Linvail: A General-Purpose Platform for Shadow Execution of JavaScript

Laurent Christophe, Elisa Gonzalez Boix, Wolfgang De Meuter, Coen De Roover

Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium
{lachrist,egonzale,wdmeuter,cderoove}@vub.ac.be

***Abstract*—We present Linvail, a novel instrumentation platform for developing dynamic analyses of JavaScript programs. Linvail is particularly well-suited to implementing shadow executions which involve tagging runtime values with analysis-specific data. In contrast to existing instrumentation platforms, Linvail is capable of tracking both tagged objects and tagged primitive values during their entire life-time in a behavior-preserving manner. To demonstrate the expressiveness of our platform, we present the implementation of several state-of-the-art analyses. Our experiments demonstrate that Linvail's accuracy comes at the price of a performance overhead, but we believe that real-world applications will remain usable under analysis.**

I. Introduction

JavaScript has become ubiquitous on server and client tiers of contemporary web applications. Accordingly, the research community has shown an increasing interest in helping web developers to understand and maintain JavaScript programs. The study of a program's execution, known as dynamic program analysis, has become a common technique in this respect [3]. In practice, however, proposed dynamic analysis tools are often built from scratch which constitutes a serious duplication of efforts – *e.g.,* [1], [10], [17]. In this paper, we introduce Linvail, a novel instrumentation platform aiming at providing a common ground for future dynamic analysis tools targeting JavaScript programs. In essence, Linvail enables tagging runtime values with meta-data, a process known as *shadow execution* [9] which is central in many dynamic analyses such as taint analysis [14] and concolic testing [12], [5]. Linvail is a pure JavaScript solution based on source code instrumentation; it does not rely on a modified JavaScript engine. Providing a custom JavaScript runtime would enable powerful analyses that break the constraints imposed by the ECMAScript specification. Given the large diversity among JavaScript engines and the fast pace at which they evolve, we focus on source code instrumentation instead. The design of Linvail has been driven by three important criteria:

Type-independent value tagging: runtime values should be taggeable regardless of their types. This is important because dynamic analyses commonly deal with all types of values.

Transparent analysis layer: the program under analysis should not be impacted by the underlying shadow execution. If this is not the case, the conclusion drawn during the analysis may be invalid.

Life-long value tracking: runtime values should keep the same tag during their entire lifetime. Many analyses implicitly rely on this assumption in order to produce precise results.

Fulfilling the second and the third criterion is straightforward for objects. For them, it suffices to maintain a mapping from pointers to tags. Unfortunately, such a tagging strategy cannot be applied to primitive values because they cannot always be properly differentiated. Binary instrumentation platforms such as PIN [7] and Valgrind [9] have successfully overcome this difficulty by mirroring the entire state of the program being analyzed. However, it remains unclear whether their approach can be implemented for JavaScript at the source code level. In this paper, we explore a tagging strategy where primitive values are wrapped into special objects. This effectively converts them into traceable references.

While wrapping is a simple yet powerful solution for tracking primitive values, it suffers from the *two body* problem [4] –*i.e.,* a wrapper and the primitive it wraps, called *wrappee*, remain two distinct entities. On the one hand, the transparency of the analysis may be compromised if wrappers escape the analysis layer to pollute the base program layer. We address this issue by installing an access control system between these two layers which is inspired by Miller's membranes [8], [16]. On the other hand, primitive values may not be tracked for their entire life-time if the link between wrappers and wrappees is lost. We preserve this link by means of an oracle that knows about the semantics of JavaScript built-in functions.

We validate Linvail by building a number of representative dynamic analyses including (1) a tracker of the origin of null and undefined values, (2) an analysis of the dependencies between "not-a-number" values, (3) a precise tracer, (4) a simple taint checker and (5) a path constraint collector. With respect to our three criteria, we show that these analyses outperform similar analyses built on top of a state-of-the-art JavaScript

instrumentation platform, namely JALANGI. Finally, we demonstrate the applicability of our approach by running our analyses on the SunSpider benchmark suite. Although we observed a significant performance overhead on the SunSpider benchmarks, we believe that real-world JavaScript programs are less computationally expensive and will remain usable.

II. MOTIVATING EXAMPLE

We illustrate the need for a new dynamic analysis platform through the concrete example of implementing a dynamic taint analysis. Dynamic taint analysis is a form of information flow analysis which aims at detecting flows of data that violate program integrity by marking or *tainting* program values [14]. The goal of our taint analysis is to detect whether password-related information can leak to the Document Object Model (DOM) and therefore become visible screen. As stated in the introduction, we wanted our approach to be independent of a specific JavaScript engine out of applicability concerns. Hence we only consider program instrumentation techniques in this section.

A. Tagging Runtime Values

In our taint analysis, the initial values to be tainted/-tagged are the strings taken from password fields. String is not the only type of value that our analysis should support though; in JavaScript, it is easy to convert a string to an array of numbers or use it as a property name inside an object. After a number of these operations, the entire range of runtime values in a program is susceptible to contain password-related information. This brings to the surface our first criterion: **A general-purpose platform for shadow execution should be able to tag values regardless of their type**.

Our taint analysis, and shadow execution in general, requires modifying language semantics to ensure that analysis tags are properly updated while a program under analysis is executed. To this end, program instrumentation is often combined with reflection if supported by the language. Reflective APIs are appreciated because they take care of low-level concerns such as separating the base layer from the meta layer. Unfortunately, current JavaScript reflection capabilities are restricted to object-related operations [15]. Hence, if our taint analysis would solely rely on the reflective API of JavaScript, it would not be able to reason about primitive values and therefore fail our first criterion.

