Static Security Evaluation of an Industrial Web Application

Gebrehiwet B. Welearegai
University of Potsdam
Potsdam, Germany
welearegai@uni-potsdam.de

Max Schlueter
University of Potsdam
Potsdam, Germany
mschlueter@uni-potsdam.de

Christian Hammer
University of Potsdam
Potsdam, Germany
hammer@cs.uni-potsdam.de

ABSTRACT

JavaScript is the most popular programming language for web applications. Static analysis of JavaScript applications is highly challenging due to its dynamic language constructs and event-driven asynchronous executions, which also give rise to many security-related bugs. Several static analysis tools to detect such bugs exist, however, research has not yet reported much on the precision and scalability trade-off of these analyzers. As a further obstacle, JavaScript programs structured in Node.js modules need to be collected for analysis, but existing bundlers are either specific to their respective analysis tools or not particularly suitable for static analysis.

In this paper we propose a novel approach to compare the precision, scalability and code coverage of two widely-used static analysis frameworks—WALA and SAFE—together with *simplePack*, which analyzer-agnostically bundles dependent modules, enabling a fair comparison. To appropriately evaluate the precision of the analyzers, we select all equivalent user object and variable references, and compute their properties' average points-to set sizes. Our evaluation indicates that SAFE provides higher precision and better code coverage at the cost of a somewhat lower scalability. Evaluating the simplePack bundler manifests that the static call graph of its bundle is more precise compared to the bundle produced by Browserify, one of the most popular module bundlers. Based on these results, we analyze the data flows of a hybrid app (JS & native) provided by an industrial partner via taint analysis. To that end we modeled the native (platform) functions of the app in a DSL for SAFE and extended its taint analysis to support tainted objects rather than only primitive type data. We show that there is potential for injection attacks, as tainted objects may reach the sink without being sanitized.

CCS CONCEPTS

• **Security and privacy → Information flow control**; *Web application security*; • **Theory of computation → Program analysis**; • **Software and its engineering** → *Software defect analysis*;

KEYWORDS

JavaScript, WALA, SAFE, comparison

ACM Reference Format:
Gebrehiwet B. Welearegai, Max Schlueter, and Christian Hammer. 2019. Static Security Evaluation of an Industrial Web Application. In *The 34th ACM/SIGAPP Symposium on Applied Computing (SAC '19), April 8–12, 2019, Limassol, Cyprus.* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3297280.3297471

1 INTRODUCTION

JavaScript is the most widely-used programming language for client-side web applications, powering over 95% of today's websites[1]. In spite of its popularity, it also introduces various errors, vulnerabilities, and ample challenges for program analysis: large-scale libraries, asynchronous event flows via user inputs, interaction between iframes, and new analysis domains like MVC frameworks and hybrid applications [32].

Researchers have proposed several static program analysis techniques to help JavaScript developers overcome some difficulties. Static analysis is challenging and imprecise due to dynamic language constructs like run-time code generation and heavy use of first-class functions [24]. Widely used static analysis frameworks like WALA [23, 30] and SAFE [16, 20, 22] represent the state of the art, however, little research assesses their strengths and weaknesses. Ko et al. [15] report the number of callees per call site of these frameworks but concentrate on the scalability gain of combining WALA and SAFE rather than their comparison. A recently released major rewrite of SAFE that aims at a more pluggable, extensible and debuggable framework (SAFE 2.0 [22][2]) raises the question which framework to leverage for static security analysis, which was the motivation for a thorough comparative analysis of WALA and SAFE.

Recently JavaScript's popularity also rose for server-side and desktop applications due to Node.js, an open-source, cross-platform runtime environment for executing JavaScript on the server-side[3]. Node.js programs are structured into modules, and can also be used in browsers by bundling up the module dependencies using *Browserify* [2], *Webpack* and *CommonJS Everywhere*. All these bundlers follow a similar pattern in how they package modules, but none of these are suitable for static analysis. Recently WALA and SAFE included direct support for Node.js modules, but their solutions are analysis-specific and thus not suitable for comparison. For example, the model in WALA is relatively complete (except for the native *core* modules), but SAFE's very preliminary. Hence, it is indispensable to have an analysis-agnostic bundler suitable for any static analysis framework for a fair comparison.

JavaScript- and Node.js-based applications can be vulnerable to various injection attacks [31]. Thus, security analyses like a taint

[1]https://w3techs.com/technologies/details/cp-javascript/all/all
[2]From now onwards SAFE represents the version SAFE 2.0.
[3]https://en.wikipedia.org/wiki/Node.js

analysis [33] are indispensable. The state of the art reports dynamic, static and hybrid taint analysis approaches [4, 26, 34]. However, none of them are implemented in SAFE, which we selected for our security analysis. SAFE supports taint analysis via a prototype implementation [25], but only reports tainted primitive type arguments such as strings that reach a sensitive sink, but not object type arguments, which our industrial application makes use of.

In this paper, we present a thorough comparison of SAFE and WALA, introduce *simplePack* to bundle module dependencies suitably for static analysis, extend the taint analysis in SAFE, and analyze a real-world industrial app using SAFE. To extract appropriate elements for comparison of the analyses, we integrate WALA's analysis into SAFE and select objects and global variables of the user program that are registered for the same source location in both analyzers. We compute the average points-to set sizes over all object fields (i.e., *pointer keys*) to evaluate the precision of both analyzers. We also evaluate their scalability and code coverage by measuring the analysis time and number of non-empty object fields (i.e. fields determined to point to an allocation site, e.g., for function objects), respectively. Our evaluation illustrates that SAFE provides higher precision and code coverage than WALA, but is less scalable. However, SAFE's lower scalability usually does not outweigh the gains in precision and code coverage. Our evaluation also shows that SAFE covers more code (determines receivers of function calls), which may also be responsible for the higher runtimes. Hence, we choose SAFE for our security analysis of the industrial app.

