.0%
bitops	9.1ms	+/- 2.5%	159.1ms	+/- 0.6%
3bit-bits-in-byte	1.0ms	+/- 0.0%	30.0ms	+/- 0.0%
bits-in-byte	3.0ms	+/- 0.0%	35.0ms	+/- 0.0%
bitwise-and	2.0ms	+/- 0.0%	28.5ms	+/- 2.7%
nsieve-bits	3.1ms	+/- 7.3%	65.6ms	+/- 1.2%
controlflow	2.1ms	+/- 10.8%	14.4ms	+/- 2.6%
recursive	2.1ms	+/- 10.8%	14.4ms	+/- 2.6%
crypto	5.0ms	+/- 16.5%	42.0ms	+/- 0.8%
md5	2.4ms	+/- 28.8%	21.0ms	+/- 0.0%
sha1	2.6ms	+/- 14.2%	21.0ms	+/- 1.6%
date	5.2ms	+/- 10.8%	9.3ms	+/- 3.7%
format-xparb	5.2ms	+/- 10.8%	9.3ms	+/- 3.7%
math	9.3ms	+/- 3.7%	81.2ms	+/- 1.0%
cordic	3.0ms	+/- 0.0%	40.7ms	+/- 1.7%
partial-sums	4.3ms	+/- 8.0%	15.0ms	+/- 0.0%
spectral-norm	2.0ms	+/- 0.0%	25.5ms	+/- 2.0%
regexp	5.7ms	+/- 10.3%	5.3ms	+/- 6.5%
dna	5.7ms	+/- 10.3%	5.3ms	+/- 6.5%
string	23.7ms	+/- 1.5%	81.5ms	+/- 2.3%
base64	4.3ms	+/- 8.0%	22.2ms	+/- 1.4%
fasta	6.1ms	+/- 3.7%	25.1ms	+/- 1.6%
tagcloud	8.9ms	+/- 2.5%	19.5ms	+/- 1.9%
validate-input	4.4ms	+/- 8.4%	14.7ms	+/- 11.5%
Total	83.1ms	+/- 3.1%	588.1ms	+/- 0.6%

Table 3: SunSpider Chrome Performance Test Results

Test	Chrome Baseline		Chrome with Virtual Values	
	Mean	95% CI	Mean	95% CI
3d	18.0ms	+/- 13.20%	75.5ms	+/- 2.40%
cube	8.8ms	+/- 12.60%	33.3ms	+/- 3.70%
morph	9.2ms	+/- 14.10%	42.2ms	+/- 2.10%
access	11.5ms	+/- 4.40%	101.7ms	+/- 0.90%
binary-trees	1.5ms	+/- 25.10%	7.9ms	+/- 5.10%
fannkuch	5.6ms	+/- 6.60%	55.4ms	+/- 1.60%
nbody	2.1ms	+/- 10.80%	23.0ms	+/- 4.10%
nsieve	2.3ms	+/- 15.00%	15.4ms	+/- 2.40%
bitops	18.9ms	+/- 2.10%	126.5ms	+/- 3.40%
3bit-bits-in-byte	1.0ms	+/- 0.00%	25.7ms	+/- 6.90%
bits-in-byte	3.8ms	+/- 7.90%	30.9ms	+/- 4.80%
bitwise-and	11.1ms	+/- 2.00%	25.5ms	+/- 4.20%
nsieve-bits	3.0ms	+/- 0.00%	44.4ms	+/- 3.40%
controlflow	1.3ms	+/- 26.60%	10.6ms	+/- 3.50%
recursive	1.3ms	+/- 26.60%	10.6ms	+/- 3.50%
crypto	7.3ms	+/- 6.60%	41.7ms	+/- 2.40%
md5	3.6ms	+/- 10.30%	20.3ms	+/- 4.40%
sha1	3.7ms	+/- 9.30%	21.4ms	+/- 2.30%
date	11.2ms	+/- 2.70%	15.7ms	+/- 2.20%
format-xparb	11.2ms	+/- 2.70%	15.7ms	+/- 2.20%
math	12.9ms	+/- 1.80%	77.9ms	+/- 1.30%
cordic	3.0ms	+/- 0.00%	36.6ms	+/- 2.30%
partial-sums	7.9ms	+/- 2.90%	21.9ms	+/- 3.90%
spectral-norm	2.0ms	+/- 0.00%	19.4ms	+/- 3.10%
regexp	5.5ms	+/- 6.80%	6.2ms	+/- 7.30%
dna	5.5ms	+/- 6.80%	6.2ms	+/- 7.30%
string	43.8ms	+/- 2.60%	88.5ms	+/- 1.90%
base64	4.2ms	+/- 7.20%	19.2ms	+/- 2.40%
fasta	11.4ms	+/- 3.20%	22.7ms	+/- 3.90%
tagcloud	22.3ms	+/- 5.00%	30.2ms	+/- 2.70%
validate-input	5.9ms	+/- 3.80%	16.4ms	+/- 5.90%
Total	130.4ms	+/- 1.90%	544.3ms	+/- 1.30%