An alternative lower-level approach to modifying the semantics of a language is to systematically replace syntactic expressions by calls to trap functions as depicted in Table I. For instance, after instrumentation, the trap `binary` can be called in place of every binary expression present in the original program. This would essentially

Original	Instrumented
`1`	`traps.literal(1,AST)`
`x ? y : z`	`traps.test(x) ? y : z`
`o.a`	`traps.get(o,"a",AST)`
`o.a = x`	`traps.set(o,"a",AST)`
`for (k in o) …`	`… traps.enumerate(o) …`
`x + y`	`traps.binary("+",x,y,AST)`
`!x`	`traps.unary("!",x,AST)`
`f(x,y)`	`traps.apply(f,null,[x,y],AST)`

Table I: Sample of program instrumentation inserting syntactic traps

enable rewriting the semantics of JavaScript's binary operations. Such a program transformation can be implemented using existing instrumentation platforms such as JALANGI2[1] and ARAN[2]. While syntactic traps enable tagging objects by comparing pointers, primitive values remain an issue. Suppose that our taint analysis is applied to a program requesting a birthdate and a password, and that the user carelessly uses its birthdate as a password. In that case, the string originating from the password field cannot be differentiated from the string originating from the date field. Yet only the password string should trigger an alarm if it happens to escape to the DOM.

Popular X86 binary instrumentation platforms such as PIN [7] and VALGRIND [9] overcome this difficulty by mirroring the entire state of the program under analysis. However, JavaScript is significantly more complex than machine code. It therefore remains unclear whether shadow states can be fully implemented at the source code level of JavaScript. This point is further discussed in Section VI. In this paper, we explore an alternative tagging strategy where primitive values are wrapped into special objects. This effectively converts them into traceable references. The following listing illustrates a manual adaptation of such a tagging strategy to the specific needs of the taint analysis:

```
1 var tainted = new WeakMap(), wrapped = new WeakSet();
2 function taint (x, t) {
3   if (t && isPrimitive(x))
4     wrapped.add(x = {inner:x});
5   if (t)
6     tainted.set(x, t);
7   return x;
8 }
9 function clean (x) { return wrapped.has(x) ? x.inner : x }
```

Listing 1: Taint using explicit wrappers

[1]https://github.com/ksen007/jalangi2
[2]https://github.com/lachrist/aran

Listing 1 tracks runtime values through two weak collections[3]: `tainted` which maps pointers to taints and `wrapped` which differentiates wrappers from the runtime values of the program under analysis. The function `clean` unwraps its argument if applicable.

B. Naive Implementation

Based on the tagging strategy of Listing 1, we provide in Listing 2 a naive implementation of our taint analysis by defining the syntactic traps of Table I.

```
1 var instrumented = new WeakSet();
2 var traps = {};
3 traps.test = clean;
4 traps.get = function (obj, key, ast) {
5   var src = obj instanceof HTMLInputElement
6          && obj.type === "password"
7          && clean(key) === "value";
8   return src ? taint(obj.value,ast) : clean(obj)[clean(key)];
9 };
10 traps.set = function (obj, key, val, ast) {
11   var sink = obj instanceof HTMLElement
12           && clean(key) === "textContent"
13   if (sink && tainted.has(val))
14     throw new Error(tainted.get(val)+" leaks at "+ast);
15   if (!wrappers.has(obj) && !isPrimitive(obj))
16     taint(obj, tainted.get(key));
17   return clean(obj)[clean(key)] = val;
18 };
19 traps.enumerate = function (obj, ast) {
20   var keys = [];
21   for (var key in clean(obj))
22     keys.push(taint(key, tainted.get(obj)));
23   return keys;
24 };
25 traps.unary = function (op, arg, ast) {
26   var res = eval(op+" clean(arg)");
27   return taint(res, tainted.get(arg));
28 };
29 traps.binary = function (op, left, right, ast) {
30   var res = eval("clean(left) "+op+" clean(right)");
31   return taint(res, tainted.get(left) || tainted.get(right));
32 };
33 traps.literal = function (val, ast) {
34   if (typeof val === "function")
35     instrumented.add(val)
36   return x;
37 };
38 traps.apply = function (fct, ths, args, ast) {
39   if (instrumented.has(fct))
40     return fct.apply(ths, args);
41   throw new Error("Call to non-instrumented function...");
42 };
```

Listing 2: Naive taint analysis

The trap `test` makes sure no wrappers are used as predicates inside conditional structures. The trap `get` checks if a password is being fetched from the DOM and the trap `set` checks that no password-related information is being leaked to the DOM. The trap `enumerate` returns an array of tainted keys if the object is tainted. Note that objects become tainted in trap `set` when tainted keys are added to them. The traps `unary` and `binary` forward operations to wrappees; if one of the arguments has been tainted, the taint is propagated to the result. To keep the code concise, unary and binary operations are forwarded using the infamous `eval` function. Alternatively, a case analysis could have been performed on the string `op`. The trap `literal` marks literal functions as being *instrumented*, *i.e.,* they may receive and return wrappers. The trap `apply` forwards the call if the function is instrumented or throws an error otherwise.

Non-instrumented functions are either built-in functions, or functions declared in code areas that the user decided to leave out of the analysis. Section III-A motivates the fine-grained and selective nature of this instrumentation. Since our naive implementation does not support calls to built-in functions at all, it is not usable in practice. To address this limitation, several challenges need to be overcome. These will be discussed in the next section.

C. Calls to Non-Instrumented Functions

When calling a non-instrumented function, a first problem arises when the function accesses a wrapper instead of its wrappee. The non-instrumented function may then behave very differently, causing the instrumented and original code to deviate in behavior. Consider Listing 3; a password is fetched from the DOM at line 1 and it is leaked to the DOM at line 3 under the condition of line 2. In standard JavaScript, this condition always succeeds and our taint analysis should flag a leak in every possible execution. However, if `JSON.stringify` accesses the wrapper of `pass` instead of its wrappee, it will return the string `{"a":{"inner":"secret"}}` instead of `{"a":"secret"}` and the password will not leak into the DOM. Our analysis would then wrongly conclude that the execution respected the taint policy.