The hybrid industrial app is structured in the form of Node.js modules that reside in different paths of the project and cannot be analyzed by SAFE directly, as it requires all code to be present in one directory. Thus, we contribute *simplePack*, a source code transformation tool that bundles module dependencies in a way that is more suitable for static analysis. We compare *simplePack* to *Browserify* by measuring the precision and recall of their bundled programs' static callgraph in WALA. The evaluation shows that *simplePack* displays better precision and recall overall. The hybrid app, built on a middleware platform, includes more than 300 modules and contains more than 230,000 lines of code, which makes a direct whole program static analysis almost impossible. Hence we remove parts of the code not relevant to the main task of the app while keeping its major semantics and features.

We analyze the security of the hybrid app by extending SAFE's taint analysis to identify tainted *objects* flowing to sinks. We also model the *JSON.stringify* function, which acts as an input sanitizer by changing the input value or object to a non-executable JSON string. Note that this function just acts as a prototypical sanitizer to evaluate whether we can support sanitization in our analysis. Which function may act as a sanitizer depends heavily on the semantics of the sink and is beyond the scope of this research. We evaluate the analysis on four components of the hybrid app and our taint analysis identifies the tainted parameter object due to the existence of a tainted primitive property in the objects' property hierarchy. In contrast, tainted values passed through the *JSON.stringify* function before reaching a sink are correctly not reported as an illegal data flow. The major contributions of this paper are:

- A comparison of WALA's and SAFE's points-to analysis using equivalent user-defined objects.
- A framework-agnostic module bundler suitable for static analysis, which we compare against the off-the-shelf bundler Browserify.
- A static taint analysis for a hybrid industrial app. To that end we model the sanitizer JSON.stringify directly in SAFE, and extended the taint analysis to complex objects rather than tainted primitive values.

```
//url==https://...#name=<script>alert("xss")</script>
var url = document.URL
var pos = url.indexOf("name=")+5
document.write(url.substring(pos, url.length))
```

Figure 1: Taint analysis example

2 BACKGROUND

2.1 Points-to Analysis

A points-to analysis is a static code analysis that attempts to determine the possible values of a pointer or heap reference in a program. There are several pointer analysis algorithms, most of which leverage Andersen-style analysis [3] expressed as subset constraints. A pointer-analysis algorithm can be either flow-sensitive or flow-insensitive. The flow-sensitive analysis takes into account the order in which the statements in the program may be executed. The program is handled as a sequence of statements and the memory location where pointer expression may refer to is computed for each statement, i.e., it is statement level analysis. On the other hand, the flow-insensitive analysis does not take into account the order rather the program is handled as a set of statements. Hence, it computes the memory locations where pointer expressions may refer to, at any time in the program execution, i.e., it is program level analysis.

2.2 Taint Analysis

Taint analysis emerged as a useful technique for discovering security vulnerabilities in web applications [33]. Security taint analysis is an information-flow analysis that automatically detects flows of untrusted user input to security-sensitive computations (integrity violations) or flows of private information into computations that expose it to public observers (confidentiality violations) [29]. The starting point of taint analysis is typically a data source API call. For instance, consider the code sample in Figure 1 which is vulnerable to a DOM-based XSS attack. The variable *name* could contain malicious JavaScript code in a script tag that is then executed by `document.write`. Considering `document.URL` as a source and `document.write` as a sink, the taint analysis raises an alarm that untrusted user input is flowing into a sink. An application developer can endorse untrusted inputs via sanitization functions, which the taint analysis needs to consider, in order to reduce the number of false positives.

2.3 Static Analysis Frameworks

Static analysis mechanisms automatically derive certain properties from a program without executing it. Currently, there are four

```
function f1(v) {
  return f2(v);
};
function f2(v) {
  return v; }
var o1 = {};
var o2 = {};
var x = {};
x.p = f(o1);
x.q = f(o2);
```

(a) context-sensitivity

```
var x={};
for (var i=0;i<3;i++){
  x[i]=i;
}
```

(b) loop-sensitivity

```
var TopString = (45).
    toString(12);
var TopNum = Date.now()\
    right)
```

(c) model-functions

Figure 2: Examples to illustrates context-sensitivity, loop sensitivity and modeled functions

frameworks that analyze JavaScript[4] applications statically, i.e., TAJS [11], JSAI [13], WALA [23], SAFE [16, 22]. Reviewing the scientific literature we found WALA and SAFE to be most commonly used, so we decided to investigate these frameworks in greater detail. Additionally, the aim of this investigation was to identify a static analyzer that precisely analyzes the hybrid industrial app. We started out analyzing this app with WALA with low precision. Hence, we decided to evaluate at least one more analyzer that is in active development. Since the results with SAFE are promising we refrained from re-implementing our ideas in the remaining tools. Evaluating the precision of TAJS is subject to future work, JSAI is no longer under active development.

WALA provides soundy [17] flow-insensitive static analysis for both Java bytecode and JavaScript. However, JavaScript's ability to create and delete properties at runtime presents a great challenge for scalable and precise points-to analysis. Although WALA provides correlation tracking [30], improving points-to analysis scalability and precision via smart handling of *for-in* loops, scalability remains a problem for *inter-procedural, finite, distributive subset* (IFDS)-based analyzers. Therefore, WALA intentionally introduced a new unsound but more scalable static analysis that constructs a field-based (FB) call graph [9], i.e., uses one abstraction for all instances of each property in the whole program. To further improve the pointer analysis scalability and to eliminate calls to eval, Schäfer et al. [27] proposed a dynamic analysis. However, the coverage of the dynamic analysis may not be sufficient as the analysis observes only one program execution at a time.

SAFE [16] is a flow- and context-sensitive static analysis framework that provides both formal specification and its open-source implementation for JavaScript. Analysis scalability is greatly improved by using loop sensitivity (LSA) [20] that handles loops more precisely, which turns out to be a determining factor in terms of analysis precision as well. It also supports the *with* statement [19], rewriting it to semantically equivalent code when it does not contain any dynamically generated code. Recently, SAFE 2.0 [22] supports *pluggability* (ability to select analysis techniques at runtime), *extensibility* (APIs for adding new phases) and *debuggability* (HTML and console debugging), improving user-friendliness. In contrast WALA's source code repository is huge and complex. SAFE is able to analyze most ES5[5]codes precisely. To analyze object properties more precisely, SAFE uses *recency abstraction* [6, 21] which performs strong updates on recently allocated objects and weak updates on joined *old* objects.

[4]JavaScript analyses usually support a subset of a standard called ECMAScript (ES), ES5 is now common, ES6 still new.