Table 4: SunSpider Firefox Performance Test Results

Test	Firefox Baseline		Firefox with Virtual Values	
	Mean	95% CI	Mean	95% CI
3d	17.0ms	+/- 4.30%	80.9ms	+/- 2.50%
cube	12.6ms	+/- 7.20%	44.2ms	+/- 2.90%
morph	4.5ms	+/- 11.20%	36.7ms	+/- 3.20%
access	13.9ms	+/- 10.40%	139.3ms	+/- 1.40%
binary-trees	3.0ms	+/- 25.10%	10.9ms	+/- 7.80%
fannkuch	5.5ms	+/- 9.20%	83.7ms	+/- 1.60%
nbody	2.8ms	+/- 16.10%	21.8ms	+/- 3.00%
nsieve	2.6ms	+/- 19.20%	22.9ms	+/- 3.40%
bitops	7.7ms	+/- 9.80%	222.4ms	+/- 0.50%
3bit-bits-in-byte	0.8ms	+/- 37.70%	46.3ms	+/- 1.50%
bits-in-byte	1.6ms	+/- 23.10%	53.2ms	+/- 1.70%
bitwise-and	2.2ms	+/- 20.50%	43.7ms	+/- 1.60%
nsieve-bits	3.1ms	+/- 7.30%	79.2ms	+/- 1.20%
controlflow	2.0ms	+/- 16.80%	16.0ms	+/- 20.60%
recursive	2.0ms	+/- 16.80%	16.0ms	+/- 20.60%
crypto	6.7ms	+/- 6.10%	62.0ms	+/- 2.40%
md5	3.7ms	+/- 13.00%	30.6ms	+/- 3.20%
sha1	3.0ms	+/- 11.20%	31.4ms	+/- 2.90%
date	11.1ms	+/- 3.70%	33.2ms	+/- 8.30%
format-xparb	11.1ms	+/- 3.70%	33.2ms	+/- 8.30%
math	10.4ms	+/- 8.10%	88.0ms	+/- 2.30%
cordic	2.2ms	+/- 13.70%	46.6ms	+/- 1.60%
partial-sums	6.6ms	+/- 9.10%	18.6ms	+/- 2.70%
spectral-norm	1.6ms	+/- 23.10%	22.8ms	+/- 4.90%
regexp	6.6ms	+/- 5.60%	7.9ms	+/- 7.90%
dna	6.6ms	+/- 5.60%	7.9ms	+/- 7.90%
string	30.9ms	+/- 3.80%	101.9ms	+/- 2.30%
base64	5.7ms	+/- 8.50%	28.8ms	+/- 4.00%
fasta	6.0ms	+/- 9.70%	25.1ms	+/- 4.30%
tagcloud	13.2ms	+/- 3.40%	30.6ms	+/- 3.20%
validate-input	6.0ms	+/- 9.7	17.4ms	+/- 5.20%
Total	142.3ms	+/- 8.20%	751.6ms	+/- 1.20%

CHAPTER 6

Conclusion

With the ever increasing popularity of web applications and the growing number of security breaches, language level features that ensure integrity and confidentiality of data are essential. We have shown that we can achieve that through the use of Sweet.js, and virtual values while ensuring cross browser compatibility. While the solution in its current form has much higher overhead than what we desire, we believe that using compiler tweaks that we mentioned, we can improve the performance to an acceptable level and improve the security of web applications.