This issue illustrates the importance of our second criterion: **The analyzed program should not be impacted by the underlying shadow execution.** To solve this issue, wrappers should be interchanged with their wrappees right before they escape to non-instrumented functions. However, this condition is technically challenging to detect in JavaScript as it can manifest itself in different ways (*c.f.,* Section III-A).

```
1 var pass = document.getElementById("pass").value;
2 if (/{"a":".*"}/.test(JSON.stringify({a:pass})))
3   document.getElementById("leak").textContent = pass;
```

Listing 3: Transparent Analysis Layer

The second problem arises when calling a non-instrumented function breaks the link between a wrappee and its wrappers. This may happen when wrappers are interchanged with their wrappees to preserve the transparency of the analysis. Listing 4 illustrates this with the `Array.prototype.push` and `Array.prototye.pop` built-in functions. A password is fetched from the DOM at

[3]Weak collections have been introduced in the latest JavaScript specification; their key feature is that they allow their elements to be garbage collected. This is sometimes appropriate in the prevention of memory leaks.

line 2, pushed to an empty array, retrieved from that array and leaked into the DOM at line 5. The analysis should indicate a leak at line 5 of password information originating from line 2. For this to happen the values referred to by `pass1` and `pass2` should share the same tag; not because they are *equal* but because they share the same *origin*. This brings us to our third criterion: **Runtime values should keep the same tag during their entire life-time**. For wrappees being standard primitive values, it is challenging to track what they become after non-instrumented functions return.

```
1 var array = [];
2 var pass1 = document.getElementById("pass").value;
3 array.push(pass1);
4 var pass2 = array.pop();
5 document.getElementById("leak").textContent = pass2;
```

Listing 4: Life-Long Value Tracking

III. Overview of the Approach

In the previous section we showed that the difficult part of implementing shadow execution through program instrumentation is the handling of calls to non-instrumented functions. In Section III-A and Section III-B, we introduce two of Linvail's key components for calling non-instrumented functions that respect the three distilled criteria. We proceed with the implementation of our approach in Section III-C and revisit the motivating example in Section III-D.

A. *Linvail's Access Control System*

The problems listed in Section II-C call for a means to control the values exchanged during calls to non-instrumented functions. To better understand how to implement such a system, let's take a step back from JavaScript and program instrumentation, and analyze the different cases in which a value is exchanged between two parties. We borrow from the software security literature two fictive parties: *Alice* and *Bob*.

1) Alice passes a *primitive value* to Bob. Primitive values being immutable and atomic by definition, Bob can only access the information carried by the primitive values and no further value exchanges may happen.
2) Alice passes a *function* to Bob. If Bob defined the function, no further value exchange may happen. But if it was Alice who defined the function, Bob can further pass values to Alice as arguments and Bob can further receive values from Alice as results.
3) Alice passes a *collection* to Bob. To reason about this case, we introduce the concept of *ownership*. If Bob owns the collection, no further value exchange may happen. But if it is Alice who owns the collection, Bob can further pass values to Alice by writing to the collection and Bob can further receive values from Alice by reading from the collection. Note that the frontier of collection ownership is arbitrary. For instance, nothing prevents Alice from owning collections that Bob created. It could even be possible to partition the ownership of a collection or change its ownership over time.

Considering "Alice" as "instrumented code areas", "Bob" as "non-instrumented code areas" and "collections" as "objects", the enumeration above provides a good idea of the cases that our access control system must cover. To make sure that Alice and Bob cannot exchange information through their scope, we fix the granularity of the selective instrumentation at the module level. Our access control system is based on *proxies* [15], [16], a new API for reflection introduced in ECMAScript6. It enables intercepting object-related operations on distinct objects called *proxies*. In reflection terminology, the process of transforming a regular object to a proxy is an instance of *virtualization*. Listing 5 provides an example of virtualization using the constructor `Proxy` which accepts a value to virtualize and a set of trap functions.

```
1 var handlers = {
2   get: function (target, key, receiver) {
3     console.log("get "+key);
4     return Reflect.get(target, key, receiver);
5   },
6   set: function (target, key, value, receiver) {
7     console.log("set "+key+" "+value);
8     return Reflect.set(target, key, receiver);
9   },
10  apply: function (target, self, arguments) {
11    console.log("apply "+arguments);
12    return Reflect.apply(target, self, arguments);
13  }
14 };
15 var proxy = new Proxy(object, handlers);
```

Listing 5: A log-and-forward proxy

In our approach, both functions defined inside instrumented code areas and objects owned by an instrumented code area are virtualized. The frontier of object ownership is discussed in Section III-B. Our access control system is parametrized by two functions: `enter` and `leave` which correspond to the instrumented code's perception for value exchanges. When a non-instrumented function is invoked, `leave` and `enter` are called on the arguments and the result, respectively. The rest of the exchanges in the enumeration above are being controlled by the traps present in our proxies. These traps have two modes: the *transparent* mode which is active when instrumented code is being executed and the *control* mode which is active when non-instrumented code is being executed. In transparent mode, the traps just forward the operations to the virtualized object. In control mode, the operations are also forwarded but the values involved pass properly through the function `enter` and `leave`.

The design of our access control system is inspired by the *membrane* pattern [16], [8]. This pattern ensures that only revocable references can be exchanged between two entities. Revoking the membrane would then instantaneously prevent any further communication between the two entities. To the best of our knowledge, we are the first to use a variant of the membrane pattern in a dynamic program analysis setting. Also, there exist two key differences between the membrane pattern and our access control system. First, in the membrane pattern, values are virtualized when they are exchanged whereas in our approach proxies exist in both entities but have two actionable modes. Second, in our approach, the ownership of an object can change over time and is not fixed by its creation location which is not the case in the membrane pattern.

