Platform-Independent Dynamic Taint Analysis for JavaScript

Rezwana Karim, Frank Tip, Alena Sochůrková, and Koushik Sen

Abstract—Previous approaches to dynamic taint analysis for JavaScript are implemented directly in a browser or JavaScript engine, limiting their applicability to a single platform and requiring ongoing maintenance as platforms evolve, or they require nontrivial program transformations. We present an approach that relies on instrumentation to encode taint propagation as instructions for an abstract machine. Our approach has two key advantages: it is *platform-independent* and can be used with any existing JavaScript engine, and it can track taint on primitive values without requiring the introduction of wrapper objects. Furthermore, our technique enables multiple deployment scenarios by varying when and where the generated instructions are executed and it supports *indirect taint sources*, i.e., situations where taint enters an application via arguments passed to dynamically registered event-listener functions. We implemented the technique for the ECMAScript 5 language in a tool called *Ichnaea*, and evaluated it on 22 NPM modules containing several types of injection vulnerabilities, including 4 modules containing vulnerabilities that were not previously discovered and reported. On these modules, run-time overheads range from 3.17x to 38.42x, which is significantly better than a previous transformation-based technique. We also report on a case study that shows how *Ichnaea* can be used to detect privacy leaks in a Tizen web application for the Samsung Gear S2 smart watch.

Index Terms—Taint analysis, dynamic analysis, JavaScript, platform-independent, instrumentation

1 Introduction

JavaScript applications commonly operate on untrusted or confidential information. The use of such data must be controlled carefully in order to avoid security vulnerabilities and privacy leaks. For example, a privacy leak may arise if confidential information is allowed to flow to an operation such as an HTTP request where data is publicly disclosed. Similarly, injection vulnerabilities may exist if unsanitized input data is allowed to flow to the `eval` function, which interprets a string value as executable code, or, on the Node.js platform, to the `child_process.exec` function, which interprets a string value as an executable shell command. Since the use of unsafe operations such as `eval` is pervasive [1], such vulnerabilities are quite common as is evident from many reported issues on forums such as https://nodesecurity.io/, and a recent study reported that many Node Package Manager (NPM) modules are riddled with injection vulnerabilities [2].

Dynamic taint analysis is a data-flow analysis technique for determining expressions whose value indirectly is derived from specified input values. Privacy leaks and security vulnerabilities can be detected using dynamic taint analysis by tracking the flow of data from "sources" where private or confidential information enters the application to "sinks" where such information is disclosed or manipulated. In principle, our technique can be used for any application of dynamic taint analysis assuming that the application's source code is available. However, we choose to focus our attention on usage scenarios where taint analysis is performed in-house by developers prior to deployment. An example of such a scenario is one where a JavaScript application depends on third-party libraries or modules that may contain vulnerabilities [3]. Due to the highly dynamic nature of the JavaScript language, such vulnerabilities may not be easily discernible through source code inspection. Using our dynamic taint analysis, developers can detect such vulnerabilities prior to deployment, by creating and running tests with taint tracking enabled.

Most previous approaches to dynamic information flow analysis[1] for JavaScript require modification of an interpreter or JavaScript engine to keep track of information flows during execution [4], [5], [6], [7]. While such an approach has significant performance advantages, the resulting analysis becomes platform-specific, which limits its applicability, especially in cases where applications behave differently on different platforms.[2] Therefore, information flows detected during execution of an application on one browser may or may not happen

- *R. Karim is with Samsung Research America, Mountain View, CA 94043 USA. E-mail: rezwana.k@samsung.com.*
- *F. Tip is with the College of Computer and Information Science, Northeastern University, Boston, MA 02115 USA. E-mail: f.tip@northeastern.edu.*
- *A. Sochůrková is with Avast, Prague 140 00, Czechia. E-mail: sochurkova.alena@gmail.com.*
- *K. Sen is with the University of California at Berkeley, Berkeley, CA 94720-5800 USA. E-mail: ksen@berkeley.edu.*

Manuscript received 9 Feb. 2018; revised 31 Aug. 2018; accepted 14 Oct. 2018. Date of publication 26 Oct. 2018; date of current version 10 Dec. 2020. (Corresponding author: Frank Tip.)
Recommended for acceptance by T. Bultan.
Digital Object Identifier no. 10.1109/TSE.2018.2878020

1. In the remainder of this paper, we will use terms 'taint analysis' and 'information flow analysis' interchangeably.

2. For example, web sites that include social media plugins commonly use conditional loading and execution of JavaScript code to avoid displaying certain information (e.g., "like counts") in order to reduce load time on mobile devices.

when another browser is used. Approaches based on modification of a JavaScript engine also have the significant drawback that ongoing maintenance is required, given that browsers tend to evolve quickly. Other previous work includes a platform-independent technique [8] that relies on wrapping primitive values and other nontrivial program transformations that incur significant runtime overheads.

In this paper, we pursue a *platform-independent* approach to dynamic taint analysis for JavaScript that is based on *code instrumentation*. While instrumentation-based approaches to dynamic taint analysis have been pursued in other settings (e.g., the DTA tool [9] implements a dynamic taint analysis on top of Intel's binary instrumentation framework Pin [10]), to our knowledge, we present the first instrumentation-based dynamic taint analysis for JavaScript. Our solution handles the ECMAScript 5 language[3] and is implemented using the Jalangi instrumentation framework [11].

The basic idea behind our approach is to instrument JavaScript source code so that, for each JavaScript source construct that is executed, instructions are issued for an abstract machine. Executing these abstract machine instructions reflects the flow of taint between abstract locations that represent memory locations manipulated by the original application. The abstract machine is implemented as a domain-specific language (DSL) that is embedded in JavaScript. This approach has two key advantages: (i) it is platform-independent and can be used with any existing JavaScript engine, and (ii) it can track taint on primitive values such as numbers and strings without requiring the introduction of wrapper objects. Furthermore, the technique provides flexibility as to when and where to perform the analysis (the instructions can either be executed on-line on the same device, or they can be transmitted for offline execution) and it supports *indirect taint sources*, where taint information enters an application as arguments passed to user-defined callback functions that are registered with a known API, a scenario that arises in Tizen web applications [12].

We implemented the technique for the ECMAScript 5 language in a tool called *Ichnaea*, which builds on Jalangi [11], an open-source code instrumentation framework for JavaScript. One of the key challenges in our implementation involved modeling how taint is propagated through native functions, particularly for higher-order functions such as `Array.prototype.reduce`.

To demonstrate the practicality of our technique, we applied *Ichnaea* to 22 modules for the Node.js platform containing several types of injection vulnerabilities, including 4 modules containing vulnerabilities that were not reported previously, and confirmed that *Ichnaea* reports the expected taint flows when these modules are invoked in ways that trigger a vulnerability. On these modules, we observed runtime overheads ranging from 3.17x to 38.42x compared to uninstrumented execution, which is significantly faster than a previous transformation-based information-flow analysis for JavaScript [8] (see Section 6). Additionally, we report on a case study that shows how the technique can be applied to detect privacy leaks in a JavaScript web application for the Tizen platform that runs on a Samsung Gear S2 smart watch.

3. Subject to some minor restrictions that we inherit from Jalangi.

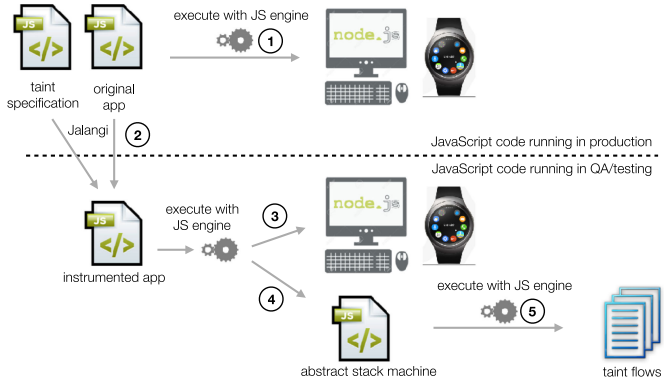

Fig. 1. Overview of approach.

In summary, this paper makes the following contributions:

- A platform-independent dynamic taint analysis for the ECMAScript 5 language. The technique is capable of tracking taint on primitive values without requiring boxing.
- Our technique is capable of tracking taint precisely through calls to native higher-order functions by relying on user-specified taint specifications.
- An implementation of the technique in a tool called *Ichnaea*.
- An evaluation of *Ichnaea* that demonstrates its practicality by confirming the flow of tainted input data in Node.js modules with known and previously unknown vulnerabilities, and through a case study in which the technique is applied to detect privacy leaks in a Tizen web application.

2 Overview

This section presents an informal overview of our dynamic taint analysis technique and illustrates it using a motivating example.

2.1 Approach

The diagram in Fig. 1 shows a high-level overview of our approach. The top part of the figure visualizes the execution of a JavaScript application with an unmodified browser or JavaScript engine (see arrow labeled ①). Here, the program's execution results in the normal application behavior. As an example, one may consider running a JavaScript application on the Node.js platform [13], or running a JavaScript web application for the Tizen operating system on a Samsung Gear S2 SmartWatch.

Our approach requires that the application's JavaScript source code be instrumented. In addition to the source code for the original application and any libraries and frameworks that it depends upon, the instrumentation takes as input a *taint specification* that specifies sources and sinks. This instrumentation is visualized in the diagram using downward arrows labeled ②. The instrumentation is implemented using the Jalangi instrumentation framework [11] and will be discussed in detail in Section 4.

Executing the resulting instrumented JavaScript application using an unmodified browser or JavaScript engine yields the same execution behavior as before (arrow ③) but additionally produces instructions for an abstract machine

```
1 var child_process = require('child_process');
2  var a = ['ls '];l1
3  a[1] = process.argv[2];l2
4  var command =l3
5   a.reducel4,l5,l6 ( function cb(x,y)l7,l8 {  return x+y;l9  }, ' ');
6 child_process.exec(command, function(err,stdout,stderr){
7  if (!err){ console.log(stdout.toUpperCase()); }
8 });
```

Fig. 2. Example Node.js application `uppercasels.js`.

