Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript

Koushik Sen*
EECS Department
UC Berkeley, CA, USA.
ksen@cs.berkeley.edu

Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs
Samsung Research America
1732 North First Street, San Jose, CA, USA
{s.kalasapur,t.brutch,s.gibbs}@sisa.samsung.com

ABSTRACT

JavaScript is widely used for writing client-side web applications and is getting increasingly popular for writing mobile applications. However, unlike C, C++, and Java, there are not that many tools available for analysis and testing of JavaScript applications. In this paper, we present a simple yet powerful framework, called JALANGI, for writing heavy-weight dynamic analyses. Our framework incorporates two key techniques: 1) selective record-replay, a technique which enables to record and to faithfully replay a user-selected part of the program, and 2) shadow values and shadow execution, which enables easy implementation of heavy-weight dynamic analyses. Our implementation makes no special assumption about JavaScript, which makes it applicable to real-world JavaScript programs running on multiple platforms. We have implemented concolic testing, an analysis to track origins of nulls and undefined, a simple form of taint analysis, an analysis to detect likely type inconsistencies, and an object allocation profiler in JALANGI. Our evaluation of JALANGI on the SunSpider benchmark suite and on five web applications shows that JALANGI has an average slowdown of 26X during recording and 30X slowdown during replay and analysis. The slowdowns are comparable with slowdowns reported for similar tools, such as PIN and Valgrind for x86 binaries. We believe that the techniques proposed in this paper are applicable to other dynamic languages.

1. INTRODUCTION

JavaScript is the most popular programming language for client-side web programming. Advances in browser technologies and JavaScript engines in the recent years have fueled the use of JavaScript in Rich Internet Applications, and several mobile platforms including Android, iOS, Tizen, Windows8, Blackberry, Gnome, support applications written in HTML5/JavaScript. A key reason behind the popularity of JavaScript programs is that they are portable. Once written, JavaScript based applications can be executed on any platform that has a web browser with JavaScript support, which is quite common in modern day devices. JavaScript being a dynamic language, also attracts developers through its flexible features that do not require explicit memory management, static typing and compilation. With a renewed interest in JavaScript, many complex applications such as Google docs, gmail, and a variety of games are being developed using HTML5/JavaScript. However, unlike C/C++, Java and C#, JavaScript is significantly shorthanded in the tools landscape. The dynamic and reflective nature of JavaScript makes it hard to analyze it statically [27, 34, 25].

In this paper, we present a dynamic analysis framework, called JALANGI, for Javacript. The framework provides a few useful abstractions and an API that significantly simplifies implementation of dynamic analyses for JavaScript. The framework works through source code instrumentation and allows implementation of various heavy-weight dynamic analyses techniques. JALANGI incorporates two ideas:

1. *Selective record-replay*, a technique which enables to record and to faithfully replay a user-selected part of the program. For example, if a JavaScript application, uses several third-party modules, such as jQuery, Box2DJS, along with an application specific library called myapp.js, our framework enables us to only record and replay the behavior of myapp.js.
2. *Shadow values*, which enables us to associate a shadow value with any value used in the program. A shadow value can contain useful information about the actual value (e.g. taint information or symbolic representation of the actual value). The framework supports *shadow execution* on shadow values, a technique in which an analysis can update the shadow values and analysis state, on each operation performed by the actual execution. For example, a shadow execution can perform symbolic execution or dynamic taint propagation.

There are a few constraints which dictated the design of the above techniques in JALANGI.

1. We wanted to design a framework that is independent of browsers and JavaScript engines. Such a design enables us to design dynamic analyses that are not tied to a particular JavaScript engine. Independence from

*The work of this author was supported in full by Samsung Research America.

ESEC/FSE '13, August 18-26, 2013, Saint Petersburg, Russia

browsers and JavaScript engines also enables us to easily maintain our framework in the face of rapidly evolving browser landscape—we do not need to upgrade or rebuild our framework whenever there is an update of the underlying browser. We achieve browser independence through selective source instrumentation. *An attractive feature of* JALANGI *is that it can operate even if certain source files are not instrumented.*

2. We wanted a framework where dynamic analysis of an actual execution on a browser (e.g. a mobile browser) can be performed on a desktop or a cloud machine. This is important when we want to perform a heavy-weight analysis, such as symbolic execution. A heavy-weight analysis is often impossible to perform on a resource constrained mobile browser. Moreover an analysis that requires access to various system resources, such as file system, cannot be implemented in a browser without significantly modifying the browser. *We address this design constraint through a two-phase analysis framework.* In the first phase, an instrumented JavaScript application is executed and recorded on a user selected platform (e.g. mobile chrome running on Android). In the second phase, the recorded data is utilized to perform a user specified dynamic analysis on a desktop environment.

3. A dynamic analyses framework should allow easy implementation of a dynamic analysis. Previous research [31, 20, 8, 6, 21] and our experience with concolic testing [13, 30] and race detection techniques have shown that support for shadow values and shadow execution could significantly simplify implementation of dynamic analyses techniques. A straight-forward way to implement shadow value would be to replace any value, say `val`, used in a JavaScript execution by an object, called *annotated* value, `{actual: val, shadow: "tainted"}`, where the field `actual` stores the actual value and the field `shadow` can store necessary information about `val`. To accomodate such replacements, we modify every operation (e.g. +, *, field access) performed by the JavaScript execution because every value, whether primitive or not, could now be wrapped by an object. The modified operations first retrieve the actual values from the annotated values representing the operands of the operation and then perform the operation on the actual values to compute the result of the operation. This simple implementation would work if we can modify every operation performed by a JavaScript engine. Unfortunately, JALANGI instruments only user-specified code. Moreover, JALANGI cannot instrument native code. Therefore, if we call `array.pop()`, where `array` is an annotated value and `pop` is a native function, we will get an exception. JALANGI alleviates this problem by using the selective record-replay engine: it only records the execution of the instrumented code and replays the instrumented code. Any code that is not and can not be instrumented, including native code, is not executed during the replay phase. Since JALANGI supports shadow values and shadow execution during the replay phase, it will never execute un-instrumented code on annotated values. Thus, JALANGI's record-replay technique is necessary for correct support of shadow values and shadow execution.