B. LINVAIL's Oracle

Our access control system alone is sufficient to preserve the transparency of analyses; it suffices to provide a function `leave` that unwraps the value escaping from instrumented code. For instance, function `clean` of Listing 1 could directly be set as the `leave` function. The simplest implementation of the function `enter` would be to ask the analysis layer directly whether it wants to wrap the value entering instrumented code. However, this would mean that we systematically loose track of all values involved in non-instrumented code. Such behavior performs poorly with respect to our third criterion. The purpose of our oracle is to improve the tracking of runtime values.

First, our oracle is responsible for rewrapping entering values. Consider the built-in `Array.prototype.forEach` which applies a callback to all the elements of an array as depicted in Listing 6. When the call `xs.forEach(f, t)` is evaluated, our access control system detects the following initial sequence of exchanges: `xs` leaves, `f` leaves, `xs.length` leaves, `xs[0]` leaves, `xs[0]` enters, `i` enters, `xs` enters. The oracle's responsibility is to detect that the `xs[0]` and `xs` values entering are not novel from the point of view of the instrumented code. Hence, the analysis layer should only be notified about the entrance of `i`. This requires our oracle to retrieve the wrappers of `xs[0]` and `xs` if any. To perform this task, our oracle uses information that has to be entered manually about important JavaScript built-ins.

```
1 Array.prototype.forEach = function (f) {
2   var l = this.length;
3   for (var i = 0; i<l; i++)
4     f(this[i], i, this);
5 }
```

Listing 6: Simplified polyfill for `Array.prototype.forEach`

The second responsibility of our oracle is to define the frontier of object ownership. For the sake of simplicity, our oracle is stateless; rewrapping is only based on the values exchanged locally. Consider again Listing 4, if `array` is not owned by the instrumented code it may not contain wrappers. Hence, our oracle cannot deduce that `pass1` and `pass2` refer to the exact same value because no information is kept between line 3 and 4. This example illustrates that our approach performs better as the instrumented code owns more objects. However, before changing an object's ownership, we must be sure that non-instrumented code has not kept an alias to this object. Otherwise, wrappers could be written to the object while non-instrumented code could by-pass our access control system by accessing this alias. As for rewrapping, our oracle also uses information that we manually input. For instance, our oracle knows that the built-in `Object.create(proto)` returns an alias-free object. An important class of objects that can be guaranteed to be alias-free are the objects pre-existing inside the global object and the global object itself. Consequently, values written to the global object will be considered as leaving instrumented code.

Code	External function
`new F(x,y)`	`Reflect.construct(F, [x,y])`
`o.a`	`Reflect.get(o, "a")`
`o.a = x`	`Reflect.set(o, "a", x)`
`delete o.a`	`Reflect.delete(o, "a")`
`for (k in o) ...`	`Reflect.enumerate(o)`
`!x`	`Reflect.unary("!", x)`
`x + y`	`Reflect.binary("+", x, y)`

Table II: Normalized language constructs as calls to reflective function; `Reflect.unary` and `Reflect.binary` are not standard.

C. LINVAIL's Implementation

We have implemented LINVAIL and made it available under the MIT license at `https://github.com/lachrist/linvail`. Figure 1 depicts the interaction between the different components of an analysis built on top of LINVAIL. On the bottom of the diagram is the instrumentation layer. It provides a simplified interface by normalizing language constructs into calls to reflective functions as depicted in Table II. Our access control system implements this interface; it expects the functions `enter` and `leave` and provides the function `chown` which makes an object belong to the instrumented code by virtualizing it. Next comes our oracle which provides the function `enter` and `leave` and manages the ownership frontier with calls to the function `chown`. Finally comes an analysis layer, which has to be implemented by the user of our approach. It is the focus of this section.

Before proceeding to the API of our approach, we briefly assess how our approach performs with respect the three criteria introduced in Section II.

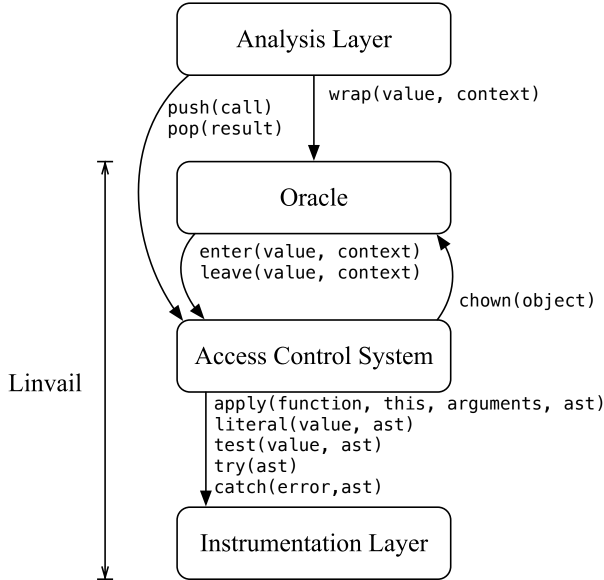

Figure 1: Component architecture of a LINVAIL analysis

Type independent value tagging: Through the `wrap` function, the analysis layer has the opportunity to tag every single value entering instrumented code; this first criterion is therefore entirely satisfied.

Transparent shadow execution: Thanks to our access control system, no wrapper may escape to non-instrumented code. The second criterion is therefore also satisfied. Note that the analysis layer can still impact the execution of the program under analysis. For instance, `stack.push` could throw an error and preclude the program under analysis from continuing its normal execution. Also, the logging required for an analysis can affect the performance of the executed program. In short, our approach enables but does not ensure transparent analyses.

Life-long value tracking: The oracle was introduced for our approach to satisfy this criterion. However, it performs based on knowledge about the implementation of the non-instrumented functions that are being executed. Without this knowledge, the oracle resorts to a conservative strategy that might result in incomplete traces for the primitive values involved.