Table 1: Points-to analysis result of program examples in Figure 2. Columns with *keys* and *avgPts* headings represent the number of pointer keys and average points-to values respectively.

program examples	1-CFA and 0-LSA				2-CFA, 3-LSA	
	SAFE		WALA		SAFE	
	keys	**avgPts**	**keys**	**avgPts**	**keys**	**avgPts**
a.	21	1.19	17	1.18	19	1.0
b.	2	1.0	3	2.67	5	1.0
c.	5	1.0	3	1.33	5	1.0

2.3.1 ***Motivating Example:*** The context- and loop-sensitivity, and the coverage of modeled functions have a great effect on the precision of static analysis frameworks. The three programs in Figure 2 and their analysis results in Table 1 illustrate this issue. For the context-sensitivity example (Figure 2a), the number of pointer keys and average points-to set size in SAFE using *2-CFA* is reduced due to the redundant old allocation sites, which apply weak updates. Particularly, using *1-CFA*, the old allocation site of argument v in function f_2 points to objects o_1 and o_2. But if we use *2-CFA* or more the old allocation does not exist and the recent allocation points-to o_2. Therefore, increasing the context-sensitivity leads to more precise points-to analysis as the weak updates are the main source of imprecision. However, WALA does not support context-sensitivity greater than *1-CFA* for JavaScript. Similarly, loop-sensitivity, which is not supported in WALA except for correlation tracking for *for-in* loops [30], is supported in SAFE and increases the precision by differentiating the pointers that depend on the iteration number. For instance, the *x[i]* pointer in Figure 2b varies and points to a different value at each iteration when *3-LSA* or more is used. Finally, the difference in the number of pointer keys for the example in Figure 2c is due to the fact that implicit primitive type conversion is modeled in SAFE but not in WALA.

2.4 Application Setup

For our security analysis and evaluation of the static analyzers, we use a real-world industry application which is structured into Node.js modules. This section presents the Node.js platform in general and the industry application setup in particular.

2.4.1 ***Node.js Platform:*** Node.js is an open-source, cross-platform JavaScript runtime environment, built on Chrome's V8 JavaScript engine, for executing JavaScript server-side code [8]. It uses an event-driven, non-blocking (asynchronous) I/O model that makes it lightweight and efficient. The node package manager (*npm*) is used to install modules and manage code dependencies from the command line. Node.js has built-in and user-defined JavaScript

[5]However, not all of ES5 is currently supported (e.g. getter/setters are not modeled).

```javascript
// main.js
var A = require("./A.js");
var a = new A();
a.foo(2) // => false
```

(a) main module

```javascript
// A.js
function A() {
  this.foo = function (x) {
    return x === 0; };
}
module.exports = A;
```

(b) dependent module

Figure 3: Node.js program example consisting of two modules that reside in the same directory.

modules. Modules structure a program into separate sub-programs to simplify the development and maintenance of complex applications. Each module has its own scope and cannot pollute the namespace of other modules.

Figure 3 illustrates how modules are used in Node.js. Module *A.js* exports a function constructor and is required by the main module `main.js`. In main.js we instantiate a new object from `A` and call the method `foo` on it. The `require` function has diverse semantics, e.g., loading built-in modules in Node.js or recursively searching through directories for modules installed by npm.

2.4.2 ***Hybrid App:*** This section provides an overview of a practical Node.js hybrid web app developed by ApproLogic GmbH that is used for our static analysis. The application is structured into Node.js modules and makes calls to middleware platform functions that are attached to the module scope during execution. Figure 4 shows an overview of the module dependencies in the hybrid app from the *LoginController* slice[6]. The *Platform* module represents the middleware platform and consists of several modules. The hybrid app includes more than 300 modules and contains more than 230,000 lines of code. The platform functions perform many interactions with built-in library functions such as *jQuery*, *Lodash* and *Backbone.js*.

3 IMPLEMENTATION

3.1 Average Points-to Set Computation

In this section, we present the general initial setup and the average points-to set computation approach of both static analyzers.

3.1.1 ***Initial Setup.*** WALA uses a flow-insensitive Andersen-style pointer analysis approach to compute the points-to set of each pointer or heap reference for the whole program. In contrast, SAFE supports flow-sensitive points-to analysis and the heap state is different from statement to statement. Hence, we take the heap status at the *exit statement* of the top-level function as the state to compare the average points-to sets.

[6]The other controllers are: *RegisterController*, *SigninController* and *AssistanceController*

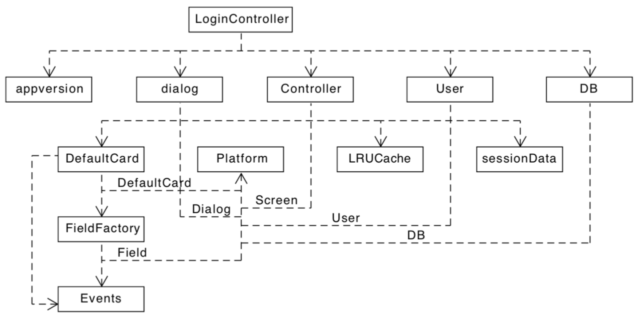

Figure 4: Module dependence graph of the ApproLogic hybrid app. The nodes represent modules and the dashed edges represent require dependencies.

To have a more appropriate comparison, we integrate the WALA analysis result into SAFE following the approach of Ko et al. [15], i.e., we added the WALA project to SAFE's and create a *PointerAnalysis* object of the whole program using the *JSCallGraphBuilderUtil* class in WALA. Then by traversing over the *InstanceKeys* of the PointerAnlysis object we map the InstancesKeys' source locations to their *creation sites*. Similarly, by iterating over SAFE's control flow graph (CFG), we compute the mapping of source locations to SAFE's object *allocation sites*. Finally, we only consider the source locations present in both maps and take the InstanceKeys objects with their corresponding allocation site in SAFE.

As the set of the modeled built-in JavaScript functions in WALA and SAFE are not the same, we only consider user-defined program object locations. Additionally, the representation of local variables and lexical variables differs between WALA and SAFE, and does not provide similar numbers and types of pointers. Hence, only the user-defined global variables, user objects and argument objects are selected as appropriate candidates for our average points-to comparison. For instance, the global objects o_1, o_2 and x in Figure 2a have the same source locations in both analyzers and they are selected for the points-to comparison. The argument and function prototype objects are compared and selected in a similiar fashion.