LIST OF REFERENCES

[1] A. Barth, J. Weinberger and D. Song Cross-origin Javascript Capability Leaks: Detection, Exploitation, and Defense, In Proceedings of the 18th Conference on USENIX Security Symposium, 2009, http://dl.acm.org/citation.cfm?id=1855768.1855780

[2] A. Barth, C. Jackson, and W. Li, Attacks on JavaScript Mashup Communication In Proceedings of the Web 2.0 Security and Privacy 2009, 2009

[3] D. Boneh and J, Mitchell CS155: Computer and Network Security - Browser Security Model https://crypto.stanford.edu/cs155/lectures/08-browser-sec-model.pdf

[4] A. Myers, B. Liskov, A Decentralized Model for Information Flow Control, In SIGOPS Oper. Syst. Rev., Pages 129-142, 1997, http://doi.acm.org/10.1145/269005.266669

[5] A. Myers, In Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1999, pages = 228-241, http://doi.acm.org/10.1145/292540.292561

[6] T. H. Austin and C, Flanagan Efficient Purely-Dynamic Information Flow Analysis, In Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security, 2009, http://doi.acm.org/10.1145/1554339.1554353

[7] D. Chandra and M. Franz, Fine-grained information flow analysis and enforcement in a java virtual machine, In Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual, Pages 463--475, Dec. 2007.

[8] T. Disney, N. Faubion, D. Herman and C. Flanagan, Sweeten Your JavaScript: Hygienic Macros for ES5, In Proceedings of the 10th ACM Symposium on Dynamic Languages, 2014, http://doi.acm.org/10.1145/2661088.2661097

[9] T. Austin, T. Disney and C. Flanagan, Virtual Values for Language Extension, In Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2011, http://doi.acm.org/10.1145/2048066.2048136,

[10] L. Meyerovich, A. Felt, A. .Porter and M. Mille, Object Views: Fine-grained Sharing in Browsers, In Proceedings of the 19th International Conference on World Wide Web, pages 721-730, 2010. http://doi.acm.org/10.1145/1772690.1772764

[11] Same Origin Policy, Retrieved on: 04/20/2016, https://www.w3.org/Security/wiki/Same_Origin_Policy

[12] A. Barth and C. Jackson and J. Mitchell, Securing Frame Communication in Browsers, In Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008), 2008.
https://seclab.stanford.edu/websec/frames/post-message.pdf

[13] Frame Navigation Policies, Retrieved on: 04/20/2016, http://novogeek-archive.azurewebsites.net/post/Frame-navigation-policies-in-web-browsers-One-big-reason-why-you-should-get-rid-of-old-browsers

[14] E. Schwartz, T. Avgerinos, and D. Brumley, All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask), In Proceedings of the 2010 IEEE Symposium on Security and Privacy, 2010, http://dx.doi.org/10.1109/SP.2010.26

[15] G. Smith, M. Christodorescu, S. Jha, D. Maughan, D. Song, Dawn and C. Wang, Principles of Secure Information Flow Analysis, In Malware Detection, pages 291-307, 2007, http://dx.doi.org/10.1007/978-0-387-44599-1_13

[16] J.S. Fenton, Memoryless subsystems, The Computer Journal, volume 17, number 2, pages 143-147, 1974, http://comjnl.oxfordjournals.org/content/17/2/143.abstract

[17] Syntactic Macros, Retrieved on 04/20/2016, https://en.wikipedia.org/wiki/Macro_(computer_science)#Syntactic_macros

[18] Macro Programming Systems, Retrieved on 04/20/2016, http://linuxfinances.info/info/macros.html

[19] E. Kohlbecker and M. Wand, Macro-by-example: Deriving Syntactic Transformations from Their Specifications, In Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 1987, http://doi.acm.org/10.1145/41625.41632

[20] J. McCarthy, Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, In Journal Communications of ACM, 1960, url-http://doi.acm.org/10.1145/367177.367199

