Using Precise Taint Tracking for Auto-sanitization

Tejas Saoji
Computer Science
San José State University
tejaspankaj.saoji@sjsu.edu

Thomas H. Austin
Computer Science
San José State University
thomas.austin@sjsu.edu

Cormac Flanagan
Computer Science
University of California, Santa Cruz
cormac@ucsc.edu

ABSTRACT

Taint analysis has been used in numerous scripting languages such as Perl and Ruby to defend against various form of code injection attacks, such as cross-site scripting (XSS) and SQL-injection. However, most taint analysis systems simply fail when tainted information is used in a possibly unsafe manner.

In this paper, we explore how *precise taint tracking* can be used in order to secure web content. Rather than simply crashing, we propose that a library-writer defined sanitization function can instead be used on the tainted portions of a string. With this approach, library writers or framework developers can design their tools to be resilient, even if inexperienced developers misuse these libraries in unsafe ways. In other words, developer mistakes do not have to result in system crashes to guarantee security.

We implement both coarse-grained and precise taint tracking in JavaScript, and show how our precise taint tracking API can be used to defend against SQL injection and XSS attacks. We further evaluate the performance of this approach, showing that precise taint tracking involves an overhead of approximately 22%.

CCS CONCEPTS

• **Security and privacy → Web application security**; *Browser security*; *Information flow control*;

KEYWORDS

taint analysis, JavaScript, web application security

1 INTRODUCTION

Code injection attacks are a constant threat to web applications. These attacks occur because input from an untrusted source is used in situations where it might be able to create code. SQL injection is one well known variant of code injection. Consider the following code that executes a query against a database:

```
execQuery("SELECT * FROM STUDENTS " +
  "WHERE NAME = '" + studentName + "'");
```

While this code will work in most cases, an attacker could exploit it by entering a carefully formatted name, such as

```
Bobby'; DROP TABLE STUDENTS;--
```

With `name` set to the above text, the query executed against the database would be

```
SELECT * FROM STUDENTS
WHERE NAME = 'Bobby';
DROP TABLE STUDENTS;--'
```

Library writers and framework designers are aware of these vulnerabilities, and provide their users tools to safely include user input. For instance, a SQL library might include an `execQueryParams` function that would take an array of arguments for user input. The library would properly escape the input before using it to build the SQL query. Since the library writers are domain experts, the possibility of a code injection attack is much reduced. Refactoring the previous example would then give us the following, safe code:

```
execQueryParams("SELECT * FROM STUDENTS"
    + " WHERE NAME = ?", [studentName]);
```

However, there is nothing that forces the programmer to use the library correctly. Inexperienced developers might use the function in the same manner as they did before and produce vulnerable code, even thought the library was designed to prevent these vulnerabilities.

```
execQueryParams("SELECT * FROM STUDENTS"
    + " WHERE NAME = '"
    + studentName + "'", []);
```

Taint analysis helps to catch cases of programmer error by identifying places where input from an untrusted source has been used unsafely. This technique can be done either using static analysis or dynamic runtime monitoring, often called *taint tracking*. Leveraging this tool, a SQL library writer might write the following function to catch developer error:

```
function execQueryParams(query,params){
  if (isTainted(query))
    throw new Error("Tainted query");
  // SQL handling code below
  ...
}
```

Despite having been a part of mainstream languages such as Perl and Ruby, taint analysis does not appear to be that widely used. One issue seems to be that taint analysis has too many false positives.

Nguyen-Tuong et al. [29] add an interesting addition to this design space with *precise tainting*. With this approach, tainted strings are tracked at the level of individual characters. This design reduces false positives and allows developers to more precisely identify tainted information.

PLAS'17, October 30, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-5099-0/17/10…$15.00
https://doi.org/10.1145/3139337.3139341

More importantly, this strategy can offer the possibility of more sophisticated strategies than simply accepting or rejecting an operation on a tainted string. The original authors take advantage of this finer granularity to identify bad characters in HTML output, but do not sanitize input to stop other common web attacks.

We extend precise taint tracking to give authors the ability to *auto-sanitize* user input when their libraries are misused by inexperienced developers. With this approach, the SQL library of our motivating example could instead warn the developers that they were not using the library correctly and then escape just the tainted portions of the SQL query string. Our taint tracking API uses a novel `taintedRegions` function that library writers can use to sanitize calls to their library. We implement precise taint tracking for JavaScript, and compare its performance to both an uninstrumented JavaScript interpreter and to a JavaScript interpreter with coarse-grained taint-tracking. Our results show that our precise taint tracking interpreter has an overhead of roughly 22% when no tainted data is involved.