Listing 7 demonstrates the usage of LINVAIL; it depicts an empty analysis that wraps every single value entering instrumented code. The top-level function of our approach expects two arguments and returns a function performing the analysis. The first argument is a stack-like object that should provide two methods: `push` and `pop`. They will be invoked before, and after a call is performed within instrumented code, respectively. The argument passed to `push` is an object containing the fields: `function`, `this`, `arguments` and `ast`. The `function`, `this` and `arguments` fields contain the values involved during the call. The `AST` field is the ESTree syntactic node[4] where the call occurred. Note that many language-level constructs are normalized into calls to built-in functions. For instance, the expression `x + y` is processed by the analysis as `Reflect.binary("+", x, y)`. Original operations can still be retrieved through `call.ast` nevertheless.

The second argument is a function that is called when our access control system detects that a value enters the instrumented code, and that this value is considered new by the oracle. The two arguments passed to this function are the entering value and some contextual information. The value returned by this function, if any, is a wrapper object containing a method `unwrap`. This method will be called with contextual information whenever the value leaves instrumented code. Although wrapping is only required for primitive values, we extend it to objects to provide a uniform API. The possible values provided by LINVAIL for the contextual information are: (i) An AST node: for literal values and conditional structures present in instrumented code; (ii) A string: produced to explain the origin of values exchanged in calls to non-instrumented functions *e.g.,* `"result"` or `"arguments[0]"`. (iii) Null: for all other values.

```
1 var stack = {};
2 stack.push = function (call) {};
3 stack.pop = function (result) {};
4 function unwrap (contex) { return this.inner }
5 function wrap (x, contex) { return {inner:x, unwrap:unwrap} }
6 var analysis = Linvail(stack, wrap);
7 analysis(jsCode);
```

Listing 7: An empty, wrap-everything analysis.

D. Revisiting the Motivating Example

In Listing 8, we provide a second implementation of our motivating example. This time, we are using LINVAIL's API instead of the syntactic traps of Table I. While the size of our two different implementations is comparable, calls to non-instrumented functions are now properly supported.

Lines 1–19 implement the call stack interface. Before an instrumented call is performed, `calls.push` is invoked and it is first checked whether the call is a source at line 3 till 6 or a sink at line 8 till 10. The call is then decorated with an empty array that will be populated later by wrappers exchanged with non-instrumented code. Note that this array will remain empty if the function about to be called is instrumented. Line 12 finally pushes the call to the stack. After performing an instrumented call, `calls.pop` is invoked and the taint of the call is propagated to all wrappers involved in the call.

[4] https://github.com/estree/estree

The remainder of the code snippet shows the `wrap` and `unwrap` functions. Whenever a new value enters instrumented code, `wrap` is invoked. If the current call is tainted or if the entering value is an object, a wrapper is returned. We track every object because they may become tainted later. When a wrapped value leaves instrumented code, `unwrap` is invoked. If the wrapper is tainted and the current call is a sink, an error is thrown at line 27 notifying the developer that password-related information is leaking to the DOM. Otherwise, the taint of the wrapper is propagated to the call and the wrapper is added to the array of wrapper involved in the call.

```
1  var calls = [];
2  calls.push = function (call) {
3    call.taint  = call.function === Reflect.get
4               && call.arguments[0] instanceof HTMLInputElement
5               && call.arguments[0].type === "password"
6               && call.arguments[1] === "value"
7               && call.ast;
8    call.sink = call.function === Reflect.set
9             && call.arguments[0] instanceof HTMLElement
10            && call.arguments[1] === "textContent";
11   call.wrappers = [];
12   Array.prototype.push.call(this, call);
13 };
14 calls.pop = function (res) {
15   var c = Array.prototype.pop.call(this);
16   for (var i=0; i<c.wrappers.length; i++)
17     c.wrappers[i].taint = c.wrappers[i].taint || c.taint;
18 }
19 calls.last = function () { return this[this.length-1] };
20 function wrap (val, ctx) {
21   var taint = calls.last().taint
22   if (taint || !isPrimitive(val))
23     return {inner:val, taint:taint, unwrap:unwrap};
24 }
25 function unwrap (ctx) {
26   if (this.taint && calls.last().sink)
27     throw tnt+" leaks at "+calls.last().ast;
28   calls.last().taint = calls.last().taint || this.taint;
29   calls.last().wrappers.push(this);
30   return this.inner;
31 }
32 var analysis = Linvail(calls, wrap);
```

Listing 8: Taint analysis built on top of Linvail

IV. Technical Discussion

In this section, we discuss some technical points that were left open until now. The programs supported by Linvail have to comply with the strict mode of ECMAScript5[5]. ECMAScript6 has since been released, but we do not foresee any challenges in supporting the strict mode of ECMAScript6 as well. Supporting standard mode of either ECMAScript5 or ECMAScript6, on the other hand, will require a larger implementation effort. This is because strict mode is an alternative to the standard semantics that protects programs from some common pitfalls. It also reduces the number of interactions of a value with the environment that Linvail has to support: (i) The language construct `with` allows using a plain object as an environment frame. (ii) The ability to pass a string to the `eval` function that defines new variables in the current environment frame. (iii) The aliases between the fields of the `arguments` object and the formal parameters of a function.

[5]Programs can activate this mode using a `"use strict";` statement.

A more pressing limitation of our implementation concerns the way in which the analyzed program is modularized. The Linvail prototype only supports monolithic scripts at the moment. Support for the server-side as well the client-side module systems that are becoming prevalent is immediate future work.

Linvail itself imposes some requirements on the JavaScript engine it is executed on. Our implementation requires the *proxy* API introduced by ECMAScript6. Currently, only the Firefox and Edge browsers fully support this API. However, it should be supported by more JavaScript engines in the near future – *e.g.,* V8[6].

Finally, our implementation of the instrumentation process is of particular note. As depicted in Listing 7, analyses created using Linvail result in an `analysis` function that instruments and executes a JavaScript program. This design decision enables supporting programs that generate code dynamically, a distinct advantage of our design decision.