3.1.2 ***Average Points-to Set Computation in SAFE.*** To compute the average points-to set in SAFE, we take the heap at the *exit statement* of the top-level function. Since SAFE uses recency-abstraction for allocations in the heap, we compute an average effect of the old and recent objects of an allocation site to make it more comparable to WALA that does not rely on recency-abstraction. Afterwards, we iterate over all user and global allocation site objects, and compute the average points-to set of all properties. However, only user-defined properties are considered while iterating over the global object. Additionally, the *length* property and properties that points-to *undefined* are ignored in all objects. The length property indicates the number of arguments and is not relevant. The points-to set for properties that point to undefined is zero in WALA and can falsely increase the precision impact of other pointers that point to more than one object. Hence, we ignore properties that point to undefined in both analyzers. However, the number of undefined properties are separately computed to estimate the percentage of JavaScript constructs that are not modeled in SAFE and WALA.

```
#Document.prototype: {
   [[Class]]: "Document",
   [[Extensible]]: true,
   [[Prototype]]: #Node.prototype,
   "write": <#Document.prototype.write, F, T, T>,
   "URL": <string, F, T, T>
},
```

Figure 5: Modeling untrusted user input example

```
var url = strTop (tainted)
var input = {user: "user1", url: url}
var card = { header: headers, data: input}
sink1(url)
sink2(input)
sink3(card)
var sanitizedValue = JSON.stringfy(card)
sink4(sanitizedValue)
```

Figure 6: Tainted object flowing to sink

3.1.3 Average Points-to Set Computation in WALA.

In WALA, we compute the heap from the points-to set of the InstanceKeys and select the heap objects which have a corresponding object in SAFE. The source location is used for filtering. After identifying the relevant objects, we iterate over all properties of each object to compute the average points-to set. In addition to the properties of these objects, the user-defined global variables and the callee function pointer variables are also considered. Because, unlike SAFE, WALA does not include the callee property in the arguments object. The same as in SAFE, the pointers pointing to zero (undefined) are computed separately to extract the percentage of unmodeled JavaScript constructs (functions).

3.2 Taint analysis

In this section, we extend SAFE's taint analysis technique, which models the sources of untrusted user inputs with the abstract value *string*[7] prototype value (see Figure 5), propagates it during the analysis and finally checks whether a tainted value might be used at the sink. In SAFE's existing taint analysis, the arguments of the sink functions are checked whether they are tainted (*strTop*) or not. However, this does not work for objects that contain tainted values in their property hierarchy, rather only for primitive values. Yet, finding only primitive argument values passed to a sink is rare. We did not encounter such a situation when analyzing the hybrid app from our industry partner.

To illustrate our contribution, let us consider the code in Figure 6. SAFE identifies taint flow only in *sink*1 but not in *sink*2 and *sink*3. In our implementation, we recursively iterate over the properties of the sink's argument objects and search for tainted values (strTop). Accordingly, we can find the taint flow to sink2 and sink3.

[7]string represents the top string value (*strTop*) in the lattice, not a regular string value. Other lattices could in principle be embedded into SAFE's to enable disambiguating more taint sources.

Some of the controllers in the hybrid app use *JSON.stringfy* to change the input data into a non-executable JSON string. However, this function is not currently modeled in SAFE. To analyze applications containing this function we extended SAFE's model such that our taint analysis approach correctly indicates that tainted value is not flowing to *sink*4, i.e., it supports the sanitization of JSON objects using *JSON.stringfy*. The implementation of the taint analysis and the analyzer comparison is available on Github[8]

3.3 SimplePack

Browsers do not support CommonJS[9] module syntax. Hence, there have been many ongoing efforts to make the utilities available in npm accessible to the browser. Browserify is one of the most popular tools able to bundle CommonJS modules for the browser by concatenating the modules in a single file.

Although Browserify bundles CommonJs modules for browsers, the bundles are not well suited for static analysis. Moreover, none of the aforementioned static analysis frameworks supported programs written in CommonJS module syntax at the time that we started this research effort. Figure 7 depicts the Browserify bundle program for the example in Figure 3. Although the bundle produced by Browserify works for browsers, it is not well suited for static analysis. The main problems with the analysis of Browserify bundles are:

(1) Flow-insensitive points-to analysis will determine that the inner function may invoke module functions in any order.
(2) At least one level of call-string sensitivity is needed to distinguish between different required modules, as a require call will first invoke the inner function.
(3) The additional function calls by the inner function renders the call graph overly complex and thus also requires a more context-sensitive static analysis.

Hence, we introduce *simplePack*, which bundles CommonJS modules in a more suitable way for static analysis frameworks. In our approach we concatenate the module functions and transform the require calls into the module functions to outer function calls, i.e., there is no inner function. Figure 8 illustrates the *simplePack* bundle for the same example in Figure 3.

In our implementation, we first compute the set of module dependencies by walking the dependency graph. In Node.js, the package *module-deps* resolves dependencies using node's module lookup algorithm. The AST of the empty program and for each module dependency is computed using the package *esprima*, which computes a SpiderMonkey AST. Then a module is wrapped in a function declaration and its AST is traversed using the *estraverse* package. During traversal every *require* call that has a string literal as argument[10] is replaced by a function call to that module. The potentially modified AST is added to the program. In the end, a function call to the entry module is added to the program and source code is generated for it using the *escodegen* package. Our implementation is available on GitHub[11].