In JALANGI, we have implemented several existing and new dynamic analyses:

- Concolic testing [13, 30]: concolic testing performs symbolic execution along a concrete execution path, generates a logical formula denoting a constraint on the input values, and solves a constraint to generate new test inputs that would execute the program along previously unexplored paths. Our implementation of concolic testing supports constraints over integer, string, and object types and *novel type constraints.*

- Tracking origins of `null` and `undefined` [6]: this analysis records source code locations where null and undefined values come into existence and reports them if they cause an error. Whenever there is an error due to such literals, such as accessing the field of a null value, the shadow value of the literal is reported to the user.

- Dynamic taint analysis [21, 9]: a dynamic taint analysis is a form of information flow analysis which checks if information can flow from a specific set of memory locations, called sources, to another set of memory locations, called sinks. We have implemented a simple form of dynamic taint analysis in JALANGI.

- Detecting likely type inconsistencies: this dynamic analysis checks if an object created at a given program location can assume multiple inconsistent type. Sometimes these kind of type inconsistencies could point us to a potential bug in the program. We have noticed such issues in two SunSpider benchmark programs.

- Simple object allocation profiler: this dynamic analysis computes the number of objects created at a given allocation site and how often the object has been accessed. If an allocation site creates too many contant objects, then it could lead to memory inefficiency. We have found such a problem in one of the web applications in our benchmark suite.

JALANGI is available at `https://github.com/SRA-SiliconValley/jalangi` under Apache 2.0 license. We evaluated JALANGI on the CPU-intensive SunSpider benchmark suite and on several user-interaction rich web applications. Our evaluation results show that JALANGI has an average overhead of 26X during recording and 30X during replay. This is better than PinPlay [22] by a factor of 2X-3X and slower than Valgrind [20]. We also found that existing dynamic analyses could easily be implemented in JALANGI.

2. TECHNICAL DETAILS

To simplify exposition of our techniques (and to avoid explanation of the nuances of JavaScript), we use a simple JavaScript-like imperative language. The syntax of this language is shown below.

$v, v1, v2,$	$v3, \ldots$ are variable identifiers	
$f, f1, f2,$	$\ldots$ are field identifiers	
$p, p1, p2,$	$\ldots$ are function parameter identifiers	
$\texttt{op}$	are operators such as +, -, *, ...	
Pgrm ::=	$(\ell\colon \text{Stmt})^*$	
Stmt ::=	$\texttt{var}\ v$	
	$v = c$	
	$v1 = v2\ \texttt{op}\ v3$	
	$v1 = \texttt{op}\ v2$	
	$v1 = \texttt{call}(v2, v3, v4, \ldots)$	
	$\texttt{if}\ v\ \texttt{goto}\ \ell$	
	$\texttt{return}\ v$	
	$v1 = v2[v3]$	
	$v1[v2] = v3$	
	$\texttt{function}\ v1(p1, \ldots)\{(\ell\colon \text{Stmt})^*\}$	function definition
c ::=	number	
	string	
	$\texttt{undefined}$	
	$\texttt{null}$	
	$\texttt{true}$	
	$\texttt{false}$	
	$\{f1\colon v1, \ldots\}$	object literal
	$[v1, \ldots]$	array literal
	$\texttt{function}\ v1(p1, \ldots)\{(\ell\colon \text{Stmt})^*\}$	function literal

A program in this language is a sequence of labeled statements. The statements in the language are in three-address code. $\texttt{if}\ v\ \texttt{goto}\ \ell$ is the only statement that allows conditional jump to an arbitrary statement. A compiler framework can be used to convert more complex statements of JavaScript into statements of this language by introducing temporary variables and by adding additional statement labels. For example, control-flow statements, such as `while`, `for`, can be converted into a sequence of statements in this language using $\texttt{if}\ v\ \texttt{goto}\ \ell$. We use the statement $v1 = \texttt{call}(v2, v3, v4, \ldots)$ to represent function, method, and constructor calls, where $v2$ denotes the function that is being called, $v3$ denotes the `this` object inside the function, and $v4, \ldots$ denote the arguments passed to the function. We use $v1[v2]$ to denote both access to an element of an array and access to a field of an object.

2.1 Selective Record-Replay

We assume that the user of JALANGI selects a subset of the JavaScript source in a web application for record-replay. JALANGI instruments the user-selected source for record-replay. During the *recording* phase, the application is executed with the instrumented files on a platform of the user's choice (e.g. a mobile browser or a node.js interpreter). During recording, the entire application is executed, i.e. all instrumented and un-instrumented JavaScript files and native codes get executed. During the *replay* phase, JALANGI only replays the execution of the instrumented sections. This asymmetry of execution in the two phases has two key advantages:

1. One could record an execution of a JavaScript application on an actual platform (e.g. a mobile browser) and then replay the execution for the purpose of debugging on a desktop JavaScript engine, such as node.js or a JavaScript engine embedded in an IDE. The replay does not require access to any browser-specific native JavaScript libraries such as libraries for manipulating the DOM.