(arrow ④), which is implemented as a domain-specific language in JavaScript. Executing these abstract machine instructions using an unmodified JavaScript engine or browser (arrow ⑤) produces a report about information flows from specified sources to specified sinks that took place in the execution labeled ③. Note that this approach provides flexibility in terms of where and when the generated abstract machine instructions are executed. For example, they can be executed on the same platform as the original application, or they can be transmitted and executed on a server elsewhere.

2.2 Example Application

Fig. 2 shows a small Node.js application, which prints the contents of a directory in uppercase font. The shading and labeling of code fragments in the figure will be explained when we discuss the generation of abstract machine instructions and can be ignored for now. Executing the application with command-line argument `"."` will print the contents of the current directory after converting all characters to uppercase. This is accomplished by concatenating the input value `"."` that was passed on the command line to a string literal `'ls '`, executing the resulting string value as a shell command using the `exec` function provided by the `child_process` module, converting the output produced by that command to uppercase, and printing the resulting value.

The example application has a command-injection vulnerability that is similar to vulnerabilities in several of the Node.js modules that we used to evaluate our technique (see Section 5). This vulnerability can be demonstrated by executing the application with a command-line argument `".;touch xyzzy"`. In this case, since the input value is passed to the `exec` command unmodified, the string `"ls .;touch xyzzy"` is passed to `exec`. Since `;` is used to delimit shell commands, the call to `exec` now has the effect of executing two commands, where the second command has the effect of creating a file `xyzzy` in the local directory.[4] Injection vulnerabilities may also arise in the presence of `eval`, a widely-used function for interpreting a string value as JavaScript code and executing it [1].[5]

Fig. 3 shows the taint specification and configuration parameters for analyzing the above Node.js application. Lines 9–14 define the sources and sinks for the analysis. In general, one can use these lines to specify the names of functions that serve as sources and sinks, respectively. In the example given here, `exec` is listed as a sink on line 12, and no function is listed as a taint source. Instead, command-line arguments are defined as sources of tainted data by setting the configuration parameter `taintNodeCommandLineInput` to `true` on line 15. Alternatively, the parameter `taintAllUserDefinedString` (set to `false` here) can be set to `true` if any user-defined string should be treated as a source of tainted data. The analysis can be further configured to report the location of the taint flow and to report either only the first taint flow, or all taint flows using the `reportFlowLocation` (line 17) and `reportAllFlows` (line 18) parameters.

4. In general, any type of shell command can be injected in a similar fashion, including commands with harmful effects such as deleting files.

5. Note that on the Node.js platform, it may be possible to inject a call to `child_process.exec` into the code that is executed by `eval`, thus providing attackers with significant control over the platform.

2.3 Execution Behavior

Before we discuss how our taint analysis tracks flows from input values to invocations of functions such as `exec`, we need to consider the execution behavior of the example application in more detail. Execution begins on line 1 by executing the `require` function in order to import the `child_process` module and assigning the resulting value to a variable `child_process`, through which the module's functionality can be accessed henceforth. Next, line 2 creates an array containing a string `'ls '` and assigns it to a variable `a`. On line 3, the argument passed to our example application is retrieved by reading the value at index 2 in the array `process.argv` (note that the element at index 0 in `process.argv` is the fully qualified filename of the `node` command, and the element at index 1 is the fully qualified filename of the application, in this case, `uppercasels.js`). Line 5 invokes a native library function `Array.prototype.reduce` to concatenate all arguments of array `a`, preceded by a string `' '`, by repeatedly invoking the callback function `cb`. The resulting value is stored into a variable `command` on line 4. Finally, on lines 6–8, the `exec` function is invoked to execute the command. The second argument passed to `exec` is a callback function that is invoked with three arguments: (i) a code `err` indicating whether an error occurred, (ii) a string `stdout` containing output written by the shell command to the standard output stream, and (iii) a string `stderr` containing output written by the shell command to the standard error stream. If no

```
9  Spec.taintSpec = {
10  "source" : [],
11  "sink" : [
12       {"name": "exec"}
13   ]
14 };
15 Spec.taintNodeCommandLineInput = true;
16 Spec.taintAllUserDefinedString = false;
17 Spec.reportFlowLocation = true;
18 Spec.reportAllFlows = true;
```

Fig. 3. Taint specification for `uppercasels.js`.

```
                                                                                    l1
19 push(false);                                  // load taint value for literal 'ls'
20 initproperty('obj7','0');                     // initialize element 0 of array 'obj7'
21 push(false);                                  // load taint value for array literal
22 writevar('frame3:a');                         // store taint for variable 'a'
23 pop();                                        // discard taint of assignment expression
                                                                                    l2
24 readvar('frame3:a');                          // load taint for variable 'a'
25 push(false);                                  // load taint for index expression '1'
26 readvar('frame5:process');                    // load taint for variable 'process'
27 push(false);                                  // taint value for 'argv'
28 readproperty('obj9', 'argv');                 // read taint of property 'argv' of object 'obj9'
29 push(false);                                  // taint value for literal '2'
30 readproperty('obj11', '2');                   // read taint for element '2' of array 'obj11'
31 pop();                                        // discard previously read taint value
32 push('(example.js:3:8)');                     // introduce taint value '(example.js:3:8)'
33 writeproperty('obj7', '1');                   // store taint in element 2 of array 'obj7'
34 pop();                                        // discard taint of assignment expression
                                                                                    l4
35 readvar('frame3:a');                          // load taint for variable 'a'
36 push(false);                                  // taint value for 'reduce'
37 readproperty('obj13','reduce');               // read taint of property 'reduce' of object 'obj13'
38 push(false);                                  // taint value for function literal
39 push(false);                                  // taint value for string literal ' '
40 push(false);                                  // taint value for receiver object
                                                                                    l5
41 pop();                                        // discard taint of receiver object
42 initvar('_accum_');                           // initialize variable _accum_
43 pop();                                        // discard taint associated with callback function
                                                                                    l7
44 readvar('_accum_');                           // read taint in variable _accum_
45 push(false);                                  // taint value for arguments array of 'reduce'
46 push(false);                                  // taint value for index '0'
47 readproperty('obj7', '0');                    // read taint of element '0' in array 'obj7'
                                                                                    l9
48 push(false);                                  // taint value for array 'obj17'
49 push(false);                                  // taint value for index '0'
50 readproperty('obj17', '0');                   // load taint for element '0' of array 'obj17'
51 push(false);                                  // taint value for array 'obj17
52 push(false);                                  // taint value for index '1'
53 readproperty('obj17', '1');                   // load taint for element '0' of array 'obj17'
54 binaryop('+');                                // apply binary '+' operator
55 writevar('_ret_');                            // store taint in special variable _ret_
56 pop();                                        // discard taint value at top of the stack
                                                                                    l8
57 readvar('_ret_');                            // load taint in special variable _ret_
58 writevar('_accum_');                          // store taint in variable _accum_
59 pop();                                        // discard taint value at top of the stack
                                                                                    l6
60 readvar('_accum_');                          // load final value in _accum_
61 writevar('_ret_');                            // store taint in special variable _ret_
62 pop();                                        // discard taint value at top of the stack
                                                                                    l3
63 pop();                                        // pop taint of invoked function
64 readvar('_ret_');                            // push returned value upon return from call
65 writevar('frame3:command');                   // store taint in variable 'command'
66 pop();                                        // discard taint of assignment expression
```

Fig. 4. Abstract machine instructions generated for the application of Fig. 2.

error was detected, the string written to standard output is converted to upper case by invoking the native `String.toUpperCase` function (line 7), and printed to the console.

2.4 Abstract Machine

The basic idea behind our taint analysis is that the source code of the original application p is instrumented so that instructions for an abstract machine are emitted during execution. The generated instructions manipulate a stack of abstract values that reflect the taintedness of values on the runtime stack of p. The abstract machine maintains maps that associate abstract values with local variables and object properties, reflecting the taintedness of values stored therein. Abstract values are sets of locations, each represented by a string that identifies a filename and line number. Instructions for the abstract machine include operations such as `push` and `pop` for pushing and popping taint values, `unaryop` and `binaryop` for the evaluation of unary and binary expressions, `initvar`, `readvar`, `writevar` for initializing, reading, and writing taint values associated with local variables, and `initproperty`, `readproperty`, and `writeproperty` instructions for initializing, reading, and writing taint values associated with object properties. In the remainder of this section, we informally discuss some of the steps in generating instructions for the example of Fig. 2. A more complete and precise exposition follows in Section 3.

2.5 Example: Generating Instructions

Fig. 4 illustrates the generation of abstract machine instructions for some of the code fragments that are shown labeled in Fig. 2. For the purposes of this example, it is assumed that command-line arguments (i.e., elements of the array `process.argv`) are *sources* of tainted data, and that the

TABLE 1
Rules for Generating Abstract Machine Instructions

program construct	generated instructions	justification
literal expression $l \in Val$	`push` (`false`)	literals in the code are never tainted
object literal $o \equiv \{ p_1 : e_1, \cdots, p_n : e_n \}$	`initproperty` ($oid(o)$, p_n) $\cdots$ `initproperty` ($oid(o)$, p_1) `push` (`false`)	initialize properties from previously pushed values; the object literal itself is untainted
function expression $f \in Fun$	`push` (`false`)	functions are never tainted
variable read $v \in Str$	`readvar` (v)	push taint value for v
property read $v[w] \in Exp$	`readproperty` ($oid(v)$, $offset(w)$)	read property in object oid bound to v
unary expression $uop\ e \in Exp$	`unaryop` (uop)	apply unary operator to top of stack
binary expression $e_1 bop\ e_2 \in Exp$	`binaryop` (bop)	apply binary operator to 2 topmost elements of stack
declaration **var** $v = e$	`initvar` (v)	initialize variable
implicit declaration of **this**	`push` (`false`)	objects are never tainted
function declaration f	`push` (`false`)	functions are never tainted
function parameter $f(\cdots, v, \cdots)$	`initvar` (v)	initialize function parameter
assignment $v = e$	`writevar` (v)	update taint for variable v
property write $v[w] = e$	`writeproperty` ($oid(v)$, $offset(w)$)	write property in object bound to v
function call $e(e_1, \cdots, e_n)$	`pop` () `readVar` ('`_ret_`')	read special `_ret_` variable for communicating the taint associated with the return value
return e	`writeVar` ('`_ret_`')	use special `_ret_` variable for communicating the taint associated with the return value

Here, it is assumed that instructions have already been generated for shaded syntax fragments.