[8]https://github.com/ghiwet/safe/tree/ subdirectories taint, walaPointsTo, safePointsTo
[9]CommonJS is a project with the goal of specifying an ecosystem for JavaScript outside the browser
[10]Dynamically computed arguments are currently not supported and were not required for our analyses.
[11]https://github.com/MaxSchlueter/bundler

```
(function outer (modules, cache, entry) { function
    inner (name) {
  if (!cache[name]) {
    if (!modules[name]) throw "MODULE_NOT_FOUND";
      var m = cache[name] = {exports:{}};
      modules[name][0].call(m.exports, function(x){
        var id = modules[name][1][x];
        return inner(id ? id : x);
      },m,m.exports);
    }
    return cache[name].exports;
  }
  inner(entry);
  return inner;
})
({1:[function(require,module,exports){
  ... // code of A.js here
},{}],2:[function(require,module,exports){
  ... // code of main.js here
},{"./A.js":1}]},{},2);
```

Figure 7: Simplified Browserify bundle for the Node.js module example in Figure 3

```
function _mod_A(module) {
  var exports = module.exports;
  function A() {
    this.foo = function (x) { return x === 0; };
  }
  module.exports = A;
  return module.exports;
}
function _mod_main(module) {
  var exports = module.exports;
  var A = _mod_A({ exports: {} });
  var a = new A();
  a.foo(2) // => false
  return module.exports;
}
_mod_main({ exports: {} });
```

Figure 8: Simplified simplePack bundle for the Node.js module example in Figure 3

4 EVALUATION

4.1 Points-to Analysis Comparison:

In this section, we evaluate WALA's and SAFE's static program analyzers answering three research questions. First, we describe the research questions, evaluation methods and subjects (benchmarks). Then, we discuss the results of the analyses.

Algorithm 1: Pseudo Code for simple Pack

```
   input  : an entry module $file$
   output : a bundle for $file$
1  $modules$ ← Compute set of module dependencies for $file$;
2  $programast$ ← Compute AST of an empty program;
   foreach $mod \in modules$ do
3      $ast$ ← Compute AST for module $mod$;
4      $fun$ ← Wrap $ast$ in a function declaration;
5      Traverse fun and replace every require call by a function
         call to that module;
6      Add fun to $programast$;
7  end
8  Add function call to the entry module of $programast$;
9  $program$ ← Generate source code from $programast$;
```

4.1.1 *Research Questions:*

We present the research question as follows:

RQ1. Precision: For the object properties (pointers keys) of each subject, to how many object locations or values do they point to on average? An analyzer that results in a lower average points-to value is more precise (provided that both analyses are sound).

RQ2. Scalability: How much time does it take to analyze a subject and how many subjects are fully analyzed within a given timeout? An analyzer that finishes the analysis of a subject in less time is more scalable.

RQ3. Coverage: How many pointer keys whose value is not undefined[12] does each analyzer identify for each subject? An analyzer that results in a higher number provides better coverage (unless it is less precise) as the points-to set is undefined when its pointer refers to an API function or object that is not modeled in the analysis.

4.1.2 *Evaluation Methodology and subjects:*

To answer these research questions we performed experiments using the WALA and SAFE 2.0 analyzers. For evaluation subjects, we used a version of the *hybrid app* from our industry partner and two benchmark sets from different categories [14, 35]: *addon* (i.e., plugins for the Firefox browser) and *standard* (i.e., from SunSpider and V8 browser benchmark suites). Each category of our benchmarks contains seven subjects [14, 35]. For all experiments, WALA used a 1-CFA sound propagation-based (PB) analysis with correlation tracking. SAFE analyzer uses 20-CFA and (10,5)-LSA, 10-length and 5-depth (to distinguish nested loops) loop strings, with recency abstraction for the hybrid app and the addon benchmark category. For the standard benchmark category, SAFE used 20-CFA and 0-LSA because loops are very complex and the analysis does not terminate in the given timeout with loop-sensitivity. WALA does not handle loops in any particular fashion except for *for-in* loops, which do not appear in these benchmarks. In the version of the industrial partner's app we replaced libraries like Lodash and jQuery by equivalent JavaScript code, and Promises by regular callbacks. We also removed certain code that is independent of the core functionality. Also, there is no model for platform-specific APIs that are written in a language other than JavaScript. This allows fair comparison but may miss certain

[12]In WALA undefined is called zero.

Table 2: Precision, scalability and coverage of analyzers. Parenthesized numbers (SLOC) denotes the subjects' line of code without comments. The values under the scalability are time of analysis in seconds. Entries marked χ denote that the analyzers do not finish analysis within the timeout of 10 minutes. The parenthesized numbers (% of undefined) measures the percentage of pointers pointing to unmodeled API functions or objects

Subject (SLOC)		Precision		Scalability		Coverage (% of undefined)	
		SAFE	WALA	SAFE	WALA	SAFE	WALA
hybrid-app (1221)		1.005	1.48	117	2.35	2391	2835
addon	odesk-job-watcher (154)	1.012	2.04	4.25	1.76	159 (0%)	112 (12.5%)
	chess (222)	1.02	1.34	8.4	2.1	265 (0.4%)	135 (1.2%)
	coffee-pods-deals (358)	1.006	1.46	2.5	2.3	313 (0.6%)	287 (1.7%)
	pinpoints (537)	1.02	1.44	3.55	2.58	440 (0.9%)	339 (3%)
	tryagain (590)	1.004	1.29	5.85	1.56	503 (1.4%)	555 (.54%)
	less-spam-please (745)	1.028	1.29	11.2	2.6	725 (0.8%)	623 (11.4%)
	live-pagerank (865)	1.03	1.29	62	3.2	818 (0.6%)	702(4.2%)
standard	access-nbody (170)	1.0	1.03	12.15	1.7	195 (1.51%)	150 (3.23%)
	crypto-sha1 (177)	1.0	1.085	24.5	1.8	226 (0.04%)	59 (7.8%)
	splay (201)	1.17	1.13	133	7.20	156 (0%)	20601 (0%)
	richards (285)	1.48	5.79	206	2.8	339 (0%)	491(0%)
	3d-cube (343)	1.11	3.97	380	4.88	1865 (0%)	3252 (0%)
	3d-raytrace (408)	χ	1.31	χ	1.23	χ	564 (2.5%)
	cryptobench (1297)	χ	χ	χ	χ	χ	χ

security issues. Therefore, as part of future work we are planning to extend our analysis in order to lift most of these restrictions. We conducted the experiments on a MacBook Pro with 2.9 GHz Intel Core i7 processor and 16GB RAM

4.1.3 ***Result and Discussion:*** In this section, we discuss the precision, scalability and coverage result of the two analyzers. Table 2 depicts the evaluation result of the *hybrid app, addon and standard* subject categories.