2. During replay, since we avoid the execution un-instrumented code and native code, we can easily implement various dynamic analysis that depend on shadow values and shadow executions.

A trivial way to perform faithful record-replay of an execution is to record every value loaded from memory during an execution and use those values for corresponding memory loads in the replay phase. This approach has two challenges: 1) How do we record values of objects and functions? 2) How do we replay an execution when an un-instrumented function or a native function, such as the JavaScript event dispatcher, calls an instrumented function? Note that we do not allow the execution of un-instrumented and native functions during the replay phase. Therefore, we need an alternative mechanism to execute instrumented functions that are being invoked by un-instrumented functions during recording. We address the first challenge by associating a unique numerical identifier with every object and function and by recording the value of those unique identifiers. We address the second challenge by explicitly recording and calling instrumented functions that are being invoked from un-instrumented functions or are dispatched by the JavaScript event dispatcher.

We avoid recording of every load of memory based on the following observation: *if we can compute the value of a memory load during the replay phase by solely executing the instrumented code, then we do not need to record the value of the load.*

In order to determine if the value of a memory load needs to be recorded, JALANGI maintains a shadow memory during the recording phase. The shadow memory is updated along with the actual memory during the execution of instrumented code. Execution of un-instrumented and native code does not update the shadow memory. During the load of memory in the recording phase, if JALANGI finds any difference between the value of the actual memory being loaded and the value stored in the corresponding shadow memory, JALANGI records the value of such memory loads. This ensures that correct values are available during the replay phase.

Figure 1 shows the instrumentation that JALANGI performs for record-replay. The instrumentation does not change the behavior of the actual execution. JALANGI introduces a shadow variable v' for every local and global variable v. JALANGI introduces a local variable p' for every formal parameter p of an instrumented function. Similarly, for every field f of every object, JALANGI introduces a shadow field f'. Note that if $v1[v2]$ denotes access of the field denoted by the string stored in $v2$, then $v1[v2 + \text{“}'\text{”}]$ denotes the access of the corresponding shadow field.

During the recording phase, JALANGI keeps the actual memory and shadow memory in sync as much as possible. Note that a field of an object may not be in sync with the corresponding shadow field if the field gets updated in native or un-instrumented code. Whenever a variable or a field of an object is updated, JALANGI adds instrumentation to update the corresponding shadow variable or shadow field of the object. For example, $v1 = v2[v3]$ gets modified to $v1' = v1 = v2[v3]$.

The instrumentation performs the following additional three transformations:

- If a local or global variable v or a field of an object $v1[v2]$ is loaded in a statement, we first call $v' = v = \texttt{sync}(v, v')$ or $v1[v2 + \text{”}'\text{”}] = v1[v2] =$

`var` v	$\Longrightarrow$	`var` v' `var` v
$v = c$	$\Longrightarrow$	$v' = v = \texttt{sync}(c)$
$v1 = v2$ `op` $v3$	$\Longrightarrow$	$v2' = v2 = \texttt{sync}(v2, v2')$ $v3' = v3 = \texttt{sync}(v3, v3')$ $v1' = v1 = v2$ `op` $v3$
$v1 =$ `op` $v2$	$\Longrightarrow$	$v2' = v2 = \texttt{sync}(v2, v2')$ $v1' = v1 =$ `op` $v2$
`if` v `goto` ℓ	$\Longrightarrow$	$v' = v = \texttt{sync}(v, v')$ `if` v `goto` ℓ
`return` v	$\Longrightarrow$	$v' = v = \texttt{sync}(v, v')$ `return` v
$v1 = v2[v3]$	$\Longrightarrow$	$v2' = v2 = \texttt{sync}(v2, v2')$ $v3' = v3 = \texttt{sync}(v3, v3')$ $v2[v3 + \text{“}'\text{”}] = v2[v3] =$ $\quad \texttt{sync}(v2[v3], v2[v3 + \text{“}'\text{”}])$ $v1' = v1 = v2[v3]$
$v1[v2] = v3$	$\Longrightarrow$	$v1' = v1 = \texttt{sync}(v1, v1')$ $v2' = v2 = \texttt{sync}(v2, v2')$ $v3' = v3 = \texttt{sync}(v3, v3')$ $v1[v2 + \text{“}'\text{”}] = v1[v2] = v3$
$v1 = \texttt{call}(v2, v3, v4, \ldots)$	$\Longrightarrow$	$v2' = v2 = \texttt{sync}(v2, v2')$ $v3' = v3 = \texttt{sync}(v3, v3')$ $v4' = v4 = \texttt{sync}(v4, v4')$ $\vdots$ $v1' = v1 = \texttt{sync}($ $\quad \texttt{instrCall}(v2, v3, v4, \ldots))$
$\{f1\colon v1, \ldots\}$	$\Longrightarrow$	$\{f1\colon v1' = v1 =$ $\quad \texttt{sync}(v1, v1'), \ldots\}$
$[v1, \ldots]$	$\Longrightarrow$	$[v1' = v1 = \texttt{sync}(v1, v1'), \ldots]$
`function` $v1(p1, \ldots)\{$ $\quad (\ell\colon \text{Stmt})^*$ $\}$	$\Longrightarrow$	`function` $v1(p1, \ldots)\{$ $\quad \texttt{enter}(v1)$ $\quad$ `var` $p1'$ $\quad \vdots$ $\quad (\ell\colon \text{Stmt})^*$ $\}$

Figure 1: Instrumentation for Record-Replay. $\texttt{sync}(c)$ is equivalent to $\texttt{sync}(c, \texttt{undefined})$.