`exec` function is a *sink*. Here, each source of taint is assumed to be uniquely identified by a program location.

The execution behavior of the generated abstract machine instructions will be discussed in Section 3.1 and summarized in Fig. 8, and the process of generating abstract machine instructions will be presented in Table 1. In the remainder of the discussion about the example, we convey the intuition behind the instruction generation process informally, referring the reader to specific instructions in Fig. 8 to further explain the instruction generation process.

The instructions in the block labeled l_1 are generated as a result of executing line 2, where an array is created and assigned to a variable `a`. First, on line 19, a `push` instruction (see instruction $Ins_{\texttt{push}}$ in Fig. 8) is generated to push the value `false`[6] onto the stack, reflecting the fact that the string literal `'ls '` on line 2 is untainted. Then, on line 20, an instruction `initproperty('obj7', '0')` (see instruction $Ins_{\texttt{initproperty}}$) is generated that has the effect of popping this taint value from the stack and associating it with element 0 of a newly created array that is uniquely identified by an identifier `obj7`. Section 4 will discuss how such object identifiers are obtained.

Next, on line 21 a value `false` is pushed onto the stack to indicate that the array literal itself is untainted as well, and the instruction `writevar('frame3:a')` on line 22 associates this taint value with variable `a`. Here, `frame3` is an identifier that uniquely identifies the run-time instance of variable `a`.

6. The value `false` is used to represent the empty set of locations.

The `writevar` instruction does not pop the stack (see instruction $Ins_{\texttt{writevar}}$ in Fig. 8), reflecting the fact an assignment expression such as the one on line 2 evaluates to the same value as its right-hand side. In this case, the value computed by the assignment is discarded (i.e., not assigned to some other variable), so an additional `pop` instruction is generated on line 23 to discard the corresponding taint value as well.[7]

The instructions in block l_2 are generated as a result of executing the statement `a[1] = process.argv[2]` on line 3, and illustrate how taint is introduced when a source is encountered. These generated instructions reflect reading a tainted value from the array `process.argv` and associating it with element `1` of array `a`. This involves the following steps: (i) pushing the taint value corresponding to the receiver expression `a` onto the stack (line 24), (ii) pushing the taint value corresponding to the index expression `1` onto the stack (line 25), (iii) computing the taint value that is to be written and pushing it onto the stack (lines 26-32), (iv) generating a `writeprop` instruction to associate this value with the specified array element (line 33) without removing it from the stack (see instruction $Ins_{\texttt{writeprop}}$ in Fig. 8), and (v) discarding the value computed by the assignment (line 34). [8]

7. Such `pop` instructions are omitted if assignments are chained. For example, consider an expression `a = b = c`, where variable `c` holds a tainted value. Here, the tainted value produced by the nested assignment expression `b = c` must be propagated to the variable `a`.

8. The purpose for leaving the current value on the stack in step (iv) and generating a separate `pop` instruction in step (v) is to enable the uniform modeling of chained property assignments of the form `d = b.c = a` in the instruction generation process.

Here, step (iii) reflects the evaluation of the expression `process.argv[2]`, which involves retrieving the taint value associated with property `argv` of object `process` (lines 26–28), and retrieving the taint value associated with element 2 of the array (lines 29–30). At this point, taint is introduced by discarding the previously read taint value (line 31) and pushing a taint value `'(example.js:3:8)'` (line 32).

The instructions in block l_4 are generated when the call to `Array.prototype.reduce` on line 5 is encountered on array `a`, and reflect reading the taint value for the property `reduce` in the prototype of object `a` (lines 35–37), and pushing taint values `false` for the two arguments and the receiver (lines 38–40).

2.6 Native Functions

At this point, instructions need to be generated for operations performed by the function `Array.prototype.reduce`. However, since this function is implemented natively, an instrumentation-based technique such as ours cannot observe these operations. To handle such cases, our approach relies on manually crafted *models* for native functions. A key challenge that arises here is that native functions such as `Array.prototype.reduce` invoke callbacks, so operations performed by native functions may be interleaved with operations performed by (native or non-native) callback functions.

To illustrate how we handle such cases, consider that `reduce` traverses an array from beginning to end and repeatedly invokes a callback function `cb` to two arguments, x and y. Here, x is bound to an "accumulator" that is initialized with the second argument passed to `reduce`, and y is bound to the array element that is currently being visited. At the end of each iteration, the value computed by `cb` is assigned to the accumulator. When the end of the array is reached, `reduce` returns the final value of the accumulator. To track the flow of taint precisely, we must track the flow of taint: (i) from the second argument of `reduce` to the accumulator, (ii) from the accumulator to the first argument of the callback function `cb`, (iii) from the return value of `cb` to the accumulator, and (iv) from the accumulator to the return value of `reduce`. To account for this, our native models consist of 4 parts, consisting of instructions to be emitted: (i) upon invocation of a native function, (ii) before a callback function starts executing, (iii) when a callback function has finished executing, and (iv) upon return from a native function. Note that this represents the most general case. In practice, many native models do not require all four of these components.

Returning to the example, the instructions in block l_5 capture the propagation of taint from the 2nd argument of `reduce` to parameter `x` of callback `cb`, which involves (i) discarding the taint of the receiver (line 41), (ii) initializing a special variable `_accum_` with the taint value associated with the accumulator (line 42), and (iii) discarding the taint value associated with the callback function itself (line 43).

The block of instructions labeled l_7 is generated upon entry to the callback function `cb` and includes instructions for reading the taint value associated with the accumulator (line 44) and the taint value associated with the function literal that is provided as the first argument to the `reduce` invocation (lines 45–47).

At this point, execution reaches the callback function `cb`. This leads to the generation of the instructions in block l_9, which involves retrieving the taint values associated with variables `x` (lines 48–50) and `y` (lines 51–53),[9] and applying the binary + operator (line 54), which has the effect of computing the union of the sets of taint locations and pushing the resulting value onto the stack. Line 55 stores this value in a special variable `_ret_`, which is used to model the passing of return values from a callee to its caller. Line 56 discards the taint value at the top of the stack.

The block of instructions labeled l_8, is generated upon exit from the callback function `cb` and includes instructions for reading special variable `_ret_` containing the taint associated with the value that was just returned by the callback function (line 57), and using this value to update the accumulator (line 58).

The instructions in block l_6 are generated just before execution returns from `reduce`, and this involves reading the final value in the accumulator (line 60) and storing it into the special variable `_ret_` (line 61). In block l_3, the value stored in `_ret_` is retrieved (line 64) and then used to update the taint value associated with the variable `command` (line 65). Additional instructions (not shown) are generated for the call to `exec` (lines 6–8). Since `exec` is specified as a sink for our analysis, a `report` instruction is issued that generates a taint report.

2.7 Indirect Taint Sources

We also explored how our technique can be used to detect possible privacy leaks in *personal-watchface*, an open-source JavaScript application for the Samsung Gear S2 SmartWatch.[10] This application requests permission to access health-related data so that it can visualize the heart rate on the screen and vibrate when the target heart rate is exceeded. Furthermore, it requests internet permission to retrieve news headlines from `https://arstechnica.com/`, which it also displays on the screen.

In principle, this combination of permissions could pose a privacy risk, because it enables an application to transmit health-related data to a third-party site by embedding it in an HTTP request. To see how such privacy leaks might arise, consider Fig. 5, which shows selected fragments of the source code of *personal-watchface*. When the application is started, function `activate` (lines 59–64 in file `main.js` is invoked. On line 61, this function registers a callback function `updateHeartRate` (lines 67–79) as a listener for the heart-rate monitor. Hence, from this point onwards, function `updateHeartRate` is invoked periodically by the runtime system with an argument `hrmInfo` that has a property `heartRate` containing the latest heart rate. Lines 71–76 show some logic that examines the heart rate to determine when the device should vibrate. In file `rss.js`, a function `getDataFromXML` constructs an XMLHttpRequest on line 89 to request news items from the `https://arstechnica.com/`. In this case, the URL originates from line 81, where it can be seen that no confidential information is transmitted as part of the HTTP request. However, it is easy to see how a privacy leak could be introduced, e.g., by changing line 89 to:

9. As identified by their position in the `arguments` array.

10. Available from https://github.com/offbynull/personal-watchface

File: main.js

```
// invoked upon activation of the application
function activate() {
  ...
  tizen.humanactivitymonitor.start('HRM',
                              updateHeartRate);
  ...
}
// invoked when new heart-rate information
// becomes available
function updateHeartRate(hrmInfo) {
  ...
  lastHeartRate = hrmInfo.heartRate;
  ...
  if (hrmInfo.heartRate < 103.95) {
    elem_outter.style.color = '';
    if (oldHeartRate >= 103.95) {
      navigator.vibrate(2000);
    }
  }
  ...
  oldHeartRate = hrmInfo.heartRate;
}
```