V. Evaluation

Our evaluation of Linvail is three-fold. First we establish that Linvail is sufficiently expressive to build several representative dynamic analyses. We then show and explain why the quality of our analyses outperforms analyses built on top of Jalangi, a state-of-the-art platform for instrumenting JavaScript. Finally, we demonstrate that analyses built using Linvail are capable of analyzing realistic programs from the SunSpider benchmark suite. All the data related to our experiments is available at `http://soft.vub.ac.be/~lachrist/linvail/validation`.

A. Ease of Implementing Analyses

To demonstrate the expressiveness of our approach, we implemented six analyses on top of Linvail. All but the `identity` analysis are practically relevant and representative for the kind of analyses targeted by shadow execution platforms [13], [9], [7]. We indicate the lines of code required for each implementation as a very rough estimation of their complexity:

`track-void` ($\approx$ 30 loc) tracks the origin of `undefined` values and `null`. As in most other languages, many errors in JavaScript programs are due to null pointer dereferences. When a value being `null` or `undefined` causes the language runtime to throw an exception, this analysis logs the origin of the faulty value to help developers fix the bug.

[6]`https://code.google.com/p/v8/issues/detail?id=1543`

`track-nan` (≈ 30 loc) tracks the dependencies between `NaN` values. In JavaScript, `NaN` is a special value that indicates a failure during mathematical and parsing-related operations. For instance, both `Math.sqrt(-1)` and `parseInt("foo")` return `NaN`. The problem with `NaN` values is that they quickly propagate through successive calls, which renders pinpointing their origin difficult. By tracking the dependencies between `NaN` values, our analysis facilitates diagnosing their occurrences.

`tracer` (≈ 70 loc) is a high-precision tracer for JavaScript. In this analysis, all internal primitive values as well as objects are tagged with a unique identifier. Every call is logged along with its arguments and results when available. The resulting trace is sufficiently precise to replay a recorded execution faithfully.

`taint` (≈ 50 loc) corresponds to the taint analysis presented in Section III. Values read from password fields in the DOM are tainted. Taint is propagated to external functions that are called. If a tainted value is written to the DOM, an error is thrown.

`constraint` (≈ 50 loc) symbolically records the path constraints encountered during an execution, as a concolic tester [?] would. Every value coming from an input field in the DOM is tagged with a fresh symbol. When a symbolic value is used inside a binary operation, its result will be tagged with a fresh symbol linked to the binary operation. When such a symbolic value is used inside a branching instruction, a path constraint encoding the branch that was taken is recorded.

`identity` (≈ 10 loc) is an identity analysis that has no effect; our proxy membrane is activated but no primitive wrapping is performed. This analysis has no purpose but to provide more insight into LINVAIL's applicability for Section V-C.

B. Quality of Implemented Analyses

To assess the quality of analyses built on top of LINVAIL, we compare them with similar analyses built on top of JALANGI. JALANGI [13] is a state-of-the-art JavaScript instrumentation platform which inspired the syntactic traps depicted in Table I. It has been employed successfully in building several dynamic analysis tools for JavaScript (*e.g.,* [11], [6]). As such, it is the closest work related to LINVAIL. We have inspected both its original version (*i.e.,* JALANGI1[7]), which is now considered legacy software, and the version that is currently maintained (*i.e.,* JALANGI2[8]) thoroughly.

The inspection of JALANGI2's implementation reveals that syntactic traps comprise the only mechanism available to implement dynamic analyses. As we observed in Section II, syntactic traps alone are not sufficient to implement shadow execution correctly for primitive values. This observation is consistent with the fact that example analyses included with `Jalangi2`, deal with primitive values by simply logging them without informing the user about their origin. Figure 2 illustrates the limitations of straightforward logging over actual shadow execution. The execution of a small program is depicted under two analyses that both aim at diagnosing `NaN` appearances. The first analysis is `CheckNaN` which can be found in `Jalangi2`'s repository as a demonstrating example; the second analysis is our own `NaN` value tracker, `track-nan`. The output of `CheckNaN` is not very useful for understanding why `y` is assigned to `NaN`, whereas `track-nan` provides a pointer to `Math.sqrt(-1)` which actually causes the propagation of `NaN`.

```
1 var x = Math.sqrt(-1);
2 var y = 2 * x;
```

```
—————————— CheckNaN (Jalangi2) ——————————
Observed NaN at testNaN1.js:1:9:1:22 1 times.
Observed NaN at testNaN1.js:2:9:2:14 1 times.
—————————— track-nan (Linvail) ——————————
NaN1: sqrt@1:8 [-1]
NaN2: binary@2:8 [*, 2, NaN1]
```

Figure 2: `NaN` tracking using JALANGI2 and LINVAIL.

In JALANGI1's repository, however, we found an additional means for implementing dynamic analyses akin to concolic testing: *concolic values*. Concolic values, originally coined *annotated values* in [13], work similarly to the wrappers of Listing 2. They carry two fields: `concrete` which contains a runtime value and `symbolic` which contains the tag associated to this runtime value. As in Listing 2, appropriate unwrapping operations are hard-coded for language-level operations such as property assignment. The key difference with our approach lies in how JALANGI1 supports external calls.

During an *online* analysis (as opposed to a *post-mortem* one), JALANGI1 relies on the use of JavaScript's `Object.prototype.valueOf()` method. It is called by the language runtime whenever a built-in function receives an object where a primitive value was expected, with the goal of coercing the object into a primitive. By overriding the `valueOf` implementation such that it returns the value of the `concrete` field, JALANGI1 supports calls to built-in functions that exclusively expect primitive values. However, this approach cannot ensure that transparency is preserved during calls to built-in functions expecting objects such as `JSON.stringify` and `Array.prototype.filter`. These claims are demonstrated in Figure 3 which compares our `track-void` analysis to the `UndefinedNullTrackingEngine` example analysis in JALANGI1's repository.