$\texttt{sync}(v1[v2], v1[v2 + "'"])$, respectively, before the actual load. In the recording phase, the function `sync` records the value stored in the memory if the values stored in the actual and shadow memory are different (i.e. if the arguments of the `sync` are different). In the replay phase, `sync` returns the first argument if the corresponding load in the recording phase was not recorded and returns the recorded value otherwise. This ensures that in the replay phase, JALANGI gets the exact value that is loaded during the recording phase.

- We replace $\texttt{call}(v2, v3, v4, \ldots)$ by $\texttt{sync}(\texttt{instrCall}(v2, v3, v4, \ldots))$. During the replay phase, function `instrCall` invokes $\texttt{call}(v2, v3, v4, \ldots)$ if function $v2$ is instrumented. Otherwise, it explicitly calls any instrumented function that is invoked while executing the un-instrumented or native function $v2$. We use the function `replay` defined in Figure 2 to call instrumented functions whose callers are not instrumented.

- We insert the statement $\texttt{enter}(v1)$ as the first statement of any instrumented function with name, say $v1$. In the recording phase, $\texttt{enter}(v1)$ records the value of the function $v1$. In the replay phase, `instrCall` invokes the recorded function if the function is called from a un-instrumented or native function.

Figure 2 defines the functions `sync`, `instrCall`, and `enter`, which are inserted by JALANGI instrumentation. The library maintains an array `trace` of the recorded values along with their types. `trace[i]` stores the value of the i^{th} memory load. The array is initialized and populated during the recording phase and is used in the replay phase. At the end of recording, `trace` is serialized to the filesystem in JSON format. During replay, the serialized file is used to initialize `trace`.

Function `sync` is defined as described before. If the second argument of `sync` is not provided, then we assume that the second argument is `undefined`. JALANGI uses the flag `recording` to indicate if an execution is meant for recording or replay. For a `recording` execution, if the two arguments of `sync` are different, then JALANGI records the value and the type of the value in the sparse array `trace`. Otherwise, JALANGI skips recording, i.e. keeps the entry `trace[i]` `undefined`. If the value of `v1` in `sync` is restricted to primitive types (i.e. number, string, boolean, undefined, or null), we can simply do `trace[i] = v1`. However, the type of `v1` could be an object or a function. To handle objects and functions, `sync` calls `trace[i] = getRecord(v1)`, where `getRecord(v1)` returns an object whose `type` field is set to the type of `v1` and `val` is set to `v1` if `v1` is of primitive type. If type of `v1` is non-null object or function, then we use the unique numerical id of the object or function as its value to be recorded. The unique numerical id of a non-null object or function is stored in its hidden field `*id*`. If the object or function has no unique id, `getRecord` creates and assigns a unique numerical id to the object or function.

In a replay execution, if `trace[i]` is `undefined` inside a call of `sync`, then `sync` returns the value present in the actual memory. Otherwise, `sync` returns the value recorded in the `trace`. `sync` could simply return `trace[i]`, if the value of `v1` in `sync` is restricted to primitive types. Since type of `v1` could be object or function, `trace[i].type` records the type of `v1` and `trace[i].val` stores the value or unique id of `v1` if `v1` is of primitive type or object/function type, respectively. If the type of `v1` is non-null object or function, we need to return the object or function that has the unique id recorded in `trace[i].val`. `sync` calls `syncRecord(rec, v1)` to achieve this. `syncRecord` maintains a map, `objectMap`, from unique identifiers to object/functions. If `syncRecord` discovers that the recorded unique id maps to an object/function in the `objectMap`, it returns that object/function. Otherwise, if `syncRecord` finds that the recorded unique identifier has no map in the `objectMap`, `syncRecord` does the following:

- If `v` is a fresh object/function (i.e. which has not been assigned an unique id in the current execution), `syncRecord` assigns the recorded unique id `rec.val` to the object `v` and updates `objectMap` to remember this mapping. `syncRecord` returns the object `v`.

- Otherwise, `syncRecord` has encountered an undefined value or a stale value. Therefore, `syncRecord` creates a mock empty object/function, assigns the recorded id

```
// persist trace after recording
// during replay initialize trace
// from persisted trace
var trace = [];
var i = 0, id = 0, objectMap = [];

function getRecord(v) {
  if (v !== null && (typeof v ==='object'||
      typeof v === 'function')){
    if (!v["*id*"]) v["*id*"] = ++id;
    return {type:typeof v, val:v["*id*"]}
  } else {
    return {type:typeof v, val:v};
  }
}

function syncRecord(rec, v) {
  var result = rec.val
  if(rec.val !==null && (rec.type==='object'||
      rec.type === 'function')){
     if (objectMap[rec.val])
       result = objectMap[rec.val];
     else {
      if(typeof v !== rec.type||v["*id*"])
       v = (rec.type==='object')?{}:function(){}
      v["*id*"] = rec.val;
      objectMap[rec.val] = v;
      result = v;
     }
  }
  return result
}

function sync(v1, v2) {
  i = i + 1;
  if (recording) {
    if (v1 !== v2)
      trace[i] = getRecord(v1);
    return v1;
  } else {
    if (trace[i])
      return syncRecord(trace[i], v1);
    else
      return v1;
  }
}

function enter(v) {
  i = i + 1;
  if (recording) {
    trace[i] = getRecord(v)
    trace[i].isFunCall = true
  }
}

function instrCall(f, o, a1, ..., an) {
  if (recording || isInstrumented(f))
     return call(f, o, a1, ..., an)
  else
     return replay()
}

function replay() {
  while (trace[i+1].isFunCall) {
     var f = syncRecord(trace[i+1], undefined)
     f()
  }
  return undefined
}
```

Figure 2: Record-Replay Library

to that object, updates the `objectMap`, and returns the mock object/function.