File: rss.js

```
var XML_ADDRESS =
     "http://feeds.arstechnica.com/
              arstechnica/index/",
   XML_METHOD = "GET",
...
function getDataFromXML() {
  var xmlhttp,
  ...
  xmlhttp = new XMLHttpRequest();
  xmlhttp.open(XML_METHOD, XML_ADDRESS, true);
```

Fig. 5. Code fragments from *personal-watchface*.

```
xmlhttp.open(XML_METHOD,
XML_ADDRESS+'?hr='+lastHeartRate, true);
```

Note that, in this scenario, taint sources have a slightly different form from the situation in Fig. 2. In that example, taint originated in elements of the array `process.argv`, i.e., in properties of objects stored in the variable `argv`. In the Tizen example, however, tainted values originate in properties of objects bound to arguments of callback functions such as `updateHeartRate` that are invoked by the runtime system, where the name of such functions is determined at run time. In other words, specifying such taint sources involves a level of indirection: the tool user must specify the name of a function such as `tizen.humanactivitymonitor.start` that registers a callback function for which taint originates in arguments of that function when it is invoked by the runtime system. Fig. 6 shows how support for such *indirect taint sources* is expressed. This taint specification lists `tizen.humanactivitymonitor.start` as an *indirect* taint source. This means that all properties of arguments of callbacks passed to this function are assumed to be tainted. In other words, from this specification, *Ichnaea* infers that if a function f is passed as a callback to `tizen.humanactivitymonitor.start`, then all properties of objects passed in calls to f are tainted.

3 Taint Analysis

For ease of exposition, we begin by presenting our technique for a core subset of JavaScript, for which a grammar is shown in Fig. 7. In defining this subset, we make several simplifying assumptions (e.g., that functions always return values using explicit `return` statements) and we omit features such as control flow constructs, exception handling, property access using the "." operator, and arrays. The features excluded here are handled similarly as ones discussed. Details on how to handle JavaScript language features outside of the core subset are discussed in Section 3.3.

```
Spec.taintSpec = {
  "source" : [
    {"name":"tizen.humanactivitymonitor.start"
     "direct":false}
  ],
  "sink" : [
    {"name": "XMLHttpRequest.prototype.open"}
  ]
};
```

Fig. 6. Taint specification for *personal-watchface*.

3.1 Abstract Machine

Fig. 8 defines the instruction set for the Abstract Machine. The abstract machine operates on a stack of taint values that reflect the "taintedness" of values on the runtime stack. Here, taint values consist of sets of source locations (each identified by a file name and position within that file). The value `false` is used as a shorthand to refer to the empty set of locations. The abstract machine also maintains maps that associate taint values with local variables and object properties, reflecting the taintedness of values stored therein.

The instruction set for the abstract machine includes operations `push` and `pop` for pushing a taint value onto the stack, and popping it off the stack, respectively. The `unaryop` instruction pops the taint value from the top of the stack and applies an operator-specific function to it (e.g., applying the unary-plus operator to a taint value results in the same taint value), and pushes the resulting taint value onto the stack. Similarly, the `binaryop` operator pops two taint values from the stack, applies an operator-specific function to it (e.g., applying the binary string-concatenation operator computes a new taint value using set-union), and pushes the resulting value onto the stack.

The `initvar`, `readvar`, `writevar`, and `setvar` instructions serve to initialize, read, write and set the taint values associated with local variables. In particular, `initvar(`v`)` creates a new map entry for the taint value associated with variable v, pops the stack, and stores the popped value in it. Likewise, `readvar(`v`)` reads the taint value associated with v and pushes it onto the stack, and `writevar(`v`)` stores the value at the top of the stack in the map entry for v. `setvar(`v`,` val`)` sets the taint value of v to val. Similarly, `initproperty`, `readproperty`, and `writeproperty` initialize, read, and write the taint values associated with object properties. Each of these instructions takes

Str	::=	$\cdots$	(string literals)
Num	::=	$\cdots$	(numeric literals)
Bool	::=	`true` | `false`	
Obj	::=	"{" *Id* ":" *Exp*, $\cdots$, *Id* ":" *Exp* "}"	(object literals)
Fun	::=	**function** *Id* "(" *Id*, $\cdots$, *Id* ")" "{" *Stmt* $*$ "}"	(functions)
uop	::=	$\cdots$	(unary operators)
bop	::=	$\cdots$	(binary operators)
Val	::=	*Str* | *Num* | *Bool* | **undefined** | **null**	(values)
Exp	::=	*Val* |	(literal expression)
		Obj |	(object literal)
		Fun |	(function expression)
		Str |	(variable read)
		Id "[" *Exp* "]" |	(property read)
		uop Exp |	(unary expression)
		Exp bop Exp	(binary expression)
Stmt	::=	**var** *Id* "=" *Exp* |	(variable declaration)
		Fun |	(function declaration)
		Id "=" *Exp* |	(assignment)
		Id "[" *Id* "]" "=" *Exp* |	(property write)
		Exp "(" *Exp*, $\cdots$, *Exp* ")" |	(function call)
		return *Exp* |	(function return)
		Stmt ";" *Stmt*	(sequencing)

Fig. 7. Syntax for a core subset of JavaScript.

TVAL	::= $\{\cdots\}$ | `false`	(sets of source locations)
VNAME	::= *Id*	(variable names)
PNAME	::= *Id*	(property names)
OID	::= *Id*	(object identifiers)

Ins::=

`push`(*TVAL*) |	(push constant onto stack)	($Ins_{\texttt{push}}$)
`pop` |	(pop stack and ignore value)	($Ins_{\texttt{pop}}$)
`unaryop`(*uop*)	(pop stack, apply unary operator, push result)	($Ins_{\texttt{unaryop}}$)
`binaryop`(*bop*) |	(pop top two elements, apply binary operator, push result)	($Ins_{\texttt{binaryop}}$)
`initvar`(*VNAME*) |	(pop stack, initialize variable with popped value)	($Ins_{\texttt{initvar}}$)
`readvar`(*VNAME*) |	(push current value of variable)	($Ins_{\texttt{readvar}}$)
`writevar`(*VNAME*) |	(write value at top of stack into variable)	($Ins_{\texttt{writevar}}$)
`setvar`(*VNAME*, *TVAL*) |	(store value into variable)	($Ins_{\texttt{setvar}}$)
`initproperty`(*OID*, *PNAME*) |	(pop stack and initialize object property with popped value)	($Ins_{\texttt{initprop}}$)
`readproperty`(*OID*, *PNAME*) |	(push value of object property)	($Ins_{\texttt{readprop}}$)
`writeproperty`(*OID*, *PNAME*) |	(write value at top of stack into object property)	($Ins_{\texttt{writeprop}}$)
`deleteop`(*OID*, *PNAME*) |	(pop stack, delete value of object property, push result)	($Ins_{\texttt{deleteop}}$)

Fig. 8. Abstract machine instructions.

two arguments: an object identifier (OID), which uniquely represents an object during the observed program execution, and a string value representing the name of the property being accessed. The process of obtaining these object identifiers is briefly discussed in Section 4. The `deleteop` instruction is emitted when a property is removed from an object using JavaScript's `delete` operator. This instruction pops the taint of the operand from the stack, removes the entry for the deleted object property from the taint map, and pushes the resulting taint value onto the stack.

3.2 Generating Instructions

Table 1 shows, for each language construct s under consideration, the abstract machine instructions that are generated as a side-effect of executing s. In these rules, it is assumed that, for complex expressions and statements, the generation of abstract machine instructions for subexpressions has already taken place (such subexpressions are shown in grey in the table). For example, when a numeric literal value "17" is encountered during execution, an instruction `push(false)` is emitted, indicating that a non-tainted[11] value is currently at the top of the stack. Similarly, executing a read-reference to a variable `x` will emit an instruction `readvar(x)`, resulting in pushing the taint value associated with `x` onto the stack. In the case of executing a binary expression `v + w`, the rule shown in Table 1 assumes that the subexpressions `v` and `w` have already executed, so the top two elements of the stack will contain the taint values associated with these variables. The execution of the + expression itself will result in issuing a `binaryop('+')` instruction.

Finally, we consider the execution of function calls and return statements. Executing a statement `return` e (for some expression e) assumes that subexpression e has already executed, causing the top of the stack to contain the

11. Here, it is assumed that literal values in the program's source code are untainted.

taint value associated with that expression. The execution of the return statement itself is modeled by emitting an instruction `writevar('_ret_')`, where `_ret_`is a special variable that we use to the store the taint associated with the return value temporarily. This variable is read in the calling function using a `readvar` instruction that is emitted when the function call returns.