Precision: To compare the precision of the two analyzers, we measure the average points-to size of object properties over all user-defined locations (objects) and global variables. Although we consider only user-defined objects and global variables, we expect the results to be representative for the whole program. For all subjects that finish execution within the timeout, SAFE provides lower average points-to sizes than WALA. For the *odesk-job-watcher*, *richards* and *3d-cube* subjects, WALA's average points-to size exceeds SAFE's by a factor of two. Considering the average over all subjects, SAFE computes 1.017 pointers for the addon and 1.15 for the standard category. In contrast, WALA determines 1.45 for addon and 2.6 for standard. Last but not least, the average points-to size over all subjects is 1.073 for SAFE and 1.93 for WALA. This demonstrates that SAFE provides more precise analysis than WALA. The improved precision of SAFE is due to its support for flow-sensitivity, context-sensitivity and loop-sensitivity in its analyses

Scalability: We evaluated the scalability of both analyzers by measuring the analysis time in seconds. For all subjects, WALA requires less time to finish analysis, indicating that it is more scalable. Analysis of *3d-raytrace* in SAFE did not terminate within the given timeout although it did in WALA. The scalability of the WALA analysis is due to its flow-insensitive points-to analysis approach. However, there is a cost in terms of precision which may be more relevant for security analysis.

Coverage: To compare the coverage of the analyzers, we counted the number of object properties for each subject. Properties pointing to non-undefined values, and those pointing to undefined are counted separately. For the majority of the subjects, SAFE computes higher non-undefined and lower undefined pointers, indicating that it models more API functions. The model coverage and the loop sensitivity has enormous effect on the number of pointers. In some cases, the number of non-undefined pointers is higher in WALA due to heap cloning, which creates different objects based on the context, for the same object locations. However, this happened only for around 30 % of the subjects. Therefore, this indicates that in general SAFE provides better coverage.

Threats to Validity: The following are identified as threats to validity.

- The subjects used for our analysis may not be representative for other JavaScript applications.
- The analysis is based only on the user program and does not consider local variables (as these are specific to the intermediate representation), which might deviate the average points-to set size over the whole application.

To generalize the comparison of both analyzers, we have listed some of the comparison features in Table 3.

4.2 Taint Analysis

In this section, we evaluate our taint analysis by investigating whether it identifies objects containing tainted values and considers sanitization using JSON.stringify.

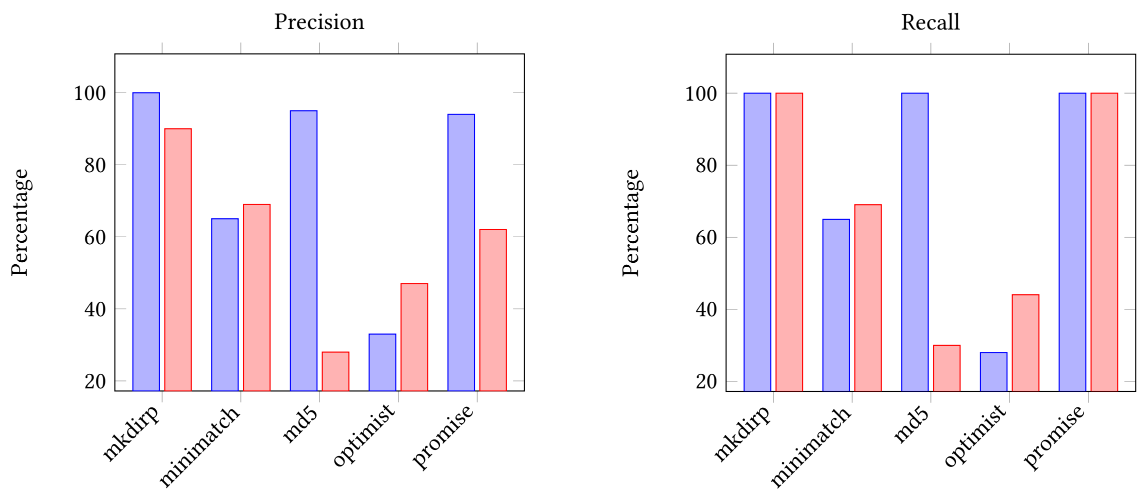

Figure 9: Precision and recall in percent for simplePack and Browserify bundles.

Table 3: Feature comparison of SAFE and WALA

Features	SAFE	WALA
Flow-sensitivity	Flow-sensitive	Flow-insensitive
Context-sensitivity	Any number	0 or 1-CFA
Loop-sensitivity	Any number	Only for-in
Precision	Relatively precise	Imprecise
Code Complexity	Easier to understand	Complex to understand
Soundiness [17]	Sound	Sound(PB) and unsound(FB)
Scalability	Less scalable	More scalable

For our evaluation, we use the hybrid app. The hybrid app contains different controllers for login (configuration setup and automatic login using saved credentials) , signin (login by entering user credentials), register and assistance. All these controllers take user input and interact with the internet using HTTPS requests and post APIs. Hence, we took the program slice of each controller (i.e. all the code that can influence its computation) bundling all modules related to the respective controller's semantics, creating four separate programs to evaluate our taint analysis on. The user inputs at the respective controllers' form fields are considered a source of tainted value and the HTTPS request and post APIs are considered to be sinks. We use the abstract value *@StrTop* to simulate a tainted value.

Table 4 illustrates the result of the four program slices of the hybrid app, which all have more than 4,000 lines of code (without comments, SLOC). The result indicates that the tainted value is identified at the sink in both the login and signin program slices. Note that the application is still safe due to the way the tainted input is sent to the database in the platform code, which we verified by manual inspection. For the register and assistance program slices, no tainted value is identified at the sink. Upon manual investigation, we found that *JSON.stringify* is used to endorse the input before reaching the sink. When we remove this call to *JSON.stringify*, our taint analysis approach identifies the tainted input at the sink in both cases. Our manual investigation shows that the depth of the tainted properties in the parameter object is one or higher in all slices, which means that it could not be identified using SAFE's existing taint analysis. However, our extended taint analysis identifies the tainted objects.