The function `replay` plays an important role in the replay phase. It ensures that any instrumented function that got invoked from an un-instrumented or native function, is called by JALANGI explicitly. The `replay` function is dependent on the `enter` function inserted at the beginning of every instrumented function. `enter` records the value of the function that is currently being executed. It also sets the field `isFunCall` of the record appended to `trace` to `true`. A true value of `trace[i].isFunCall` indicates the record appended to `trace` corresponds to the invocation of the function denoted by `trace[i].val`. Now let us see how this record is used in the replay phase. JALANGI calls `instrCall` in place of any `call` statement in the code. `instrCall`, in turn, invokes `call` if JALANGI is in the recording phase, or during replay phase when function `f` is instrumented. This ensures that JALANGI executes any function, whether instrumented or un-instrumented, normally during the recording phase, and that JALANGI only executes instrumented functions normally during the replay phase. If the function `f` inside `instrCall` is un-instrumented, then there is a possibility that `f` could have called some instrumented function in the recording phase. In order to replay the execution of those instrumented functions, JALANGI calls `replay`. `replay` first computes the function object by looking at the next record in the trace and then invokes it if `isFunCall` is true. The invocation does not pass any argument because JALANGI has no record of the arguments being passed to the function. The arguments get synced inside the function as they are being read inside the function.

JALANGI starts the replay phase by calling the `replay` function instead of calling the entry function of the application.

```
function AnnotatedValue(actual, shadow) {
  this.actual = actual;
  this.shadow = shadow;
}

function a(v) {
  if (v instanceof AnnotatedValue)
    return v.actual
  return v
}

function s(v) {
  if (v instanceof AnnotatedValue)
    return v.shadow
  return undefined
}
```

Figure 3: Annotated Value

2.2 Shadow Values and Shadow Execution

JALANGI enables a robust framework for writing dynamic program analyses through shadow values and shadow execution. A user-defined shadow execution can be performed by JALANGI during the replay phase. JALANGI only performs shadow execution of instrumented code: without instrumentation, JALANGI cannot analyze the behavior of uninstrumented or native code.

In shadow execution, JALANGI allows the replacement of any value used in the execution by an *annotated value*. The annotated value can carry extra information about the actual value. For example, an annotated value can carry taint information in a taint analysis or a symbolic expression describing the actual value in symbolic execution. In JALANGI, we denote an annotated value using an object of type `AnnotatedValue` defined in Figure 3. An object of type `Anno-`

tatedValue has two fields: the field `actual` stores the actual value and the field `shadow` stores the shadow value, i.e. extra information about the actual value. A value, say v, in JavaScript can be associated with shadow value, say s, by simply replacing v by `new AnnotatedValue`(v, s). The projection function $\underline{\texttt{a}}(v)$ returns the actual value of v, if v is an annotated value and returns v otherwise. Similarly, the projection function $\underline{\texttt{s}}(v)$ returns the shadow value associated with v if v is an annotated value and returns `undefined` otherwise.

If a JavaScript value is replaced by a user-defined annotated value during an analysis, the built-in JavaScript operations will fail. For example, if we replace the number value 53 by the annotated value `new AnnotatedValue`$(53, null)$, then addition of this value with another number, say 31, would result in `NaN` instead of 84. To avoid such situations, we instrument code so that JALANGI performs the built-in JavaScript operations on the actual values instead of the annotated values. For example, $v1$ `op` $v2$ is replaced by $\underline{\texttt{a}}(v1)$ `op` $\underline{\texttt{a}}(v2)$. Similarly, $v1[v2]$ is replaced by $\underline{\texttt{a}}(v1)[\underline{\texttt{a}}(v2)]$. The instrumentation inserted by JALANGI to perform shadow execution with shadow values along with record-replay is shown in Figure 4.

```
var v                    ==>  var v'
                              var v
                              if(anlys && anlys.literal)
                                v = anlys.literal(undefined)

v = c                    ==>  v' = v = sync(c)
                              if(anlys && anlys.literal)
                                v = anlys.literal(c)

v1 = v2 op v3            ==>  v2' = v2 = sync(v2, v2')
                              v3' = v3 = sync(v3, v3')
                              v1' = v1 = a(v2) op a(v3)
                              if(anlys && anlys.binary)
                                v1 = anlys.binary(op, v2, v3, v1)

v1 = op v2               ==>  v2' = v2 = sync(v2, v2')
                              v1' = v1 = op a(v2)
                              if(anlys && anlys.unary)
                                v1 = anlys.unary(op, v2, v1)

if v goto ℓ              ==>  v' = v = sync(v, v')
                              if(anlys && anlys.conditional)
                                anlys.conditional(v)
                              if a(v) goto ℓ

return v                 ==>  v' = v = sync(v, v')
                              return v

v1 = v2[v3]              ==>  v2' = v2 = sync(v2, v2')
                              v3' = v3 = sync(v3, v3')
                              a(v2)[a(v3) + "'"] = a(v2)[a(v3)] =
                                sync(a(v2)[a(v3)], a(v2)[a(v3) + "'"])
                              v1' = v1 = a(v2)[a(v3)]
                              if(anlys && anlys.getField)
                                v1 = anlys.getField(v2, v3, v1)

v1[v2] = v3              ==>  v1' = v1 = sync(v1, v1')
                              v2' = v2 = sync(v2, v2')
                              v3' = v3 = sync(v3, v3')
                              a(v1)[a(v2) + "'"] = a(v1)[a(v2)] = v3
                              if(anlys && anlys.putField)
                                a(v1)[a(v2)] =
                                        anlys.putField(v1, v2, v3)

v1 =                     ==>  v2' = v2 = sync(v2, v2')
  call(v2, v3, v4, ...)       v3' = v3 = sync(v3, v3')
                              v4' = v4 = sync(v4, v4')
                              ⋮
                              v1' = v1 = sync(
                                instrCall(a(v2), v3, v4, ...))
                              if(anlys && anlys.call)
                                v1 = anlys.call(v2, v3, v4, ..., v1)

{f1: v1, ...}            ==>  {f1: v1' = v1 =
                                sync(v1, v1'), ...}

[v1, ...]                ==>  [v1' = v1 = sync(v1, v1'), ...]

function v1(p1, ...){    ==>  function v1(p1, ...){
  (ℓ: Stmt)*                    enter(v1)
}                               var p1'
                                ⋮
                                (ℓ: Stmt)*
                              }
```