3.3 Other JavaScript Features

While Table 1 only covers a small subset of JavaScript, our implementation covers the ECMAScript 5 language, except for some minor limitations imposed by the Jalangi framework (e.g., Jalangi does not currently support `strict` mode at the file level). We briefly discuss how some key features are handled:

- *arrays.* In JavaScript, objects and arrays behave almost identically. Accordingly, our abstract machine does not distinguish between array and object access and handles them in a uniform manner using a single set of instructions. In each case, we rely on the fact that we can precisely identify the array index or object property being accessed at run time, and generate property access instructions accordingly. Property and array accesses using the `.` and the `[···]` operators are handled identically.
- *getters and setters.* Getters and setters (specified using `get/set` syntax in JavaScript) enable programmers to bind an object property to a function so that the function will be invoked when that property is looked up or assigned a value. Instead of treating such accesses as a regular property read/write, we model them as function invocation where the base object, arguments, and return value of the invocation are modeled in accordance with the semantics of the operation.
- *apply and call*. These frequently-used native methods allow programmers to explicitly set the `this` value for a target function invocation. Calls to these methods are handled as an invocation to the target function where the modeling of `this`, arguments and return value reflects the actual semantics.
- *eval.* We treat `eval` as a sink in our analysis. However, our analysis is able to report all taint flows across the program, not just the first taint flow that reaches any sink. Therefore, our analysis also needs to track any taint flow in the code generated by the eval. The `eval` construct is treated as the execution of an additional script where code inside `eval` is instrumented normally.
- *exceptions.* The data flow of exception objects in `try`, `catch`, and `finally` constructs is handled similarly as function return values, using a special variable `_throw_`. In particular, an instruction `writevar('_throw_')` is issued when a `throw` statement is encountered, and instructions `readvar('_throw_'); initvar(`v`)` are generated when an exception handler `catch(`v`)` is executed.
- *arguments.* Within each JavaScript function, `arguments`, a special array-like object, is available as a local variable that corresponds to the arguments passed to that function. A function's argument can therefore be accessed either by its name or by its index into the `arguments` array. Our analysis models each argument access as an access of the `arguments` object in order to have a uniform representation of arguments in the abstract machine.
- *arguments in function calls.* In JavaScript, functions may be invoked with more or fewer arguments than they are declared with. Missing arguments are bound to the untainted value `undefined` and are accounted for in the taint analysis for by pushing dummy taint values. Extra arguments can be accessed via the `arguments` array and their taint is modeled as discussed above.
- *asynchronous callbacks.* Asynchronous callbacks are frequently used in JavaScript, e.g., for I/O, event handling and timers. Several functions native to JavaScript (e.g., `setTimeout`) and Node.js (e.g., `writeFile`) define one of their parameters as an asynchronous callback. To handle this, a mapping is maintained between native functions and the functions that they call back asynchronously. This mapping is used to emit the appropriate instructions before and after the callback's execution.
- *for .. in loops.* In JavaScript, a loop of the form `for` v `in` o construct enables one to iterate through properties in an object o, where the loop variable v assumes the name of the next enumerable property in each iteration. Assuming that object o has properties with names $p_1, \ldots, p_n$, we model this by emitting instructions of the form `readproperty(`o`,` p_i`);writevar(`v`)` at the beginning of each iteration of the loop (for $0 \leq i \leq n$).

4 Implementation

We implemented the technique in a tool called *Ichnaea*, on top of Jalangi,[12] a popular instrumentation framework for JavaScript. *Ichnaea* relies on Jalangi to attach hooks at various points during program execution and generate instructions for the abstract machine in accordance with the rules in Table 1. Jalangi's support for shadow memory enables us to associate object identifiers (e.g., `obj11` in Fig. 2) with objects and arrays, and to properly account for different instances of a program variable in different scopes (e.g., `frame3` in Fig. 2). Our analysis is implemented in about 4.5 KLOC of JavaScript code. The abstract machine is implemented as a domain-specific language that is embedded in JavaScript, thus allowing the instructions to be executed by any JavaScript engine. It comprises 321 lines of JavaScript code.

We created models for approximately 90 native functions that are referenced in the NPM modules used in our evaluation (see Section 5) and/or in the *personal-watchface* application discussed in Section 2. This includes functions on arrays, strings, functions, and objects as well as some native functions from the Tizen runtime. A considerable number of these are higher-order functions that take callbacks.[13] Since our models for native functions are based on the

12. We use Jalangi2 (see https://github.com/Samsung/jalangi2), which, similar to Jalangi1, supports shadow memory.

13. In principle, a native function may invoke multiple callback functions. Our implementation currently only supports the case where a single callback function is used since we have not observed cases involving multiple callback functions.

```
var arrayReduce = {
    functionName: "Array.prototype.reduce",
    pre: function (name, arrayOID, length, args){
        var model = "pop();";
        if (args.length === 1){
            model += "pop(); push(false); push(false);"
                + "readProperty(\'" + arrayOID + "\', \'0\');"
                + "initVar('_accum_');";
        } else if (args.length === 2){
            model += "initVar('_accum_'); pop();";
        }
        return model;
    },
    post: function (base, args, result){
        return "readVar('_accum_'); writeVar('_ret_'); pop();";
    },
    callbackpre: function(args) {
        var total = args[0], value = args[1],
            index = args[2], arrayObj = args[3];
        var arrayOID = shadowMemory.getObjectID(arrayObj);
        return "readVar('_accum_'); push(false); push(false);"
            + "readProperty(\'" + arrayOID + "\', \'"+ index + "\');"
            + "push(false); push(false); push(false);";
    },
    callbackpost: function() {
        return "readVar('_ret_'); writeVar('_accum_'); pop();";
    }
};
```

Fig. 9. Native model for `Array.prototype.reduce`.

ECMAScript 5 specification [14], the platform-independence of our approach is not compromised provided that the actual implementations of these functions also match their specification.

In general, a native model for a native function f that takes a callback function g as an argument consists of the following four components:

- *pre:* Abstract machine instructions that are emitted immediately after a call to f,
- *post:* Abstract machine instructions that are emitted upon return from a call to f,
- *callbackpre:* Abstract machine instructions that are emitted just before g is entered, and
- *callbackpost:* Abstract machine instructions that are emitted when g is exited.

For many native functions, the full generality of this approach is not required and one or more components can be omitted. Furthermore, note that these models only reflect the native function's impact on propagating taint, and constructing such a "taint model" is generally much less work than creating a model that accurately reflects the function's execution semantics.

As an example, Fig. 9 shows the model for `Array.prototype.reduce`. For this model, the `pre` component initializes a special variable `_accum_` that is used to store the taint associated with intermediate results. Depending on whether the function is invoked with one or two arguments, the accumulator is initialized with either the taint associated with the supplied initial value, or with the taint associated with the first array element. The `post` component loads the final value of `_accum_` and writes it to `_ret_`, to make it available to the caller. The `callbackpre` component of the model reads the taint values associated with the accumulator and with the array element currently being visited and adjusts the stack to account for index value, array object, and receiver. Lastly, the `callbackpost` model reads `_ret_`, containing the value being returned by the callback function, and writes it to `_accum_`.

We modeled DOM APIs to track propagation of taint values across the DOM structure by intercepting each call to a DOM API function, and modeling its effect on a shadow representation. This enables us to capture the taint values that are stored in DOM nodes and retrieved later in the program execution. We rely on a similar abstraction to model the Tizen APIs.

5 Evaluation

In order to evaluate the practicality of our technique, we aim to answer the following research questions:

RQ1: Is *Ichnaea* capable of detecting flows of tainted data that correspond to security vulnerabilities in real JavaScript software?

RQ2: What is the run-time overhead of *Ichnaea*?

RQ3: How large are generated sequences of instructions, and how much time is needed to execute them?

The first research question aims to determine whether *Ichnaea* is an effective tool for detecting data flows from sources where tainted data enters an application to sinks where sensitive operations are performed. The second research question aims to determine by how much execution speed is slowed down due to the instrumentation added by *Ichnaea*. The third research question aims to determine how much space is required by the generated instructions, and how much time is required to execute them.

5.1 Experimental Methodology

As mentioned, the primary use case that we envision for our technique is a situation where a JavaScript application depends on third-party modules or libraries that use functions such as `eval` or `exec`. In such cases, as was illustrated in Section 2, injection vulnerabilities may exist that could be very difficult to detect through manual code examination. We now discuss key aspects of the experimental methodology.

Selecting Subject Applications. Since we are interested in evaluating our technique on real software, we decided to apply *Ichnaea* to 22 modules from the Node Package Manager (NPM),[14] which provides 250,000+ JavaScript packages for the Node.js platform that implement a wide range of functionalities including server-side I/O, Internet of Things, mobile applications, to give just a few examples. It is well-known that NPM modules may suffer from various types of security vulnerabilities, as is evident, e.g., from a large number of advisories on www.nodesecurity.io. Unlike browser-based JavaScript applications that execute in a sandbox, Node.js applications have full access to the underlying file system and operating system. Therefore, command-injection vulnerabilities may cause serious harm in Node.js applications.

Recently, Staicu et al. [2] presented a static analysis and associated runtime monitoring technique for detecting and securing calls to `eval` and `exec` that may be subject to injection vulnerabilities. They conducted a large-scale study of NPM modules in which they detect previously undetected injection vulnerabilities. In our experiments, we

14. See https://www.npmjs.com/

TABLE 2
NPM Modules Used for the Evaluation of *Ichnaea*

npm module	description	version	LOC	files	type	vulnerability
chook-growl-reporter	growl reporter for the chook unit test runner	0.0.1	1,515	21	`exec`	[2]
cocos-utils	utilities for developers of cocos2d-html5 game engine	1.0.0	2,098	35	`exec`	previously unreported
gm	image processing for Node.js	1.20.0	3,539	24	`exec`	http://nodesecurity.io/advisories/54
fish	jQuery of filesystem for Node.js	0.0.0	73	4	`exec`	[2]
git2json	convert git log to json	0.0.1	247	12	`exec`	[2]
growl	growl support for Nodejs	1.9.2	356	4	`exec`	http://github.com/tj/node-growl/issues/60
libnotify	libnotify support for Node.js	1.0.3	97	3	`exec`	http://nodesecurity.io/advisories/20
m-log	rich-text logging support	0.0.1	1,184	21	`eval`	http://github.com/m-prj/m-log/pull/1
mixin-pro	simulate mixin-based inheritance in JavaScript	0.6.6	480	5	`eval`	http://github.com/floatinghotpot/mixin-pro/issues/1
modulify	generate nodejs modules from source code	0.1.0-1	114,798	180	`eval`	http://github.com/matthewkastor/modulify/issues/2
mongo-parse	parser for MongoDB queries	1.0.5	101,268	257	`eval`	http://github.com/fresheneesz/mongo-parse/issues/7
mongoosemask	filter for Mongoose model attributes	0.0.6	34,275	18	`eval`	http://github.com/mccormicka/MongooseMask/issues/1
mongoosify	Javascript library for converting a JSON schema into a Mongoose schema	0.0.3	26,385	421	`eval`	[2]
node-os-utils	operating system utility library	1.0.7	1,120	14	`exec`	previously unreported
node-wos	determine which OS is being used	0.2.3	557	6	`execSync`	none
office-converter	convert office documents into PDF/HTML by invoking `unoconv` command	1.0.2	143	5	`exec`	previously unreported
os-uptime	get operating system's uptime as a date	2.0.1	129	6	`execSync`	none
osenv	look up OS-specific environment settings	0.1.5	206	6	`exec`	none
pidusage	fetch process cpu% and memory usage of a PID	1.1.4	525	7	`exec`	http://nodesecurity.io/advisories/356
pomelo-monitor	tool for monitoring OS and process information	0.3.7	290	7	`exec`	previously unreported
system-locale	get locale from OS	0.1.0	65	3	`execFileSync`	none
systeminformation	system and OS information library	3.42.4	13,155	31	`exec`	none