[21] N. Fitzgerald and R. Nyman, Compiling to JavaScript, and Debugging with Source Maps, 2013, https://hacks.mozilla.org/2013/05/compiling-to-javascript-and-debugging-with-source-maps

[22] Commons JS Specification, Retrieved on 04/20/2016 http://wiki.commonjs.org/wiki/CommonJS

[23] Sun Spider Benchmark, Retrieved on 04/20/2016 https://webkit.org/blog/2364/announcing-sunspider-1-0

APPENDIX

Virtual Values

A.1 Macros

```
operator ++ 15 { $op }     => #{ vvalues.unary("++", $op) }
operator -- 15 { $op }     => #{ vvalues.unary("--", $op) }
operator ! 14 { $op }      => #{ vvalues.unary("!", $op) }
operator ~ 14 { $op }      => #{ vvalues.unary("~", $op) }
operator + 14 { $op }      => #{ vvalues.unary("+", $op) }
operator - 14 { $op }      => #{ vvalues.unary("-", $op) }
operator typeof 14 { $op } => #{ vvalues.unary("typeof", $op) }
operator void 14 { $op }   => #{ vvalues.unary("void", $op) }

operator * 13 left { $left, $right }
                           => #{ vvalues.binary("*", $left, $right) }
operator / 13 left { $left, $right }
                           => #{ vvalues.binary("/", $left, $right) }
operator % 13 left { $left, $right }
                           => #{ vvalues.binary("%", $left, $right) }
operator + 12 left { $left, $right }
                           => #{ vvalues.binary("+", $left, $right) }
operator - 12 left { $left, $right }
                           => #{ vvalues.binary("-", $left, $right) }
operator >> 11 left { $left, $right }
                           => #{ vvalues.binary(">>", $left, $right) }
operator << 11 left { $left, $right }
                           => #{ vvalues.binary("<<", $left, $right) }
operator >>> 11 left { $left, $right }
                           => #{ vvalues.binary(">>>", $left, $right) }
operator < 10 left { $left, $right }
                           => #{ vvalues.binary("<", $left, $right) }
operator <= 10 left { $left, $right }
                           => #{ vvalues.binary("<=", $left, $right) }
operator >= 10 left { $left, $right }
                           => #{ vvalues.binary(">", $left, $right) }
operator > 10 left { $left, $right }
                           => #{ vvalues.binary(">=", $left, $right) }
operator in 10 left { $left, $right }
                           => #{ vvalues.binary("in", $left, $right) }
```

```
operator instanceof 10 left { $left, $right }
                           => #{ vvalues.binary("instanceof", $left, $right) }
operator == 9 left { $left, $right }
                           => #{ vvalues.binary("==", $left, $right) }
operator != 9 left { $left, $right }
                           => #{ vvalues.binary("!=", $left, $right) }
operator === 9 left { $left, $right }
                           => #{ vvalues.binary("===", $left, $right) }
operator !== 9 left { $left, $right }
                           => #{ vvalues.binary("!==", $left, $right) }
operator & 8 left { $left, $right }
                           => #{ vvalues.binary("&", $left, $right) }
operator ^ 7 left { $left, $right }
                           => #{ vvalues.binary("^", $left, $right) }
operator | 6 left { $left, $right }
                           => #{ vvalues.binary("|", $left, $right) }
operator && 5 left { $left, $right }
                           => #{ vvalues.binary("&&", $left, $right) }
operator || 4 left { $left, $right }
                           => #{ vvalues.binary("||", $left, $right) }
```