[7]https://github.com/SRA-SiliconValley/jalangi.
[8]https://github.com/Samsung/jalangi2

```
1 console.log(Math.sqrt(null));
2 console.log(JSON.stringify(null));
3 [1,2,3].filter(function (x) { return x });
```

```
UndefinedNullTrackingEngine (Jalangi1-Online)
0
{"concrete":null,"symbolic":"null initialized at ..."}
TypeError: function x { ...
track-void (Linvail)
Null1: literal@1:22
Null1 accessed at 1:12
0
Null2: literal@2:27
Null2 accessed at 2:12
null
```

Figure 3: JALANGI1's online concolic values

```
1 var x = null;
2 var arr = [];
3 arr.push(x);
4 console.log("Before try");
5 try { arr.pop().a }
6 catch (e) { process.stderr.write("Caught null error\n") }
7 if (Math.sqrt({valueOf: function () { return 1 }}) !== 1)
8   throw "this should never happen";
```

```
UndefinedNullTrackingEngine (Jalangi1-Postmortem)
** RECORD **
Before try
Caught null error
** REPLAY **
Before try
null initialized at ...:5:7:5:16
This should never happen
track-void (Linvail)
Null1: literal@1:8
Before try
Null1 accessed at 3:6
Caught null error
```

Figure 4: JALANGI1's postmortem concolic values.

JALANGI1 supports post-mortem analyses through a *record-and-replay* system. During the replay phase, calls to external functions are simply replaced by their results gathered during the record phase. On external functions featuring side effects such as `Array.prototype.pop`, such simulation will make the replay execution deviate from the recorded execution. To hide these deviations from the user, JALANGI1 performs systematic synchronization operations during the replay phase. Not executing external calls cleverly avoids the problem of interfacing them with wrappers. However, it brings about the following two shortcomings. On the one hand, concolic values cannot live throughout the execution of external functions. This defeats the entire purpose of our oracle and results in values being tracked for a much shorter time. Consider Figure 4 which compares the same analyses as Figure 3. In JALANGI1's analysis, the origin of the faulty `null` value is reported to be at line 5 whereas our analysis correctly points to line 1. On the other hand, since the execution is not entirely replayed, users have to reason about partial traces. Consider again Figure 4. It appears that `console.log` is replayed, but not `process.stderr.write`. Based on our experience in using JALANGI, this somewhat arbitrary decision makes it harder to reason about its outputs. We believe these two limitations are the reasons for abandoning the record-and-replay system in JALANGI2. Additionally, the logged message `This should never happen` shows an unreported case in which the replay diverges from the previously recorded execution.

To summarize, the version of JALANGI that is currently maintained only provides syntactic traps which does not suffice to implement shadow execution. The concolic values provided by the legacy version satisfy our first two criteria, but not the last as they cannot live through external calls during the replay phase — significantly reducing the precision of resulting analyses.

C. *Completeness of* LINVAIL

To assess the completeness of our approach in terms of support for JavaScript, we apply our six analyses to the SunSpider benchmark suite[9]. This experiment was conducted using FireFox 38 on a MacBookPro with a CPU 2,5 GHz Intel Core i7 and with 16GB of RAM. The primary goal of our experiment is to demonstrate that LINVAIL can handle complex JavaScript programs. We also want to evaluate the performance impact of our approach, even though our implementation prioritizes quality and expressiveness over performance. Table III summarizes the outcome of applying our six analyses to the 26 JavaScript programs of the SunSpider benchmark suite. For each of these, we provide the lines of code of the original program, the lines of code of the instrumented program, and the slowdown factor induced by the analysis.

No assertion violations were observed during the experiment, indicating that our analyses remained fully transparent and were able to handle the complexity of the programs. However, our slowdown factors are a order magnitude higher than those reported by JALANGI [13]. This additional overhead can be seen as the cost of the qualitative difference demonstrated in Section V-B. We believe developers are willing to make this trade-off, at least while developing an application. Our prototype implementation is, however, less suitable for analyses that are deployed alongside the application. Moreover, the SunSpider benchmark suite performs demanding CPU operations primarily involving primitive values. For instance, the benchmarks prefixed by `string`, `math` and `bitop` respectively manipulate strings, numbers and

[9] https://wiki.mozilla.org/Sunspider_Info

Benchmark	LoC	LoC*	Identity	Track-NaN	Track-Void	Taint	Constraint
3d-cube	357	815	336	396	386	376	365
3d-morph	65	36	3719	3856	4013	4286	4236
3d-raytrace	449	704	556	741	720	730	710
access-binary-trees	56	70	1739	3086	2180	1958	1786
access-fannkuch	72	72	11681	16068	1474	13547	9436
access-nbody	176	145	9202	14502	9249	10985	7045
access-nsieve	48	51	4278	6924	6910	5190	3705
bitops-3bit-bits-in-byte	42	29	10677	29102	18902	17654	13346
bitops-bits-in-byte	32	31	11122	19895	17802	12938	12855
bitops-bitwise-and	37	11	4547	16267	10932	7031	6303
bitops-nsieve-bits	44	45	15413	14581	14201	32	13534
controlflow-recursive	34	72	5605	2826	7201	6391	5700
crypto-aes	428	735	2199	1820	2521	2990	2266
crypto-md5	294	830	2289	2128	2819	2911	1785
crypto-sha1	230	248	2617	3471	3185	4500	2532
date-format-tofte	303	379	1851	2036	2015	2243	1959
date-format-xparb	422	829	43	101	183	48	43
math-cordic	108	79	12436	15227	15401	17350	13376
math-partial-sums	47	44	474	577	601	693	462
math-spectral-norm	61	81	10087	10162	14891	14186	10865
regexp-dna	1723	105	5	7	4	5	5
string-base64	139	199	2022	1492	1720	2465	2029
string-fasta	92	189	1674	1931	2184	2623	2628
string-tagcloud	271	347	18	18	12	22	21
string-unpack-code	83	185	75	29	45	74	44
string-validate-input	94	74	87	355	286	217	251
Average	–	–	4444	6446	5378	5055	4511
First Quartile	–	–	897	396	386	10985	365
Median	–	–	3719	3856	4013	4286	2628
Third Quartile	–	–	10087	10162	7201	2623	6303

Table III: Slowdown factor on the SunSpider benchmarks for five analyses built on top of Linvail. LoC resp. LOC* stands for the number of line of code of the original resp. instrumented benchmark.

even bits. Our approach is expected to perform poorly, but correctly on these kind of programs. Common web applications, however, are expected not to perform as many CPU-intensive operations. The typical slowdown factor is expected to be much lower than 5000X.