2 RELATED WORK

Our principal inspiration is Nguyen-Tuong et al.'s design for precise taint tracking [29]. By tracking information for tainted data at the character level, the authors are able to reduce the false positives compared to coarse-grained taint tracking approaches. They also show how their approach can sanitize data to prevent against output attacks, but do not extend this idea to sanitize SQL queries.

Our work extends their ideas to the language rather than the framework level, and therefore permits library and framework designers to apply these techniques to other forms of code injection attacks. By choosing JavaScript, we also offer the ability to defend against both client and server attacks.

Other research has leveraged precise taint tracking to either avoid false positives or to take more sophisticated strategies. Su and Wassermann [39] use precise taint tracking to detect SQL injection attacks in their SqlCheck tool. Stock et al [38] integrate precise taint tracking into Chromium with a goal of detecting XSS vulnerabilities. DexterJS by Parameshwaran et al. [30] follows a similar goal, but uses a JavaScript transpiler to introduce dynamic taint tracking, thereby avoiding any modifications to the browser implementation. Closest to our work, Pietraszek and Vanden Berghe [31] use precise tainting in their context-sensitive string evaluation (CSSE) implementation for PHP; as with our approach, they enable automatic correction of many injection errors. Their work includes a rich discussion of different types of injection vulnerabilities, but does not offer much discussion on their API.

Schoepe et al. [34] offer an alternative design for taint tracking in the realm of Android apps. They use special *facelifted values*, related to secure multi-execution [16] and faceted evaluation [3, 8], in order to track both trusted and untrusted views of sensitive fields. This approach gives them greater precision, and similarly allows code to self-repair. Guha et al. [18] use a similar approach with their work on Fission to run JavaScript programs on both the client and server.

Tsankov et al. [43] use a similar fine-grained approach in their Functionality-Aware Security Environment (FASE), though they use this technique to anonymize data for Android apps. Their work considers confidentiality as well as integrity, which brings some additional challenges as highlighted in the information flow analysis discussion later in this section.

Taint analysis has previously been explored for JavaScript. ACTARUS by Guarnieri et al. [17] is a static taint analysis tool that the authors use to defend against XSS attacks, code injection, and unvalidated redirects. Their analysis is sound, with the exception of reflective calls, and they test their tool on an impressive 9,726 web pages. Wei and Ryder [45] use a mix of static analysis and dynamic taint tracking, which they call *blended taint analysis*. They argue that static taint analysis alone is a difficult fit for JavaScript, due to the language's many dynamic features. Wei et al. [46] analyze features within JavaScript that make static analysis difficult, which they dub *root-cause functions*. Tripp et al. [41] develop JSA, which uses code rewriting and partial evaluation; their approach greatly reduces false positives compared to purely static analysis, while still capturing all true warnings.

Other branches of research have focused on creating general purpose analysis frameworks for JavaScript, such as Sen et al's Jalangi [35] and Christophe et al's Linvail [11]. Similarly, Kannan et al. [25] show how *virtual values* [1], related to JavaScript proxies [15], can be used to integrate both taint tracking and information flow monitoring into applications without native support in the language for those security features. Guha et al. [19] provide a static analysis tool for Ajax programs; their focus is on intrusion detection, but they note that there are several other possible applications of their tool.

Information flow analysis is a closely related technique to taint analysis. The central difference is that information flow analysis focuses more on confidentiality rather than integrity, and as a result must consider *implicit flows* carefully, where secret values may be deduced by reasoning about public outputs. Bielova and Rezk [9] provide an excellent overview of the different types of information flow monitors. JavaScript in particular has been a popular domain for dynamic information flow analysis research, beginning with Vogt et al. [44] using information flow analysis to detect cross-site scripting attacks. Jang et al. [24] use information flow analysis to survey real-world examples of websites attempting to circumvent privacy protections. Chudnov and Naumann [12] integrate a dynamic information flow monitor by rewriting JavaScript. Bauer et al. [4] show how a dynamic information flow monitor can prevent many common attacks in the Chromium web browser. Kerschbaumer et al. [26] show how to create an efficient information flow monitor for JavaScript. Bicchawat et al [6] implement an information flow monitor at the bytecode level using the permissive-upgrade strategy [2, 5]. The same author also propose WebPol [7], an API for fine-grained information flow policies. JSFflow [20–22] is a JavaScript interpreter designed to track information flow; it is implemented as a Firefox extension, allowing it to be integrated into the web browser without modifying the underlying JS engine. Hedin et al. [23] use *stateful marshalling* to track information flow through third-party libraries. Stefan et al. [37] develop COWL, another system intended to prevent the exfiltration of private data in a web browser. Chugh et al. [13] use a mix of static and dynamic analysis in their staged information flow analysis; in their approach, they statically analyze the JavaScript code to guarantee as much as

possible, with additional runtime *residual checks* for properties that cannot be guaranteed in advance.