Table 4: Taint analysis result of four program slices from the real world hybrid app

program (SLOC)	Tainted	Sanitized	min−depth
login (4788)	yes	no	1
signin (4890)	yes	no	1
register (4501)	no	yes	2
assistance (4730)	no	yes	2

4.3 simplePack

To evaluate *simplePack*, we compare the static call graph (CG) of a bundled program to the dynamic one. This is done for programs transformed by simplePack and by Browserify. The dynamic CG is constructed by a Jalangi2[13] [28] analysis records all the functions that are invoked at a call site in one execution of the program. A further restriction in this evaluation is that only user-defined functions in the code are considered as call site targets. The static CG is constructed in WALA and conservatively approximates all the functions that can be invoked at some call site. The most precise static CG is the union of the dynamic CGs over all (potentially infinitely many) possible program executions.

The static and dynamic CGs are compared by computing the average precision and recall of all the call sites covered by the dynamic CG as in [10]. For a given call site the precision is the percentage of "true" function targets and recall is the percentage of correctly identified true targets with respect to all targets. For the evaluation we use five Node.js programs based on five different packages obtained from npm considering the list of most-depended packages. The selected packages have at least one module dependency, can be transformed by Browserify and simplePack and the resulting

[13] https://github.com/Samsung/jalangi2

bundles are executed in node v0.12.x. Packages with dependency cycles or dynamic requires are excluded[14].

The average precision and recall of the bundles produced by simplePack and Browserify are depicted in Figure 9. Precision and recall are very high for the simplePack bundles of mkdirp, md5 and promise, achieving 100% recall and over 90% precision. The Browserify bundles for mkdirp and promise also achieve 100% recall but have lower precision measures than in simplePack. Recall and precision are particularly low for the md5 Browserify bundle when compared to the one by simplePack. The precision and recall measures for the mkdirp and optimist bundles are quite low and the Browserify bundles seem to be far better than their corresponding simplePack bundles. A recall measure less than 100% means that some call sites or function targets are missing in the static CG computed by WALA. Thus, the static analysis done by WALA for these bundles is unsound. Recall is at 65% for the minimatch simplePack bundle and at 69% for the corresponding Browserify bundle. Upon manual inspection we found that WALA was unable to resolve the same call sites and function targets in the Bundler bundle as in the corresponding Browserify bundle. Three unresolved call sites and their subsequent function calls lead to a low recall measure.

5 RELATED WORK

JavaScript static analysis frameworks. TAJS [11], JSAI [13], WALA [9, 23, 27, 30], SAFE [16, 20, 22] are JavaScript static analysis frameworks among which WALA and SAFE are most commonly used in research projects according to citation numbers. WALA provides soundy flow-insensitive static analysis supporting correlation tracking [30] to improve the analysis scalability and precision. Additionally, WALA introduced a new unsound but more scalable static analysis that performs field-based (FB) call graph [9]. SAFE [16] is a flow- and context-sensitive scalable static analysis framework which supports loop-sensitivity analysis (LSA) [20]. LSA enhances the analysis precision in loops improving analysis scalability of JavaScript applications. The current version SAFE 2.0 [22] supports *pluggability*, *extensibility* and *debuggability* which makes it more user-friendly than WALA and SAFE 1.0. SAFE leverages a *recency abstraction* [6, 21], which performs strong updates on recently allocated objects and weak updates on joined old objects.

Points-to Analysis Comparison. The average points-to set size analysis is a convenient way to evaluate the precision of static analysis frameworks. For instance, Wei et. al. [35] used the average points-to comparison to systematically select a precise context-sensitivity type (call-site, object, and parameter) per function. A context-sensitivity type that provides the smallest average points-to set size is selected for each function. This motivated us to evaluate the precision and scalability trade-off between WALA and SAFE using average points-to set sizes. However, both frameworks follow different points-to analysis approaches and leverage different data structure representations, which makes the comparison non-trivial. Hence, we use the approach followed by Ko et al. [15] to integrate the WALA analysis results into SAFE. This simplifies the selection of comparable objects. Ko et al. [15] also evaluates the precision of WALA's FB, SAFE and their combinations based on the average number of callees of the call sites. In contrast, we use the object properties and their points-to set to evaluate the precision of WALA's PB and SAFE. The object properties include the callees and other fields which broadens our analysis scope. Moreover, unlike the related paper, which integrates WALA with SAFE 1.0, we integrate WALA with the latest, written completely from scratch, version SAFE 2.0 to identify the comparable objects in both analyzers.

Taint analysis. SAFE's existing taint analysis [25] forms the basis of our approach. However, it does not support primitive arguments reaching the sinks, nor sanitizers. In contrast, our analysis supports object arguments. Additionally, we modeled *JSON.stringify*, and our analysis correctly identifies when the tainted user input is sanitized using this function. Taint analysis is well-established for security purposes. In the domain of JavaScript, however, due to its dynamic nature, there is only limited related work [4] leveraging static analysis: Kannan et al. [12] and Saoji et al. [26] present dynamic analyses to detect illicit taint flows, which can only assert security for the given program execution. Wei and Ryder [34] leverage a combination of static and dynamic analysis to identify security vulnerabilities due to data integrity violations in JavaScript codes in websites. Again, the dynamic analysis component may miss certain security-relevant facts. Skoruppa et al. [5] detect security violations in an online voting client written in JavaScript via pure static analysis with WALA. None of these approaches, though, analyzes module-based code based on node.js, which increases complexity significantly. Besides, we are the first to assess the precision and scalability of different static analysis frameworks precisely. This allows us to scale the analysis to even more complex language features, which we are actively pursuing at the time of this writing.

String analysis Costantini et al. [7] propose an abstract interpretation-based framework for string analysis and instantiates the framework for four different abstract domains. But they focus on the theoretical aspects of string domains and do not discuss string equality which is a key issue for dynamic field access. To precisely analyze dynamic field access in JavaScript, Madsen and Andreasen [18] present 12 string domains, among which 7 are new. SAFE uses the string set domain, one of the 12 string domains, for the string analysis. Recently, Amadini et al. [1] present SAFE$_{str}$ as an extension to SAFE and support a number of string analysis domains, as well as arbitrary combinations of these domains. However, we use only the string set domain in our analysis as SAFE$_{str}$ is not included in the public SAFE repository.