A.2 Harness

```
function unary(a0, a1) {
    if (isVProxy(a1)) {
        var operator = a0;
        var op = a1;
        var target = unproxyMap.get(op).target;
        return unproxyMap.get(op).handler.unary(target, operator, op);
    }
    if (a0 === '-') {
        var op = a1;
        return -op;
    }
    if (a0 === '+') {
        var op = a1;
        return +op;
    }
    if (a0 === '++') {
        var op = a1;
        return ++op;
    }
```

```
    if (a0 === '--') {
        var op = a1;
        return --op;
    }
    if (a0 === '!') {
        var op = a1;
        return !op;
    }
    if (a0 === '~') {
        var op = a1;
        return ~op;
    }
    if (a0 === 'typeof') {
        var op = a1;
        return typeof op;
    }
    if (a0 === 'void') {
        var op = a1;
        return void op;
    }
    throw new TypeError('No match');
}
// @ (Str, Any, Any) -> Any
function binary(a0, a1, a2) {
    if (isVProxy(a1)) {
        var operator = a0;
        var left = a1;
        var right = a2;
        var target = unproxyMap.get(left).target;
        return unproxyMap.get(left).handler.left(target, operator, right);
    }
    if (isVProxy(a2)) {
        var operator = a0;
        var left = a1;
        var right = a2;
        var target = unproxyMap.get(right).target;
        return unproxyMap.get(right).handler.right(target, operator, left);
    }
    if (a0 === '*') {
        var left = a1;
        var right = a2;
        return left * right;
```

```
}
if (a0 === '/') {
    var left = a1;
    var right = a2;
    return left / right;
}
if (a0 === '%') {
    var left = a1;
    var right = a2;
    return left % right;
}
if (a0 === '+') {
    var left = a1;
    var right = a2;
    return left + right;
}
if (a0 === '-') {
    var left = a1;
    var right = a2;
    return left - right;
}
if (a0 === '>>') {
    var left = a1;
    var right = a2;
    return left >> right;
}
if (a0 === '<<') {
    var left = a1;
    var right = a2;
    return left << right;
}
if (a0 === '>>>') {
    var left = a1;
    var right = a2;
    return left >>> right;
}
if (a0 === '<') {
    var left = a1;
    var right = a2;
    return left < right;
}
if (a0 === '<=') {
```

```
        var left = a1;
        var right = a2;
        return left <= right;
    }
    if (a0 === '>') {
        var left = a1;
        var right = a2;
        return left > right;
    }
    if (a0 === '>=') {
        var left = a1;
        var right = a2;
        return left >= right;
    }
    if (a0 === 'in') {
        var left = a1;
        var right = a2;
        return left in right;
    }
    if (a0 === 'instanceof') {
        var left = a1;
        var right = a2;
        return left instanceof right;
    }
    if (a0 === '==') {
        var left = a1;
        var right = a2;
        return left == right;
    }
    if (a0 === '!=') {
        var left = a1;
        var right = a2;
        return left != right;
    }
    if (a0 === '===') {
        var left = a1;
        var right = a2;
        return left === right;
    }
    if (a0 === '!==') {
        var left = a1;
        var right = a2;
```

```
        return left !== right;
    }
    if (a0 === '&') {
        var left = a1;
        var right = a2;
        return left & right;
    }
    if (a0 === '^') {
        var left = a1;
        var right = a2;
        return left ^ right;
    }
    if (a0 === '|') {
        var left = a1;
        var right = a2;
        return left | right;
    }
    if (a0 === '&&') {
        var left = a1;
        var right = a2;
        return left && right;
    }
    if (a0 === '||') {
        var left = a1;
        var right = a2;
        return left || right;
    }
    throw new TypeError('No match');
}
```

A.3 Taint Functions

```
function taint(originalValue) {
    // don't need to taint and already tainted value
    if (isTainted(originalValue)) {
        return originalValue;
    }

    var p = new Proxy(originalValue, {
        // store the original untainted value for later
        originalValue: originalValue,
```

```
        unary: function (target, op, operand) {
            return taint(unaryOps[op](target));
        },

        left: function (target, op, right) {
            return taint(binaryOps[op](target, right));
        },

        right: function (target, op, left) {
            return taint(binaryOps[op](left, target));
        }
    }, taintingKey);
    return p;
}

function isTainted (x) {
    // a value is tainted if it’s a proxy created with the ‘taintingKey‘
    if (unproxy(x, taintingKey)) {
        return true;
    }
    return false;
}

function untaint (value) {
    if (isTainted(value)) {
        // pulls the value out of its tainting proxy
        return unproxy(value, taintingKey).originalValue;
    }
    return value;
}
```