Figure 4: Instrumentation for Record-Replay and Shadow Execution

The instrumentation assumes that the global variable $\underline{\texttt{anlys}}$ could point to an user-defined analysis object during the replay phase. After the execution of a JavaScript statement, the corresponding method in the $\underline{\texttt{anlys}}$ object is called to perform a user-specific analysis. For example, consider the statement $v1 = v2$ `op` $v3$. After the execution of this statement in the replay phase, JALANGI calls the $v1 = \underline{\texttt{anlys}}.\underline{\texttt{binary}}(\texttt{op}, v2, v3, v1)$ to perform an analysis specific function for the binary operation `op`. For example, if $v2$ is the number 53 and $v3$ is the annotated value `new AnnotatedValue`(31,"tainted"), then after the execution of the actual statement $v1$ will be 84 and then execution of $v1 = \underline{\texttt{anlys}}.\underline{\texttt{binary}}(\texttt{op}, v2, v3, v1)$ could store `new AnnotatedValue`(84,"tainted") in $v1$ to represent the fact if one of the operands of a binary operation is tainted, then the result of the operation is also tainted. Following is another example in the context of symbolic execution. If $v2$ is the annotated value `new AnnotatedValue`$(1, "2x_1 + 1")$ and $v3$ is the annotated value `new AnnotatedValue`$(3, "x_2 - x_1")$, then after the execution of $v1 = \underline{\texttt{anlys}}.\underline{\texttt{binary}}(+, v2, v3, v1)$, where $\underline{\texttt{anlys}}$ performs symbolic execution, $v1$ will be the annotated value `new AnnotatedValue`$(4, "x_1 + x_2")$. Note that in the symbolic execution, the symbolic expression corresponding to a concrete value is represented as a string in the shadow value.

2.3 Example Analysis: Tracking Origin of null and undefined Values

In Figure 6 we describe a simple dynamic analysis using the shadow execution framework for JALANGI. The analysis tracks the origin of `null` and `undefined` in a JavaScript execution. If during an execution, access is made to the field of a `null` or `undefined` value, or if an invocation of a value which is `null` or `undefined` is encountered, the analysis could report the line number of code where the `null` or `undefined` value originated.

The analysis creates an object $\underline{\texttt{anlys}}$, where we define the methods $\underline{\texttt{literal}}$, $\underline{\texttt{getField}}$, and $\underline{\texttt{call}}$. The operations corresponding to these methods could create `null` and `undefined` values. Therefore, if the value returned by any of these operations is `null` or `undefined`, we annotate the return value with the location information. `getLocation()` returns the line number in the original code where the instrumentation was inserted by JALANGI.

The above example shows how one could implement a

```
function syncRecord(rec, tv) {
M: var v = a(tv), result = rec.val
  if(rec.val !==null && (rec.type==='object'||
      rec.type === 'function')){
     if (objectMap[rec.val])
       result = objectMap[rec.val];
     else {
      if(typeof v !== rec.type||v["*id*"])
       v = (rec.type==='object')?{}:function(){}
      v["*id*"] = rec.val;
      objectMap[rec.val] = v;
      result = v;
     }
  }
M: if (a(tv) === result)
M:   result = tv
  return result
}
```

Figure 5: Updated syncRecord for Shadow Execution. Modified lines are labeled with `M:`

```
anlys = {
  literal: function(c) {
    if (c === null || c === undefined) {
      return new AnnotatedValue(c,getLocation())
    }
  },

  getField: function(v1, v2, r) {
    if (r === null || r === undefined){
      return new AnnotatedValue(r,getLocation())
    }
  },

  call: function(f, o, a1,...,an, r) {
    if (r === null || r === undefined){
      return new AnnotatedValue(r,getLocation())
    }
  }
}
```

Figure 6: Tracking Origins of undefined and null

dynamic analyses using JALANGI. In our framework, we have implemented full concolic testing and taint analysis using shadow execution. We believe that many other dynamic analyses could be implemented easily using JALANGI.