VI. Related Work

In this section we discuss alternative techniques for implementing generic shadow executions. Valgrind [9] is a popular framework for instrumenting binaries for the purpose of a dynamic analysis. We select this among many others because it describes the process of shadow execution in detail. During analysis, Valgrind does not wrap primitive values but rather maintains a structure called a *shadow state*. At any time, this shadow state mirrors the state of the program under analysis but contains meta-information rather than runtime values. Since the program state remains free from meta-data, shadow states are not subject to the problems listed in Section II-C. Instead, the primary concern is keeping the shadow state in sync with the program state during system calls. Our approach does not suffer from this concern because side effects of external functions are being controlled by the access control system. In the next paragraph, we discuss how shadow states have been transposed to JavaScript.

In [13], Jalangi1 was presented as a record-and-replay dynamic analysis framework for JavaScript. We already discussed in detail the replay phase which incorporates syntactic traps and concolic values. We now focus on its record phase which embeds a structure called *shadow memory* that can be seen as an adaptation of the aforementioned shadow state. Jalangi mirrors the JavaScript environment by duplicating global variables, local variables and formal parameters. Similarly, the store is mirrored by duplicating the fields of every object.

We finally present works on *virtual values* which enable programmers to redefine the semantics of various operations normally hardwired by the language. Most of the time, virtual values are a collection of traps, each of which is a user-defined function that describes how a particular operation should behave on that virtual value. If a virtual value is encountered during an operation, its corresponding trap is triggered. The API provided in Section III-C can be seen as a virtualizing the runtime values of the program under analysis. The new proxy API for JavaScript [15] previously introduced in Section II-A comprises another example of value virtualization. But as proxies cannot virtualize primitive values, we could not use them as such in our work. In [2],

Austin et al. proposed a reflective API for JavaScript capable of virtualizing any value regardless of their type. The approach is prototyped using the meta-circular interpreter NARCISSUS. As a proof of concept, it is not capable of dealing with the complexity of real-world JavaScript applications. In particular, it is not clear how its virtual values interact with built-in JavaScript functions — which is the primary concern of our work.

VII. CONCLUSION

We presented LINVAIL, a novel approach for supporting developers in building dynamic analysis tools for JavaScript. LINVAIL enables shadow execution through two key components: (i) An access control system that prevents analysis-related data from escaping to non-instrumented areas in the code. (ii) An oracle that uses its knowledge about JavaScript built-ins to improve the precision with which runtime values are tracked. We have shown example analyses built on top of LINVAIL to be more precise than similar analyses built on top of JALANGI, a state of the art JavaScript instrumentation framework. However, our experiments demonstrate that LINVAIL's accuracy comes at the cost of a performance overhead. We believe that real-world JavaScript programs are computationally less expensive and will remain usable under analysis.

REFERENCES

[1] Shay Artzi, Julian Dolby, Simon Holm Jensen, Anders Moller, and Frank Tip. A framework for automated testing of JavaScript web applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE11)*, pages 571–580, 2011.

[2] Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual values for language extension. In *Proceedings of the 26th International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA11)*, pages 921–938, 2011.

[3] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, pages 684–702, 2009.

[4] Patrick Eugster. Uniform proxies for Java. In *Proceedings of the 21st International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA06)*, pages 139–152, 2006.

[5] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation (PLDI05)*, pages 213–223, 2005.

[6] Liang Gong, Michael Pradel, and Koushik Sen. Jitprof: pinpointing jit-unfriendly JavaScript code. Technical report, Electrical Engineering and Computer Sciences University of California at Berkeley, 2014.

[7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation (PLDI05)*, pages 190–200, 2005.

[8] Mark Samuel Miller and Jonathan S Shapiro. *Robust composition: towards a unified approach to access control and concurrency control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, 2006.

[9] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 Conference on Programming Language Design and Implementation (PLDI07)*, pages 89–100, 2007.

[10] FS Ocariza, Karthik Pattabiraman, and Ali Mesbah. Autoflox: an automatic fault localizer for client-side JavaScript. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation, (ICST12)*, pages 31–40, 2012.

[11] Michael Pradel, Parker Schuh, and Koushik Sen. Typedevil: dynamic type inconsistency analysis for JavaScript. In *Proceedings of the 37th International Conference on Software Engineering (ICSE15)*, 2015.

[12] Koushik Sen and Gul Agha. CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International ConferenceComputer Aided Verification (CAV06)*, pages 419–423, 2006.

[13] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. Jalangi: a selective record-replay and dynamic analysis framework for JavaScript. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE13)*, pages 488–498, 2013.

[14] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. TAJ: effective taint analysis of web applications. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation (PLDI09)*, pages 87–97, 2009.

[15] Tom Van Cutsem and Mark S Miller. Proxies: design principles for robust object-oriented intercession APIs. In *Proceedings of the 6th Symposium on Dynamic Languages (DLS10)*, pages 59–72, 2010.

[16] Tom Van Cutsem and Mark S Miller. Trustworthy proxies. In *Proceedings of the 27th European Conference on Object-Oriented Programming (ECOOP13)*, pages 154–178, 2013.

[17] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS07)*, 2007.