While research on scripting languages has focused on dynamic taint tracking, research for statically typed languages prefers static taint analysis. Tripp et al. [42] develop Taint Analysis for Java (TAJ). Their approach statically analyzes flows from sources to sinks in order to prevent common attacks against web applications. Livshits and Lam [27] integrate a static analysis tool into Eclipse, allowing developers to easily check for many common security vulnerabilities. Sridharan et al [36] develop taint analysis for Java web application frameworks, with a focus on correctly handling reflective calls within these frameworks.

C programs are another popular language for studying with taint tracking. Newsome and Song [28] have implemented TaintCheck, to detect format string vulnerabilities and buffer overruns. Xu et al [47] explore source-to-source transformations of C programs, introducing taint analysis to enforce their security policies. Chang et al. [10] extend this approach as more of a general tool, using taint tracking as their use case. Their transformation tool performs static analysis on the program to distinguish between innocuous data flows and possible leaks. If their system finds possible violations of the security policies, an analysis is performed to identify all the positions in the code that demand dynamic analysis. The code is modified accordingly to enforce those policies dynamically. Thus, the amount of dynamic tracking needed is minimized by doing static analysis first. Clause et al. [14] develop Dytan, a framework for securing x86 executables. Their tool uses dynamic taint tracking, but can also provide information flow guarantees through the use of control flow graphs.

3 COARSE-GRAINED TAINT TRACKING

Coarse-grained taint tracking is the dominant approach used in scripting languages today. It tracks when a value comes from an untrusted external source or is in any way derived from untrusted values. Since most of the real-world cases of interest in code injection involve strings, we focus our discussion on strings specifically.

We modified the Rhino implementation of JavaScript, which is an open-source JavaScript engine written in Java and managed by the Mozilla Foundation [32]. We selected Rhino since it is a long-established JavaScript engine, and since its codebase is comparatively easy to work with. We used a *string mangling* strategy for our library, where we append a special token to the end of the internal string representations to mark them as tainted; this approach might lead to inadvertent tainting of values, but these cases can be kept to a minimum if an unusual enough token is chosen.

Our coarse-grained taint tracking API consists of three functions:

(1) `taint(input)` returns a copy of the `input` string marked as tainted. The original `input` string remains untainted.
(2) `isTainted(str)` allows a developer to examine a string to determine whether the `str` value is in any way derived from untrusted sources.
(3) `untaint(tainted)` takes the `tainted` string and returns a copy of the string without the taint mark. After a developer has carefully validated and modified the input string, this function permits untainted data to be *endorsed*.

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

Figure 1: String tainting

```
var name = taint("O'Reilly");
var str = "Hi, my name is " + name;
var substr = str.substring(0,2);
isTainted(substr);    // returns true
```

Figure 2: Coarse-grained Taint Tracking False Positive

Figure 1 shows how coarse-grained taint analysis taints and endorses a string variable. In Figure 1 `taint(a)` returns a tainted string "O'Reilly" and assigns it to variable `b`. `isTainted(b)` returns true as the string `b` is referring to is marked as tainted. `b.replace(/'/g, "''")` sanitizes the tainted string in variable `b` by replacing a single quote with two single quotes, and `untaint` marks the sanitized string as untainted and returns it, which is assigned to variable `c`. Hence, `isTainted(c)` returns false.

All operations on strings are modified to taint the resulting value if any of the inputs are tainted. In some cases, the result might be tainted even if the original untrusted information is no longer part of the result. Figure 2 shows an example. Even though `substr` contains no tainted information from the `name` variable, the result is marked as tainted because it was derived from the tainted variable `str`.

Since coarse-grained taint analysis only tracks whether a string is tainted, and not which portions of the string are tainted, it is difficult to sanitize the tainted string, even if only a small portion of the string is derived from tainted data. Hence, in the case of coarse-grained taint analysis the best way to prevent an attack is to raise an exception whenever a tainted string is passed to a security critical operation. Once developers are made aware of their misuse of the library, they can update their code to use the library properly[1].