3. EXAMPLE

Consider the example JavaScript program in Figure 7. Let us assume that the entire program is instrumented except the body of the function `foo`. The trace generated by an execution of the program in a browser is also shown in the Figure. Note that during the recording phase, we create an unique identifier for each of the objects accessed inside the body of the program. The object `document` is available in the browser, but the object never got created in the body of the program. During the replay, a mock object is created for `document` and `document["*id*"]` is set to 2, an identifier obtained from the recorded trace. `document.URL` is set to `"http://127.0.0.1/index.html"`, a string value obtained from the trace. During the recording phase, `mydoc` gets set to document inside un-instrumented code. Therefore, after the execution of `foo`, `mydoc` will contain the object document and the shadow variable `mydoc'` will still be `undefined`.

```
// un-instrumented
function foo() {
  mydoc = document;
}
// to be instrumented
var mydoc;

function myapp() {
  document.onload = function myload () {
    var url = document.URL;
    foo();
    return mydoc;
  }
}();

trace = [
    // sync function literal myapp and
    // set myapp["*id*"] = 1
 {type: "function", val: 1},
    // record enter(myapp)
 {type: "function", val: 1, isFunCall: true},
    // sync load of document and
    // set document["*id*"] = 2
 {type: "object", val: 2},
    // sync function literal myload
    //and set myload["*id*"]= 3
 {type: "function", val: 3},
    // record enter(myload)
    //where myapp is called by the event dispatcher
 {type: "function", val: 3, isFunCall: true},
    // sync getField, document["URL"]
 {type:"string",val:"http://127.0.0.1/index.html"},
    // sync function literal foo and
    // set foo["*id*"]= 4
 {type: "function", val: 4},
    // sync load of mydoc on return
 {type: "object", val: 2}
]
```

Figure 7: An example JavaScript program. Assume that the function `foo` is not instrumented. Executing the program on a browser generates the `trace`.

JALANGI will, therefore, record the value of `mydoc`, when it is returned from `myload`. During the replay, JALANGI will sync the value of `mydoc`, which it will discover in `objectMap`. The value of `mydoc` will be set to the mock object with id 2 created during the replay. Thus the replay phase will faithfully mimic the recorded execution even in a non-browser environment.

4. IMPLEMENTATION

We have implemented JALANGI in JavaScript. The code of this framework is available under Apache 2.0 open-source license at `https://github.com/SRA-SiliconValley/jalangi`. In the actual implementation, we do not transform JavaScript into the three-address code described in Section 2. Rather we modify the AST in place by replacing each operation with an equivalent function call. We perform instrumentation manually; in future, we plan to support automated instrumentation using a proxy.

Handling `eval`

JALANGI exposes the instrumentation library as a function `instrumentCode`. This enables us also to dynamically instrument any code that is created and evaluated at runtime. For example, we modify any call to `eval(s)` to `eval(instrumentCode(s))`.

Handling Exceptions

Exceptions do not pose any particular challenge in Jalangi except for uncaught exceptions being thrown from un-instrumented code. We wrap every function within a try-catch-finally block. In the catch block, we re-throw the exception. In the finally block, we call any analysis specific code corresponding to the function call.

Handling AJAX Calls and Event Handlers

Event handlers and handlers of AJAX calls appear as top-level function invocations in the recorded trace. If the handlers are instrumented, then the `replay` function defined in Figure 2 invokes them in the order in which they were invoked in the recorded execution.

In record-replay described in Figure 1, we record any literal value, any value returned by a function call, and any function value that is executed. This could still result in large amount of record data. In our implementation, we avoid recording any literal value. We only record the return value of a function, if the function is un-instrumented or native. Similarly, we avoid recording a function value at the beginning of the execution of the function, if the function is called from an instrumented function.

Concolic Testing

We have implemented concolic testing as an analysis in Jalangi. We store the symbolic expression corresponding to each concrete value in its shadow value. Concolic execution takes place during the replay phase: the shadow execution updates the shadow value of each value. We perform record-replay execution of the program for each generated input.

In our implementation of concolic testing, we handle linear integer constraints and string constraints involving concatenation, length, and regular expression matching. We also handle type constraints and a limited set of constraints over pointers. For example, if the type of an input variable is unknown, we infer the possible types of the variable by observing the operations performed on the variable.

Dynamic Taint Analysis

A dynamic taint analysis is a form information flow analysis which checks if information can flow from a specific set of memory locations, called sources, to another set of memory locations, called sink. We have implemented a simple form of dynamic taint analysis in Jalangi. In the analysis, we treat read of any field of any object, which has not previously been written by the instrumented source, as a source of taint. We treat any read of a memory location that could change the control-flow of the program as a sink. We attach taint information with the shadow value of an actual value. Taint information is propagated by implementing the various operations in the analysis. For example, if any of the operands of an operation is tainted, then we return an annotated value which is marked as tainted.

Detecting Likely Type Inconsistencies

The dynamic analysis checks if an object/function created at a given program location can assume multiple inconsistent types. It computes the types of object and function values created at each definition site in the program. Specifically, the analysis associates every object/function value with the static program location where the object/function value got created. It also maps each such program location to the type that the objects/functions created at the location can assume during the course of the execution. If an object or a function value defined at a program location has been observed to assume more than one type during the execution, the analysis reports the program location along with the observed types. Sometimes these kind of type inconsistencies could point us to a potential bug in the program. We have noticed such issues in two SunSpider benchmark programs.