The columns in the table show (from left to right): the name of the module, brief description of the module's functionality, version number, number of lines of source code (includes code in imported modules), number of files, type of vulnerability (classified in terms of the function that serves as the sink), information where the vulnerability was first reported (here, 'previously unreported' indicates that the vulnerability was not previously reported elsewhere, and 'none' indicates that no vulnerability exists).

apply *Ichnaea* to a subset of the NPM modules in which they report injection vulnerabilities,[15] as well as to some additional NPM modules that we identified by analyzing recent advisories on www.nodesecurity.io. Furthermore, we manually explored GitHub repositories and identified a number of additional NPM modules that invoke functions such as `exec`, `execSync`, and `execFileSync`. Here, we deliberately included some modules in which it is impossible for tainted input values to flow to these functions so that we could confirm that *Ichnaea* does not spuriously report tainted flows in such cases.

Constructing Test Drivers. For each module under consideration, we created a small test driver that invokes the module in a way that triggers the execution of a "sink" function. For modules with real vulnerabilities, we created the test in such a way that the vulnerability was triggered. We then determined whether *Ichnaea* reports any flow of tainted data from the input value to the operation in the module where the injection takes place. These test drivers consist of a call to a function specified in the API of the NPM module in which a "payload" (e.g., a string value containing an embedded "`touch`" command) is passed as an argument to the function. For each test, we check whether the payload is executed (e.g., by checking the timestamp on the `touch`ed file), and check that taint flows are only reported in cases where the injected command reaches the call to `eval` or `exec`. The 'libnotify', 'chook-growl-reporter', and 'office-converter' subjects required the creation of shell commands that are not available on the Mac platform where we conducted our experiments.

Constructing Taint Specifications. For each module under consideration, we created a taint specification in which any string constant occurring in the test driver is a source, and that any call to `eval`, `exec`, `execSync`, and `execFileSync` anywhere in the application is a sink. In other words, no detailed knowledge of the subject application code was required.

5.2 Subject Programs

Table 2 states the characteristics of the 22 NPM modules that we use to evaluate our technique, and that exhibit injection vulnerabilities using a variety of different "sink" functions. The modules also exhibit a range of different programming styles, and are of varying sizes. Furthermore, in the aggregate, these modules make use of a considerable number of native functions involving arrays, strings,

15. Of the subject applications considered by Staicu et al., we exclude the ones that involve data flow via the network or file system. Furthermore, of the 15 modules they consider that do not involve data flow via the network or file system, we exclude two applications that do not run on a Mac, and one that exposed a bug in Jalangi for which we are awaiting a fix.

TABLE 3
Experimental Results

npm module	# exectd. instr.	# gen. instr.	# gen. / # exectd. instr.	time (s) (orig) mean	time (s) (orig) variance	time (s) (*Ichnaea*) mean	time (s) (*Ichnaea*) variance	overhead	trace size (KB)	relative trace size (B/instr)	time (s) (taint) mean	time (s) (taint) variance
chook-growl-reporter	1,340	1,662	1.24	0.100	1.76e-05	0.670	5.82-04	6.70	56	42.79	0.236	2.76e-05
cocos-utils	536	778	1.45	0.102	1.64e-05	0.652	5.81e-05	6.39	32	61.13	0.084	6.62e-06
gm	2,616	4,269	1.63	0.109	2.1e-06	1.486	3.44e-04	13.62	124	48.54	0.096	8.71e-06
fish	143	163	1.14	0.098	7.88e-06	0.311	3.46e-05	3.17	16	114.57	0.079	1.22e-05
git2json	603	876	1.45	0.101	7.73e-06	0.488	5.32e-05	4.83	36	61.13	0.087	1.23e-05
growl	572	704	1.23	0.087	1.19e-05	0.448	4.09e-05	4.57	32	57.29	0.084	1.11e-05
libnotify	194	287	1.45	0.098	1.11e-05	0.327	2.78e-05	3.34	20	105.57	0.079	4.23e-06
m-log	7,848	11,426	1.46	0.092	1.43 e-05	1.169	2.63e-04	12.48	332	43.32	0.113	1.47e-05
mixin-pro	200	250	1.25	0.082	1.85 e-05	0.424	1.20e-05	5.14	20	102.4	0.10	6.54e-06
modulify	19,664	23,822	1.21	0.113	4.22e-05	3.331	1.3e-03	29.42	688	35.83	0.135	2.25e-05
mongo-parse	362	504	1.39	0.087	5.38 e-05	0.631	5.44 e-05	7.28	28	79.20	0.082	2.22e-06
mongoosemask	20,487	25,031	1.22	0.094	5.34e-06	1.972	3.58e-04	21.04	720	35.99	0.137	1.71e-05
mongoosify	72,296	79,353	1.10	0.110	1.65e-05	4.230	2.67e-03	38.42	2355	33.36	0.258	7.54e-05
node-os-utils	722	919	1.27	0.102	1.03e-05	0.836	7.99e-05	8.19	36	51.06	0.085	1.20e-05
node-wos	756	952	1.26	0.094	8.01e-06	0.435	4.16e-05	4.65	36	48.76	0.085	8.04e-06
office-converter	174	184	1.06	0.096	1.34e-04	0.34	6.68e-06	3.53	16	94.16	0.084	1.26e-05
os-uptime	145	162	1.12	0.095	9.96e-06	0.323	3.03e-05	3.4	16	112.99	0.083	8.54e-06
osenv	482	554	1.15	0.098	8.99e-06	0.457	3.32e-05	4.66	28	59.46	0.084	3.29e-06
pidusage	462	619	1.34	0.096	5.78e-06	0.528	2.78e-05	5.44	28	62.06	0.125	1.00e-05
pomelo-monitor	377	434	1.15	0.123	7.79e-06	0.507	1.08e-04	4.13	24	65.19	0.085	2.0e-05
system-locale	175	203	1.16	0.096	6.9e-06	0.339	4.57e-05	3.54	16	93.62	0.082	6.18e-06
systeminformation	9,514	10,726	1.13	0.124	1.65e-05	3.128	1.78e-03	25.24	284	30.57	0.108	2.49e-05

objects, and functions. The number of lines of code reported in the column labeled 'LOC' includes lines with comments and whitespace and includes code in modules imported using the `require` function. The column labeled 'files' counts the number of JavaScript source files. The column labeled 'type' indicates whether the injection vulnerability was due to the use of `eval`, `exec`, `execSync`, or `execFileSync`, respectively, and the last column shows where the vulnerability (if any exists) was reported. Note that this includes four modules in which we identified a previously unreported injection vulnerability, and five modules where a "sink" function is executed but where no vulnerability exists.[16] Several of the modules that we found during our manual search on GitHub relied on ECMAScript 6 features that are not yet supported by Jalangi. We manually refactored these features into equivalent ECMAScript 5 features, and the data reported in Table 2 reflect the refactored code.

5.3 Experimental Results

We ran *Ichnaea* on each test case. For each module with a vulnerability, we confirmed that it reported taint flows from string literals created in the test to the location in the module where the vulnerability was reported. For each module without a vulnerability, we confirmed that *Ichnaea* reports that only untainted data flows to sinks. Table 3 summarizes our experimental results.

The first group of columns in the table show, from left to right, the number of instructions executed in the original test (measured in terms of the number of executed program constructs in the subject program; a subset of these program constructs was listed in Table 1), the number of instructions generated for the abstract machine, and the ratio between the number of generated instructions and the number of executed instructions. The second group of columns in the table show the mean running time (and variance) of the original test, the mean running time (and variance) with *Ichnaea*'s instrumentation, and the runtime overhead, (measured as a slowdown factor obtained by dividing the latter by the former). The column labeled 'trace size' shows the size of the generated executable trace, which includes both the generated instructions and the abstract machine itself. The column labeled 'relative trace size' shows the trace size relative to the number of instructions executed in the original test. The last column, labeled 'time (taint)' shows the time required for executing the generated trace.

All results were computed on an Apple MacBook Pro with a 2.5 GHz Intel Core i7 processor with 16 GB 1,600 MHz DDR3 RAM, running MacOS High Sierra 10.13. We used Node.js version v4.8.4. All reported running times are averages over 10 executions.

Based on the results in Table 3, we are now in a position to answer the research questions:

RQ1: *Ichnaea* is capable of detecting taint flows corresponding to real security vulnerabilities in NPM modules that cover a considerable range of programming styles and native functions. We manually investigated the taint flows reported by *Ichnaea* and confirmed each reported flow of tainted data from a source to a sink. We also confirmed that, on the modules without vulnerabilities, *Ichnaea* reports only untainted data to flow to a sink. In other words, there were no false positives, as could be expected from a precise dynamic analysis. In general, a dynamic

16. In these cases, the argument passed to the sink function is constructed entirely of string literals that occur in the module's source files. In a few cases (e.g., system-locale and osenv) non-trivial concatenation of strings literals that occur in the module's files takes place).

analysis like *Ichnaea*'s may suffer from false negatives if some control-flow paths are not executed. However, we confirmed through manual investigation that no flows of tainted data from sources to sinks went unreported in the executions that we considered.