4 PRECISE TAINT TRACKING API

While coarse-grained taint tracking is useful for preventing unsafe operations, it suffers from false positives and inflexibility. Instead, we can track the flow of tainted data at a finer granularity. Precise taint tracking [29] tracks taint at the level of characters. With this approach, library writers may more gracefully recover when their library is misused.

Our precise taint tracking API extends the coarse-grained taint tracking API with two new functions:

[1] Of course, there is nothing that forces the developer to fix their code in the correct manner. It is entirely possible that willful developers might just endorse the inputs to the library with no attempt at validation in order to quickly get their code working.

```
var a = "Hi " + taint("Jane");
isTainted(a); // true
isTainted(a.substring(0,3)); // false
isTainted(a.substring(3,7)); // true
taintedRegions(a);  // [[3,7]]
var b = sanitize(a, function(s) {
    return s.replace(/'/g,"''");
})
isTainted(b);    // false
```

Figure 3: String tainting

- The `taintedRegions(taintedStr)` function returns an array of pairs[2] specifying the left and right boundaries of tainted regions in `taintedStr`.
- The `sanitize(taintedStr, f)` function allows a tainted string to be repaired by applying the function `f` to each tainted region of the string.

The other functions of the API remain, with some slight modifications. The `isTainted` function returns `true` if any portion of the string is tainted. The `untaint` function removes taints from any portion of the tainted string. As with our coarse-grained taint tracking implementation, we use a string mangling approach to track tainted data. However, in the case of precise taint tracking, we also append a comma separated list of the ranges of the tainted regions within the String.

Figure 3 shows an example using precise taint analysis. The variable `a` contains both tainted and untainted data, and so `isTainted(a)` returns `true`. However, in contrast to coarse-grained taint tracking, precise taint tracking recognizes that the substring `"Hi "` is not tainted.

Our precise taint tracking API allows us to sanitize strings that contain a mix of tainted and untainted characters. The `taintedRegions` function returns `[[3,7]]`, indicating the start and end positions of the tainted substring in `a`. Using this function, the library writer could carefully sanitize the portions of the string that came from untrusted sources with this information. However, in most cases we expect that the data could be sanitized by applying a sanitization function to each tainted substring and joining it with the untainted sections of the string. In this case, the `sanitize` function can be used instead. In the case of Figure 3, we escape single quotes with pairs of single quotes, such as we might do for a SQL query.

5 APPLICATIONS OF PRECISE TAINT TRACKING

In this section, we show how precise taint tracking can be used to auto-sanitize input when a library is misused. Initially we focus on the `taintedRegions` function before showing how `sanitize` can simplify the code for most use cases.

5.1 Preventing SQL Injection Attack

In our introduction, we discussed how coarse taint tracking could be used to prevent an inexperienced developer from misusing a SQL library in a way that might cause a SQL injection vulnerability. We now consider the same example using precise taint analysis.

Figure 4 shows how we could use the `taintedRegions` function to identify tainted regions of the `query` string and sanitize those portions of the string. For simplicity, we elide the behavior of `runQuery`, which executes the query after sanitizing the arguments in `params` and building the final SQL query. We assume that the params argument is either missing or empty if `query` is tainted, which seems the likely case if the library is misused. If that is not the case, we raise an error; we note that it is still possible to sanitize the query, but at the expense of increased complexity.

The `untaintQuery` function sanitizes a tainted portion of the `query` string by replacing single quotes with two single quotes, and returns the untainted substring. While this sanitization function is simple, it illustrates the concept of how the query string could be sanitized.

At the end of the `for` loop in the `execQueryParams` function, the variable `untaintedQuery` contains an untainted, sanitized query. Using our example from the introduction, the final query would be

```
SELECT * FROM STUDENTS
WHERE NAME = 'Bobby'';
DROP TABLE STUDENTS:--'
```

Critically, the quote after "Bobby" is escaped, and so the untainted query can be safely executed. Thus, precise taint tracking prevents the system from a SQL injection attack without having to crash the program execution.

5.2 Preventing Cross-Site Scripting

We now show how this approach can be used to defend against cross-site scripting (XSS) attacks. Consider the web-page in Figure 5, which uses JavaScript to greet the user by name. The function `window.location.search` returns the contents of the page URL after the question mark. The dynamically generated content is then written to the document using `document.write()`. The URL used to access the page is:

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

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

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

In the case of precise taint analysis, all the characters in the string in the variable `name` are marked as tainted. The `taintedRegions` function is used to sanitize the tainted string. Figure 6 shows how precise taint tracking handles the tainted string. We note that while this strategy escapes special characters with their equivalent HTML entities, other options are possible; for instance, a library writer might wish to support some markup formatting, but not allow script tags. One advantage of our approach is that library writers can tailor their handling of tainted data to their needs.