6 CONCLUSION

We provide a thorough comparison of SAFE and WALA by integrating the analysis result of WALA into SAFE and computing the average points to set of pointers (object properties) considering user-defined objects, global variables and others comparable elements only. The source location is used in both analyzers to select the comparable objects and variables. The average points-to set analysis result indicates SAFE is very precise at the cost of some scalability. SAFE also provides better model coverage. For static analyses, precision is often more crucial than scalability (as long as it finishes execution in a given timeframe), especially for security-related analysis. Hence, we selected SAFE to analyze a

[14]This only happened for one package encountered in the list of most-depended packages and never for our industry application.

real-world hybrid app from our partner company. However, the app is structured in CommonJs modules and we design *simplePack* that bundles the module dependencies into a single file suitable for SAFE analysis. *SimplePack* provides better precision and recall of a static callgraph compared to Browserify.

To analyze the security of the hybrid app, we took some of its program slices with independent sources and perform taint analysis by simulating tainted values as @StrTop in SAFE. Since the existing taint analysis in SAFE supports only primitive values, we extend it to identify tainted objects as well. As the result, our extended taint analysis tool correctly identified the flow of tainted values from sources to sinks in the hybrid app. However, as there is only one taint, our taint analysis cannot identify the exact source of the tainted value at the sink, an extension for future work.

The significantly better precision of SAFE allows us to extend static JavaScript analysis to domains that we ignored or manually rewrote in this study, which we are actively developing for future work. This is supported by the improved user-friendliness of the SAFE 2.0 library, which eases code understanding and facilitates contributions from external developers.

Acknowledgments. This work was supported by the German Federal Ministry of Education and Research (BMBF) through the project SimoBA (16KIS0440).

REFERENCES

[1] Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J Stuckey, and Chenyi Zhang. 2017. Combining string abstract domains for JavaScript analysis: an evaluation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 41–57.

[2] Tim Ambler and Nicholas Cloud. 2015. Browserify. In *JavaScript Frameworks for Modern Web Dev.* Springer, 101–120.

[3] Lars Ole Andersen. 1994. *Program analysis and specialization for the C programming language.* Ph.D. Dissertation. University of Cophenhagen.

[4] Esben Andreasen, Liang Gong, Anders Møller, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. 2017. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 66.

[5] M. Backes, C. Hammer, D. Pfaff, and M. Skoruppa. 2016. Implementation-level Analysis of the JavaScript Helios Voting Client. In *31st ACM Symposium on Applied Computing (SAC'16).* https://doi.org/10.1145/2851613.2851800

[6] Gogul Balakrishnan and Thomas Reps. 2006. Recency-abstraction for heap-allocated storage. In *International Static Analysis Symposium.* Springer, 221–239.

[7] Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2011. Static analysis of string values. In *International Conference on Formal Engineering Methods.* Springer, 505–521.

[8] Ryan Dahl. [n. d.]. Node.js. https://nodejs.org/

[9] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *Proceedings of the 2013 International Conference on Software Engineering.* IEEE Press, 752–761.

[10] Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *Software Engineering (ICSE), 2013 35th International Conference on.* IEEE, 752–761.

[11] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type Analysis for JavaScript.. In *SAS*, Vol. 9. Springer, 238–255.

[12] 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.*

[13] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 121–132.

[14] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering.* ACM, 121–132.

[15] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically tunable static analysis framework for large-scale JavaScript applications (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on.* IEEE, 541–551.

[16] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, Vol. 10.

[17] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. https://doi.org/10.1145/2644805

[18] Magnus Madsen and Esben Andreasen. 2014. String analysis for dynamic field access. In *International Conference on Compiler Construction.* Springer, 197–217.

[19] Changhee Park, Hongki Lee, and Sukyoung Ryu. 2014. All about the with statement in javascript: Removing with statements in javascript applications. *ACM SIGPLAN Notices* 49, 2 (2014), 73–84.

[20] Changhee Park and Sukyoung Ryu. 2015. Scalable and precise static analysis of JavaScript applications via loop-sensitivity. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[21] Jihyeok Park, Xavier Rival, and Sukyoung Ryu. 2017. Revisiting Recency Abstraction for JavaScript Towards an Intuitive, Compositional, and Efficient Heap Abstraction. In *SOAP.*

[22] Jihyeok Park, Yeonhee Ryou, Joonyoung Park, and Sukyoung Ryu. 2017. Analysis of JavaScript web applications using SAFE 2.0. In *Proceedings of the 39th International Conference on Software Engineering Companion.* IEEE Press, 59–62.

[23] IBM Research. [n. d.]. WALA The TJ Watson Libraries for Analysis. http://wala.sourceforge.net/

[24] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. 2010. An analysis of the dynamic behavior of JavaScript programs. In *ACM Sigplan Notices*, Vol. 45. ACM, 1–12.

[25] Sukyoung Ryu, Alexander Jordan, and Dongsun Kim. [n. d.]. SAFE Tutorial: Taint Analysis for Web Applications. http://plrg.kaist.ac.kr/lib/exe/fetch.php?media=research:publications:pldi2017_tutorial2.pdf

[26] Tejas Saoji, Thomas H Austin, and Cormac Flanagan. 2017. Using Precise Taint Tracking for Auto-sanitization. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security.* ACM, 15–24.

[27] Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Dynamic determinacy analysis. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 165–174.

[28] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering.* ACM, 488–498.

[29] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: taint analysis of framework-based web applications. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 1053–1068.

[30] Manu Sridharan, Julian Dolby, Satish Chandra, Max Schäfer, and Frank Tip. 2012. Correlation tracking for points-to analysis of JavaScript. *ECOOP 2012–Object-Oriented Programming* (2012), 435–458.

[31] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. 2016. Understanding and automatically preventing injection attacks on node. js. (2016).

[32] Kwangwon Sun and Sukyoung Ryu. 2017. Analysis of JavaScript Programs: Challenges and Research Trends. *ACM Computing Surveys (CSUR)* 50, 4 (2017), 59.

[33] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. In *ACM Sigplan Notices*, Vol. 44. ACM, 87–97.

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

[35] Shiyi Wei and Barbara G Ryder. 2015. Adaptive context-sensitive analysis for JavaScript. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.