RQ2: On the NPM modules under consideration, the use of *Ichnaea* slows down execution time by a factor ranging from 3.17x to 38.42x (9.96x on average).

RQ3: On the NPM modules under consideration, the ratio between the number of generated instructions and the number of executed instructions lies within a small range between 1.06 and 1.63. The generated executable trace files range between 16 KB and 2.3 MB, the relative trace sizes range from 30.57 bytes per instruction to 112.99 bytes per instruction, and executing these trace files to produce a report on tainted flows requires less than 0.3 seconds in all cases.

5.4 Threats to Validity

We are aware of several threats to validity.

Selection of Experimental Subjects. The NPM modules used in our evaluation may not be representative of code running on the Node.js platform, or of JavaScript software more generally. Similarly, the flow of taint from values passed into the module to the place where the injection takes place may not be reflective of injection vulnerabilities in general. We have attempted to address this concern by analyzing all NPM modules with such vulnerabilities that we could find and run with our infrastructure, and that did not involve significant additional effort (e.g., modules that require additional software to be installed on a server).

Short Execution Times. The execution times for our test drivers for the NPM modules under consideration are quite short. This may distort the observed runtime overheads (e.g., because of JIT warmup time). We have attempted to address this concern by reporting the average running time of 10 executions and by reporting the variance in the running times that we observed.

Hand-crafted Models for Native Functions. The models created for native functions were constructed manually following a careful review of the specification. It is possible for our models to contain errors, or for the implementation of native functions on some platforms to vary from the officially specified behavior. Such errors could cause false positives and false negatives in the results, in principle. To minimize this risk, our implementation is accompanied by an extensive test suite that covers all native functions used in our experimental subjects.

Potential for Missing Taint Reports. As with any dynamic analysis, the issues reported by *Ichnaea* are limited to code that is executed. Additional issues might occur in code that is not executed, so it is important to develop a comprehensive test suite. In particular, *Ichnaea* may fail to report issues if a test suite fails to cover sources, if it fails to cover sinks, or if it fails to exercise paths from sources to sinks. To gain confidence that there are no false negatives, developers can inspect the code, locate where sources and sinks occur, and ensure that the test suite covers all execution paths from these sources to these sinks.

6 Related Work

There has been a long history of research on information flow analysis going back to the 1970s [15]. Broadly speaking, previous research can be classified as static techniques (see, e.g., [16], [17], [18], [19]) dynamic techniques (see, e.g., [5], [20], [21]), or hybrid static/dynamic techniques (see, e.g., [6], [22], [23]). Below, we focus on dynamic and hybrid information flow analyses for JavaScript.

Austin and Flanagan [4] present an information flow analysis for λ_{info}, a variant of the untyped λ-calculus, in which values are labeled with information flow labels that are ordered in a lattice, and evaluation rules of a standard big-step operational semantics manipulate these labels as program execution proceeds. The approach relies on the no-sensitive-upgrade check [24] to correctly handle implicit flows without explicitly reasoning about the behavior of branches that are not executed. Later work [25] extended the approach to Featherweight JavaScript, a small language with objects, arrays and dynamic prototype chains. Austin and Flanagan [26] also propose a more permissive approach in which an additional security label represents partially leaked data. Such labels are introduced when variables are assigned in branches controlled by private information, and execution can proceed until a value with such a label is used in a conditional branch.

Hedin and Sabelfeld [27] present a dynamic information-flow analysis for a core subset of JavaScript, including higher-order functions and objects. The approach handles both explicit and implicit flows and takes the form of a big-step operational semantics in which values have associated security labels. Later, Hedin et al. [28] extended the work to the full non-strict ECMAScript 5 language in the context of JSFlow, a specialized JavaScript interpreter, which itself is implemented in JavaScript, and Snowfox, a Firefox extension that uses JSFlow as the execution engine for web pages. Similar to our work, Hedin et al. rely on models for native functions, and observe that "deep" models must be defined for methods defined on arrays. However, they do not discuss in detail how their models are defined (e.g., how taint is propagated from the array itself to the arguments and return values of callback functions passed to the methods). Hedin et al. report that JSFlow is two orders of magnitude slower than a fully JITed JavaScript engine. Currently, JSFlow does not run on the Node.js applications that we consider due the absence of models for Node.js native functions. Later work by Hedin et al. [29] and Sjösten et al. [30] is concerned with developing concise models for tracking information flow in libraries, focusing on a small functional language in each case.

Several dynamic approaches rely on modification of a browser. Kerschbaumer et al. [7] present CrowdFlow, a specialized browser built on top of WebKit that associates different taint labels with data originating from different domains. Potential information-flow violations are reported when values originating from different domains are used in HTTP requests. CrowdFlow distributes the tracking of

information flows (and the associated run-time overhead) over a crowd of users. Dhawan and Ganapathy [31] implement a dynamic information flow analysis for browser extensions in the Firefox browser. Their approach tracks both explicit and implicit flows, but is unable to reason about code in branches that are not executed.

Jang et al. [32] present a dynamic information flow analysis for JavaScript that was implemented in the Chrome browser by rewriting abstract syntax trees. The AST rewriting involves boxing and unboxing objects, which is not required by our technique. Jang et al. report on a large-scale empirical study that demonstrates the existence of privacy-violating flows reflecting information about users' browsing behavior in several popular sites. Jang et al. did not implement taint tracking for native methods such as `Array.prototype.join`, causing false negatives in their analysis. At the time of writing this paper, their tool was no longer available.

Kannan et al. [33] propose *virtual values*, which extend JavaScript proxies with support for primitive values. They argue that, using virtual values, dynamic information flow analyses can be implemented without modification of a JavaScript engine. However, in the absence of support for this feature, the Sweet.js macro system [34] is used to introduce proxies where primitive values are used.

Saoji et al. [35] present a dynamic taint analysis for JavaScript that is *precise* in the sense that taint is tracked at the level of individual characters in strings. An API is provided to programmers for tainting, untainting, and checking the taintedness of character regions within strings. The technique has been implemented by modifying Mozilla's Rhino JavaScript engine, and Saoji et al. present performance experiments on small benchmarks from the SunSpider suite, showing a modest increase in overhead compared to coarse-level taint tracking.

Several information flow analyses that are implemented using browser modification rely on local static analysis to reason about implicit flows. Vogt et al. [36] implement one such analysis in the context of Firefox, by extending the semantics of bytecode instructions to propagate taint. In this work, a simple intraprocedural static analysis is used to conservatively overapproximate indirect control dependences that occur in branches that were not executed. Additional approximations serve to track indirect flows (e.g., writing a tainted element to an array causes the entire array to be tainted to ensure that methods such as `length` return a tainted value). Just et al. [6] and Bichhawat et al. [37] report on similar information flow analyses for JavaScript that are implemented in WebKit and that also use static analysis to reason about implicit flows.

Chudnov and Naumann [38] and Santos and Rezk [39] present approaches that, like ours, rely on shadow memory representations of variables and objects to keep track of taint. However, both of these works only consider small idealized subsets of JavaScript and do not report on concrete tools or experiments, or on the pragmatics of dealing with native code. In later work, Chudnov and Naumann [8] present a platform-independent information flow analysis that performs a whole-program transformation to wrap primitive values in "boxes" in which security labels are stored. Operations are rewritten to box and unbox values as needed and calls to API functions are re-routed through API facades. These transformations become quite complex and involve introduction of the `with` construct, and consolidation of all JavaScript code into a single file. The use of `eval` requires run-time instrumentation and is currently handled using an HTTP proxy server. While Chudhov and Naumann discuss some of the challenges associated with native method calls, they do not discuss the handling of higher-order functions such as `Array.prototype.reduce` for which taint tracking requires careful modeling of callbacks. They implemented their analysis in a tool called JEST and report on case studies involving security-related issues in mashup applications that they manually constructed and on performance experiments involving benchmarks from the SunSpider and Kraken suites for which they report running times 101 to 364 times slower than the original code. Applying Chudnov and Naumann's tool to the Node.js applications that we consider would require the creation of API facades for all APIs used in these applications, and would involve significant effort.

Chugh et al. [22] present a staged information-flow analysis for a subset of JavaScript in which a static analysis constructs a set of constraints for tracking direct and indirect information flows. Then, a residual policy is inferred by solving these constraints and enforced by way of run-time checks. The paper mentions an implementation of the approach as an extension for the Firefox browser, but this appears to be no longer available.

Wei and Ryder [23] gather dynamic execution information from a set of executed tests using TracingSafari, an instrumented version of WebKit. From this, a program is constructed that represents the observed calling structure and that is free from dynamic features such as `eval`. A static taint analysis is then applied to this program. At the time of writing this paper, Wei and Ryder's tool is no longer available.

With the notable exception of [8], all of the works discussed above that support the full JavaScript language involve the construction of a specialized interpreter, modification of a browser or JavaScript engine, or addition of nontrivial new features to JavaScript. Our technique is platform-independent, does not require complex code transformations, and has been applied successfully to detect known vulnerabilities in modules for the Node.js platform.

7 Conclusions and Future Work

We presented a platform-independent dynamic taint analysis for JavaScript. Our technique instruments a JavaScript application so that, as a side effect of program execution, instructions for an abstract machine are emitted. Executing these instructions produces a report on observed taint flows in the application. Our approach has two key advantages: (i) it is platform-independent and can be used with any JavaScript engine, and (ii) it is capable of tracking taint on primitive values without requiring boxing. Furthermore, higher-order native functions can be accommodated using models, it enables various deployment scenarios (the generated instructions can be executed alongside normal program execution, or they can be transmitted for offline execution), and it handles situation where taint originates from indirect sources.