[2] In our implementation, a pair is represented by an array of size 2.

```
function execQueryParams(query, params) {
  if(isTainted(query)){
    if (params && params.length !== 0) {
      throw new Error("Tainted query and non-empty params");
    }
    let untaintedQuery = "";
    let idxInQuery = 0;
    let taintedregions = taintedRegions(query);
    for (i = 0; i < taintedregions.length; i++) {
      let taintLBound = taintedregions[i][0];
      let taintRBound = taintedregions[i][1];

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

    runQuery(untaintedQuery);
  } else {
    runQuery(query, params);
  }
}

function untaintQuery(query,idxInQuery,taintLBound,taintRBound){
  let untaintedStr = query.substring(indexInQuery,taintLBound);

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

  return untaintedStr + sanitizedStr;
}
```

Figure 4: Untainting a query string

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

Figure 5: Sample web-page

5.3 Sanitize Function

Contrasting Figure 4 with Figure 6, we see that the code to sanitize a string to prevent a cross-site scripting attack is very similar to the code to sanitize the query string in our SQL library. The major difference is the actual method of sanitization. In case of SQL injection it is done as

```
replace(/'/g, "''")
```

while in the case of cross-site scripting it is done as

```
replace(/</g, "\&lt;").replace(/>/g, "\&gt;")
```

We can avoid this redundancy by using the `sanitize` function in lieu of `taintedRegions`. The `sanitize` function takes 2 arguments — a tainted string and a callback function to sanitize the tainted portions of this string. Figure 7 shows the `sanitize` function.

Figure 8 shows how the `sanitize` function can be used to sanitize a tainted query in the case of a SQL injection attack, and a tainted string in the case of a cross-site scripting attack. In the case of a SQL injection attack, the callback function replaces all the single quotes with two single quotes. The `sanitize` function returns a sanitized and untainted query string, which is safe to use. The untainted query string in the variable `untaintedQuery` could then be executed safely.

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

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

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

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

function untaintString(name,idxInStr,taintLBound,taintRBound) {
  let untaintedStr = name.substring(idxInStr,taintLBound);

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

  return untaintedStr + sanitizedStr;
}
```

Figure 6: Untainting a user input string

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

    let untaintedStr = str.substring(idxInStr,taintLBound);

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

Figure 7: Sanitize function

5.4 Context-sensitive Auto-sanitization

Our previous defense against SQL injection assumes that tainted data will only appear in a SQL string, but other forms of SQL injection may occur. Consider the following code:

```
execQueryParams("SELECT * FROM STUDENTS"
    + " WHERE ID = " + studentID, []);
```

```
//Example - SQL injection
function execQueryParams(query,params) {
  let untaintedQuery = "";
  if(isTainted(query)){
    if (params && params.length !== 0) {
      throw new Error("Tainted query");
    }
    untaintedQuery = sanitize(query,
        function(s){
          return s.replace(/'/g, "''");
        });
    runQuery(untaintedQuery);
  } else {
    runQuery(query);
  }
}

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

Figure 8: Sanitize function call

If the attacker is able to change `studentID` to `"666 OR 1=1"`, then the query would instead return all student records. This code highlights how the appropriate sanitization operation might depend on the context of the tainted data. In this case, the library writer might wish to auto-sanitize tainted strings, but to disallow tainted data outside of a string context.

The `sanitize` function in our API does not support context-sensitive sanitization, but we can achieve this result by using the `taintedRegions` function. A simple approach might count the total number of quotes, and throw an exception if attempting to evaluate a tainted region of the string where there is an even number of quotes preceding it in untainted regions. More sophisticated approaches might involve a SQL tokenizer, similar to the approach used by Stock et al. [38].

5.5 When Auto-sanitization Fails: Eval Injection

Precise taint tracking enables auto-sanitization for many cases, but there are some domains where it does not appear to be useful. Defending against `eval` injection is one such case.

Many scripting languages support dynamic code evaluation, and it is an especially widely used feature in JavaScript, often in cases where using `eval` has no obvious benefit [33]. Unfortunately, this

```
let oldEval = eval;
eval = function(s) {
  if (isTainted(s))
    throw new Error("Trying to eval a
        tainted string");
  return oldEval(s);
}
```

Figure 9: "Safe" eval

feature can allow an attacker to inject code into a system. Consider the following code, which uses eval to parse JSON formatted strings:

```
function parseJSON(s) {
  eval("var o =" + s);
  return o;
}
```

Assume that `withdrawalAmount` and `accountNum` are two strings coming from an untrusted source. If the strings contain numerical data, the following code will return a well-formed JavaScript object:

```
var o = parseJSON("{" +
  "withdraw:" + withdrawalAmount + "," +
  "acc:" + accountNum +
  "}");
```

However, if the string assigned to one of these fields is instead

```
"(function() { console.log('I hak you!') })()"
```

then this arbitrary function will be invoked on the system.

Taint tracking can be used to stop these attacks. Figure 9 shows a version of `eval` that raises an error rather than evaluating a tainted string.

Unfortunately, there is no obvious way to sanitize the input. Auto-sanitization seeks to prevent strings from being evaluated as code, but that is precisely the function of `eval`. While our precise taint tracking API can prevent these attacks, it offers no benefit over coarse-grained taint tracking in this case.

6 PERFORMANCE TESTS

In order to understand the performance overhead of precise taint tracking, we evaluated both our coarse-grained and precise taint tracking implementations against the SunSpider benchmark suite [40] and compared our results against the baseline, unmodified version of Rhino. SunSpider contains tests that are balanced between different areas of the language and different types of code. These tests were performed on a MacBook Pro with a 2.7 GHz dual-core Intel Core i5 processor, 8 GB 1867 MHz DDR3 RAM, and an Intel Iris Graphics 6100 graphics processor with 1536 MB of memory. Our code base used Rhino version 1.7.8, and we ran all three versions in interpreted mode. Five tests from the test suite were removed due to unresolved errors in the modified variants of Rhino.

In our first experiment, we ran all three variants of Rhino against SunSpider with no tainted data. This test gives us a comparison of the performance overhead caused by the implementation of taint

```
// Test 0 -- no values tainted
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
```

(a) Test 0

```
// Test 1 -- 1 of 10 values tainted
ret = fastaRepeat(2*count*100000,
    taint(ALU));
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
ret = fastaRepeat(2*count*100000, ALU);
```

(b) Test 1

Figure 10: Incremental taint introduction

analysis in the two modified variants of Rhino against the unmodified version of Rhino. As shown in Table 1, coarse-grained taint analysis adds an overhead of approximately 4% over the unmodified version of Rhino, while the fine-grained taint analysis version adds an overhead of 22%.

For our second experiment, we wished to see the overhead of precise taint tracking when significant amounts of tainted data were involved. We modified the *fasta* test to introduce tainted variables in an incremental fashion. Each case performs 10 hashes of strings. In Test0 (Figure 10a) none of the inputs are marked as tainted, in Test1 (Figure 10b) the first input is tainted, in Test2, the first two inputs are marked as tainted, and so on until Test10 with all inputs marked as tainted.

Table 2 shows the execution time of the tests on both versions of our JavaScript interpreter with support for tainting. Our results show that there is some modest slowdown as the amount of tainted data increases. Further, we see that there is some slight overhead for the precise taint tracking approach, ranging from 1-20% overhead.

7 CONCLUSION

Taint tracking provides a powerful mechanism to prevent code injection attacks. While coarse-grained taint tracking is useful for identifying library misuse, we believe that precise taint tracking offers a useful alternative in the design space. By giving library writers the ability to auto-sanitize corrupted inputs, precise taint tracking offers an alternative to crashing code at runtime.

We have shown how precise taint tracking can be implemented in JavaScript through the use of a `taintedRegions` and a `sanitize` function. It is our hope that this approach will allow library and framework developers the ability to create more robust, fault-tolerant code.

One limitation for our API is that it does not consider cases where "double sanitization" is required. For instance, a tainted string might need to be sanitized to be used in a SQL query, and sanitized again if that entry would be used in a webpage. For future work, we intend to explore how our API could be adapted to these cases. We also plan to explore ways that precise tainting could be optimized to reduce its overhead.

REFERENCES

[1] Thomas H. Austin, Tim Disney, and Cormac Flanagan. 2011. Virtual values for language extension. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 921–938.

[2] Thomas H. Austin and Cormac Flanagan. 2010. Permissive dynamic information flow analysis. In *Programming Languages and Analysis for Security*. ACM, 1–12.

[3] Thomas H. Austin and Cormac Flanagan. 2012. Multiple facets for dynamic information flow. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 165–178.