We implemented the technique in a tool called *Ichnaea*, consisting of approximately 4.5 KLOC of JavaScript code, and demonstrated its practicality by using it to detect taint flows corresponding to known injection vulnerabilities in 22 modules for Node.js. On these modules, we observed runtime overheads ranging from 3.17x to 38.42x compared to uninstrumented execution. We also demonstrated how *Ichnaea* can be used to determine privacy leaks in Tizen apps for the Samsung Gear S2 smartwatch.

While we have not explored the tracking of implicit flow, we believe that its implementation would be largely similar to that of tracking explicit flows. In its essence, our technique relies on a shadow memory representation to associate a taint value with each value stored in memory. Whenever a JavaScript instruction reads/writes to memory, we emit instructions that update the corresponding taint values in shadow memory accordingly. Right now, we only emit instructions that track taint flows that correspond to data flow in the JavaScript code. To track implicit flows, additional instructions would have to be emitted that have the effect of tainting additional values in shadow memory when the JavaScript code executes a branch that depends on tainted data. Doing this would require emitting instructions that signify that a branch is entered or exited, which could be accomplished by way of a small extension to the instruction set for the abstract machine. Additional implicit flows may arise in specialized settings (e.g., the value of the `length` property of an array is implicitly modified by the run-time system when the size of an array changes). Such flows could be tracked by emitting additional instructions to taint the `length` property whenever tainted values are written to an array.

Other directions for future work include support for flexible taint propagation policies that allow the user to customize the taint propagation behavior of operators, and support for user-specified sanitization functions that have the effect of erasing taint from values they return.

Acknowledgments

Alena Sochurkova's work was carried out while she was a student at Czech Technical University in Prague, and was supported by funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 695412). Frank Tip was supported in part by NSF grant CCF-1715153. Most of the work of Frank Tip was carried out during his employment at Samsung Research America. The work of Koushik Sen was carried out during his employment at Samsung Research America.

References

[1] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do - A large-scale study of the use of eval in JavaScript applications," in *Proc. 25th Eur. Conf. Object-Oriented Program.*, 2011, pp. 52–78.

[2] C.-A. Staicu, M. Pradel, and B. Livshits, "SYNODE: understanding and automatically preventing injection attacks on NODE.JS," in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018.

[3] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "You are what you include: Large-scale evaluation of remote JavaScript inclusions," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 736–747.

[4] T. H. Austin and C. Flanagan, "Efficient purely-dynamic information flow analysis," in *Proc. Workshop Program. Lang. Anal. Secur.*, 2009, pp. 113–124.

[5] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Symp. Operating Syst. Des. Implementation*, 2010, pp. 393–407.

[6] S. Just, A. Cleary, B. Shirley, and C. Hammer, "Information flow analysis for JavaScript," in *Proc. 1st ACM SIGPLAN Int. Workshop Program. Lang. Syst. Technol. Internet Clients*, 2011, pp. 9–18.

[7] C. Kerschbaumer, E. Hannigan, P. Larsen, S. Brunthaler, and M. Franz, "CrowdFlow: Efficient information flow security," in *Proc. 16th Inf. Flow Secur. Conf.*, 2013, pp. 321–337.

[8] A. Chudnov and D. A. Naumann, "Inlined information flow monitoring for JavaScript," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 629–643.

[9] A. Zavou, G. Portokalidis, and A. D. Keromytis, "Taint-exchange: A generic system for cross-process and cross-host taint tracking," in *Proc. 6th Int. Workshop Advances Inf. Comput. Secur.*, 2011, pp. 113–128.

[10] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, P. G. Lowney, S. Wallace, V. J. Reddi, and K. M. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 190–200.

[11] K. Sen, S. Kalasapur, T. G. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for JavaScript," in *Proc. Joint Meeting Eur. Softw. Eng. Conf./ ACM SIGSOFT Symp. Found. Softw. Eng.*, 2013, pp. 488–498.

[12] H. Jaygarl, C. Luo, Y. Kim, E. Choi, K. Bradwick, and J. Lansdell, *Professional Tizen Application Development*. Hoboken, NJ, USA: Wiley, 2014.

[13] M. Cantelon, M. Harter, T. J. Holowaychuk, and N. Rajlich, *Node.JS in Action*. Shelter Island, NY, USA: Manning Publications, 2014.

[14] ECMAScript 5 Language Specification, 2011. [Online]. Available: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[15] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[16] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: Effective taint analysis of web applications," in *Proc. 30th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2009, pp. 87–97.

[17] Y. Liu and A. Milanova, "Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows," in *Proc. 14th Eur. Conf. Softw. Maintenance Reengineering*, 2010, pp. 146–155.

[18] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable JavaScript," in *Proc. 20th Int. Symp. Softw. Testing Anal.*, 2011, pp. 177–187.

[19] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, Art. no. 29.

[20] J. A. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proc. ACM/SIGSOFT Int. Symp. Softw. Testing Anal.*, 2007, pp. 196–206.

[21] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. 31st IEEE Symp. Secur. Privacy*, 2010, pp. 317–331.

[22] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner, "Staged information flow for JavaScript," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2009, pp. 50–62.

[23] S. Wei and B. G. Ryder, "Practical blended taint analysis for JavaScript," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 336–346.

[24] S. A. Zdancewic, "Programming languages for information security," Department of Computer Science, PhD thesis, Cornell Univ., Ithaca, NY, 2002.

[25] T. H. Austin, T. Disney, C. Flanagan, and A. Jeffrey, "Dynamic information flow analysis for featherweight JavaScript," Univ. California at Santa Cruz, Santa Cruz, CA, Tech. Rep. UCSC-SOE-11–19, 2011.

[26] T. H. Austin and C. Flanagan, "Permissive dynamic information flow analysis," in *Proc. Workshop Program. Lang. Anal. Secur.*, 2010, Art. no. 3.
[27] D. Hedin and A. Sabelfeld, "Information-flow security for a core of JavaScript," in *Proc. 25th IEEE Comput. Secur. Found. Symp.*, 2012, pp. 3–18.
[28] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld, "JSFlow: Tracking information flow in JavaScript and its APIs," in *Proc. 29th Annu. ACM Symp. Appl. Comput.*, 2014, pp. 1663–1671.
[29] D. Hedin, A. Sjösten, F. Piessens, and A. Sabelfeld, "A principled approach to tracking information flow in the presence of libraries," in *Proc. 6th Int. Conf. Principles Secur. Trust, Held as Part Eur. Joint Conf. Theory Practice Softw.*, 2017, pp. 49–70.
[30] A. Sjösten, D. Hedin, and A. Sabelfeld, "Information flow tracking for side-effectful libraries," in *Proc. 38th IFIP WG 6.1 Int. Conf. Formal Techn. Distrib. Objects Components Syst., Held as Part 13th Int. Federated Conf. Distrib. Comput. Techn.*, 2018, pp. 141–160.
[31] M. Dhawan and V. Ganapathy, "Analyzing information flow in JavaScript-based browser extensions," in *Proc. 25th Annu. Comput. Secur. Appl. Conf.*, 2009, pp. 382–391.
[32] D. Jang, R. Jhala, S. Lerner, and H. Shacham, "An empirical study of privacy-violating information flows in JavaScript web applications," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 270–283.
[33] P. Kannan, T. H. Austin, M. Stamp, T. Disney, and C. Flanagan, "Virtual values for taint and information flow analysis," in *Proc. Workshop Meta-Program. Techn. Reflection*, 2016.
[34] T. Disney, N. Faubion, D. Herman, and C. Flanagan, "Sweeten your JavaScript: Hygienic macros for ES5," in *Proc. 10th ACM Symp. Dynam. Lang.*, 2014, pp. 35–44.
[35] T. Saoji, T. H. Austin, and C. Flanagan, "Using precise taint tracking for auto-sanitization," in *Proc. Workshop Program. Lang. Anal. Secur.*, 2017, pp. 15–24.
[36] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Krügel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2007.
[37] A. Bichhawat, V. Rajani, D. Garg, and C. Hammer, "Information flow control in WebKit's JavaScript bytecode," in *Proc. 3rd Int. Conf. Principles Secur. Trust, Held as Part Eur. Joint Conf. Theory Practice Softw.*, 2014, pp. 159–178.
[38] A. Chudnov and D. A. Naumann, "Information flow monitor inlining," in *Proc. 23rd IEEE Comput. Secur. Found. Symp.*, 2010, pp. 200–214.
[39] J. F. Santos and T. Rezk, "An information flow monitor-inlining compiler for securing a core of JavaScript," in *Proc. 29th IFIP TC 11 Int. Conf. ICT Syst. Secur. Privacy Protection*, 2014, pp. 278–292.

Rezwana Karim received the PhD degree from Rutgers University, in 2015. She is currently a senior research engineer at Samsung Research America in Mountain View, CA. Her research interest primarily lies in improving security and quality of software using techniques from program analysis and software engineering. Previously she has worked on the domain of UI technologies, network security and ubiquitous computing.

Frank Tip received the PhD degree from the University of Amsterdam, in 1995. He is currently a professor and associate dean for Graduate Programs with the College of Computer and Information Science, Northeastern University. His research interests include program analysis, refactoring, test generation, fault localization, and automated program repair.

Alena Sochůrková received the master's degree from Czech Technical University in Prague, in 2016. She is currently an Android Malware Analyst at Avast in Prague. Her interests include computer security, cryptography and embedded systems.

Koushik Sen received the PhD degree from the University of Illinois at Urbana-Champaign, in 2006. He is currently a professor with the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. His research interest lies in software engineering, programming languages, and formal methods. He is interested in developing software tools and methodologies that improve programmer productivity and software quality.