[4] Lujo Bauer, Shaoying Cai, Limin Jia, Timothy Passaro, Michael Stroucken, and Yuan Tian. 2015. Run-time Monitoring and Formal Analysis of Information Flows in Chromium. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[5] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Generalizing Permissive-Upgrade in Dynamic Information Flow Analysis. In *Programming Languages and Analysis for Security*. ACM.

[6] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Information Flow Control in WebKit's JavaScript Bytecode. In *Principles of Security and Trust (POST)*. Springer, 159–178.

[7] Abhishek Bichhawat, Vineet Rajani, Jinank Jain, Deepak Garg, and Christian Hammer. 2017. WebPol: Fine-grained Information Flow Policies for Web Browsers. (2017).

[8] Nataliia Bielova and Tamara Rezk. 2016. Spot the Difference: Secure Multi-execution and Multiple Facets. In *European Symposium on Research (ESORICS)*. Springer, 501–519. https://doi.org/10.1007/978-3-319-45744-4_25

[9] Nataliia Bielova and Tamara Rezk. 2016. A Taxonomy of Information Flow Monitors. In *Principles of Security and Trust (POST)*. Springer.

[10] Walter Chang, Brandon Streiff, and Calvin Lin. 2008. Efficient and extensible security enforcement using dynamic data flow analysis. In *Conference on Computer and Communications Security (CCS)*. ACM, 39–50.

[11] Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, and Coen De Roover. 2016. Linvail: A General-Purpose Platform for Shadow Execution of JavaScript. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE Computer Society, 260–270.

[12] Andrey Chudnov and David A. Naumann. 2015. Inlined Information Flow Monitoring for JavaScript. In *Conference on Computer and Communications Security (CCS)*. ACM, 629–643. http://doi.acm.org/10.1145/2810103.2813684

[13] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged information flow for JavaScript. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[14] James A. Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *International Symposium on Software Testing and Analysis, ISSTA*. ACM, 196–206.

[15] Tom Van Cutsem and Mark S. Miller. 2010. Proxies: Design Principles for Robust Object-oriented Intercession APIs. In *Dynamic Languages Symposium (DLS)*. ACM.

[16] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *Symposium on Security and Privacy*. IEEE, Los Alamitos, CA, USA, 109–124.

[17] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the world wide web from vulnerable JavaScript. In *International Symposium on Software Testing and Analysis, ISSTA*. ACM, 177–187. https://doi.org/10.1145/2001420.2001442

[18] Arjun Guha, Jean-Baptiste Jeannin, Rachit Nigam, Jane Tangen, and Rian Shambaugh. 2017. Fission: Secure Dynamic Code-Splitting for JavaScript. In *Summit on Advances in Programming Languages (SNAPL)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 5:1–5:13. https://doi.org/10.4230/LIPIcs.SNAPL.2017.5

Table 1: Performance Test Results With No Tainted Variables

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

Table 2: Taint Performance Test Results

Tainted variables	Coarse-Grained		Precise	
	Mean	95% CI	Mean	95% CI
0	471.6ms	±62.3%	485.4ms	±97.3%
1	466.7ms	±162.9%	529.0ms	±72.8%
2	481.1ms	±66.4%	524.4ms	±119.8%
3	476.1ms	±89.6%	517.7ms	±77.4%
4	468.7ms	±104.9%	524.6ms	±88.8%
5	489.4ms	±79.0%	557.1ms	±93.2%
6	494.4ms	±95.5%	579.0ms	±78.3%
7	487.1ms	±244.6%	548.7ms	±303.9%
8	521.4ms	±67.4%	553.7ms	±85.6%
9	478.9ms	±192.2%	572.3ms	±227.2%
10	592.1ms	±48.0%	597.7ms	±78.3%

[19] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. 2009. Using static analysis for Ajax intrusion detection. In *Web 2.0 Security & Privacy 2012*. 561–570. https://doi.org/10.1145/1526709.1526785

[20] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: tracking information flow in JavaScript and its APIs. In *Symposium on Applied Computing (SAC)*. ACM, 1663–1671. https://doi.org/10.1145/2554850.2554909

[21] Daniel Hedin and Andrei Sabelfeld. 2012. Information-flow security for a core of JavaScript. In *Computer Security Foundations Symposium (CSF)*. IEEE.

[22] Daniel Hedin and Andrei Sabelfeld. 2015. Web Application Security Using JSFlow. In *17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2015, Timisoara, Romania, September 21-24, 2015*. IEEE, 16–19.

[23] Daniel Hedin, Alexander Sjösten, Frank Piessens, and Andrei Sabelfeld. 2017. A Principled Approach to Tracking Information Flow in the Presence of Libraries. In *Principles of Security and Trust (POST)*. Springer, 49–70. https://doi.org/10.1007/978-3-662-54455-6_3

[24] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An empirical study of privacy-violating information flows in JavaScript web applications. In *Computer & Communications Security*. ACM.

[25] Prakasam Kannan, Thomas H. Austin, Mark Stamp, Tim Disney, and Cormac Flanagan. 2016. Virtual Values for Taint and Information Flow Analysis. In *Workshop on Meta-Programming Techniques and Reflection, META*. ACM.

[26] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Towards Precise and Efficient Information Flow Control in Web Browsers. In *Trust and Trustworthy Computing Conference*. Springer.

[27] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*. USENIX Association.

[28] James Newsome and Dawn Xiaodong Song. 2005. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Network and Distributed System Security Symposium (NDSS)*. The Internet Society.

[29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. 2005. Automatically hardening web applications using precise tainting. In *IFIP International Information Security Conference*. Springer, 295–307.

[30] Inian Parameshwaran, Enrico Budianto, Shweta Shinde, Hung Dang, Atul Sadhu, and Prateek Saxena. 2015. DexterJS: robust testing platform for DOM-based XSS vulnerabilities. In *Special Interest Group on Software Engineering (SIGSOFT)*. 946–949. https://doi.org/10.1145/2786805.2803191

[31] Tadeusz Pietraszek and Chris Vanden Berghe. 2005. Defending Against Injection Attacks Through Context-Sensitive String Evaluation. In *Recent Advances in Intrusion Detection Symposium (RAID)*. 124–145. https://doi.org/10.1007/11663812_7

[32] Rhino JavaScript homepage 2016. Rhino JavaScript homepage. (2016). Available at https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino.

[33] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do: A Large-scale Study of the Use of Eval in Javascript Applications. In *European Conference on Object-oriented Programming (ECOOP)*. Springer-Verlag, 52–78.

[34] Daniel Schoepe, Musard Balliu, Frank Piessens, and Andrei Sabelfeld. 2016. Let's Face It: Faceted Values for Taint Tracking. In *European Symposium on Research (ESORICS)*. Springer.

[35] Koushik Sen, Swaroop Kalasapur, Tasneem G. Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Special Interest Group on Software Engineering (SIGSOFT)*. ACM, 488–498.

[36] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: taint analysis of framework-based web applications. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 1053–1068. https://doi.org/10.1145/2048066.2048145

[37] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, David Herman, Brad Karp, and David Mazières. 2014. Protecting Users by Confining JavaScript with COWL. In *Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*. USENIX Association, 131–146.

[38] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. 2014. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *USENIX Security Symposium*. USENIX Association, 655–670.

[39] Zhendong Su and Gary Wassermann. 2006. The essence of command injection attacks in web applications. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 372–382. https://doi.org/10.1145/1111037.1111070

[40] Sun Spider. 1.0.2 JavaScript Benchmark. https://webkit.org/perf/sunspider/sunspider.html. (????). Accessed: April 2016.

[41] Omer Tripp, Pietro Ferrara, and Marco Pistoia. 2014. Hybrid security analysis of web JavaScript code via dynamic partial evaluation. In *International Symposium on Software Testing and Analysis, ISSTA*. ACM, 49–59. https://doi.org/10.1145/2610384.2610385

[42] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 87–97.

[43] Petar Tsankov, Marco Pistoia, Omer Tripp, Martin T. Vechev, and Pietro Ferrara. 2016. FASE: functionality-aware security enforcement. In *Annual Computer Security Applications Conference (ACSAC)*. IEEE, 471–483.

[44] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Krügel, and Giovanni Vigna. 2007. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Network and Distributed System Security Symposium (NDSS)*.

[45] Shiyi Wei and Barbara G. Ryder. 2013. Practical blended taint analysis for JavaScript. In *International Symposium on Software Testing and Analysis, ISSTA*. ACM, 336–346.

[46] Shiyi Wei, Omer Tripp, Barbara G. Ryder, and Julian Dolby. 2016. Revamping JavaScript static analysis via localization and remediation of root causes of imprecision. In *Special Interest Group on Software Engineering (SIGSOFT)*. ACM, 487–498. https://doi.org/10.1145/2950290.2950338

[47] Wei Xu, Sandeep Bhatkar, and R. Sekar. 2006. Taint-Enhanced Policy Enforcement: A Practical Approach to Defeat a Wide Range of Attacks. In *USENIX Security Symposium*. USENIX Association.