Simple Object Allocation Profiler

This dynamic analysis records the number of objects created at a given allocation site and how often the fields of the objects created at a given allocation site has been accessed. The analysis also tracks if the objects' fields have been updated, that is the analysis tracks if the objects created at a given allocation site are read-only or a constant. The analysis reports the maximum and average difference between the object creation time and the most recent access time of the object . Time is reported in terms of the number of instructions being executed. If an allocation site creates too many contant objects, then it could lead to performance issues. We have found such an issue in one of the web applications in our benchmark suite.

Limitations

Jalangi implementation has the following known limitations which can prevent it from doing high-fidelity replay in some scenarios. Some of these limitations arise in ECMAScript 5. Jalangi will fail to record if a JavaScript engine disallows access to `arguments.callee` at the beginning of a function call. This is a problem if an entire script is made to run in *strict mode*. Jalangi cannot handle updates made through the `setter` method of an object in ECMAScript 5. Jalangi cannot handle the `with` construct properly. Jalangi cannot track implicit type conversion of externally created objects to primitive types. `JSON.stringify` cannot accurately record a floating point number, which could lead to imprecision during replay. We believe that these limitations could be addressed in a future release.

5. EVALUATION

We next report our results of evaluating Jalangi on several benchmark programs. In our evaluation, we focussed on four aspects: 1) ease of writing dynamic analyses, 2) fidelity and robustness of record-replay, 3) performance of Jalangi, and 4) programming issues detected during dynamic analyses.

5.1 Ease of Writing Dynamic Analyses

We have written five dynamic analyses and a condition coverage tool on top of Jalangi. The condition coverage tool has 47 lines of JavaScript code, the origin tracker for null and undefined has 61 lines of JavaScript code, taint analysis has 68 lines of code, the object allocation tracker has 174 lines of code, the type inconsistency checker has 543 lines of code, and concolic testing has 2225 lines of code. In comparison, a concolic testing tool for Java with lesser functionalities had more than 20,000 lines of code. Even though number of lines of code is not a good measure for the ease of writing a dynamic analysis, it provides a rough estimate of the complexity of writing an analysis on top of JALANGI.

5.2 Fidelity and Robustness

By fidelity, we mean the similarity between recording and replay executions. By robustness, we mean the ability of JALANGI to handle a program without introducing any errors or exceptions of its own. To check fidelity of JALANGI, we recoded all the memory loads both in record and replay phases and checked if the two sequences of loads are the same. Despite the limitations described in the previous section, we managed to run JALANGI without any error on all programs that we considered for evaluation. and JALANGI produced exactly the same sequence of memory loads and followed exactly the same execution paths.

5.3 Performance of JALANGI

We performed record-replay on 26 programs in the JavaScript SunSpider (`http://www.webkit.org/perf/sunspider/sunspider.html`) benchmark suite and on five web apps written for the Tizen OS using HTML5/JavaScript (`https://developer.tizen.org/downloads/sample-web-applications`). The web apps include *annex*—a two-player strategy game, *shopping list*—which uses local storage API of HTML5, *scientific calculator*, *go*—a two-player strategy game, and *tenframe*—a math-based three-game combo for kids. During the replay phase of these benchmark programs, we ran three dynamic analyses: no analysis (denoted by *empty*), tracking origins of null and undefined (denoted by *track*), and a taint analysis (denoted by *taint*). We report the overhead associated with the recording and replay phases in Table 1. The experiments were performed on a laptop with 2.3 GHz Intel Core i7 and 8 GB RAM. We ran the web apps on Chrome 25 and performed the replay executions on node.js 0.8.14.

The SunSpider benchmarks have relatively small number of lines of code, but they perform CPU intensive computations. The web apps perform both CPU intensive computations and manipulation of the DOM. We didn't measure the slowdown of the web apps because these are mostly interactive applications. For the SunSpider benchmark suite, we observed an average slowdown of 26X during the recording phase with a minimum of 1.5X and a maximum of 93X. On the *empty* analysis during the replay phase, we observed an average slowdown of 30X with a minimum of 1.5X and a maximum of 93X. *Track* analysis showed an average slowdown of 32.75X with a minimum of 1.5X and a maximum of 96X. The slowdown in recording is 2X-3X lower than that of PinPlay [22] and the slowdown in the analysis phase is slightly higher than slowdown noticed in valgrind [20], a heavy-weight dynamic analysis tool for x86. We didn't make any effort to optimize our implementation, but we believe suitable optimizations could further reduce the overhead. For some programs in the SunSpider suite we noticed that the number values recorded is quite high and recording phase has higher overhead than replay. This is because these programs made many expensive native calls. The return values of those calls were recorded. Replay skipped the execution of those native calls, so we noticed lower overhead for replay.

In JALANGI, if we record every memory load, then we notice a slowdown of 300X -1000X. Our proposed use of shadow memory significantly reduces the number of loads that we had to record for a faithful replay. The column titled "% of Loads Recorded" reports the reduction in percentage. We noticed an average reduction of 6.52% and a median reduction of 0.73%. Programs doing a lot of native calls and performing frequent manipulation of the DOM resulted in large recoding of the memory loads.

Based on our evaluation, we are optimistic about the utility of JALANGI as a tool framework aiding web developers. We believe that the utility offered by JALANGI is much more valuable compared to the additional performance penalty that the developers observe. Moreover, this additional penalty would be incurred only during the development phase, and the instrumentation introduced by JALANGI would not become a part of the actual applications deployed to users.

5.4 Performance of concolic testing

We ran concolic testing on several programs ported from a concolic testing engine for Java. Even though concolic testing is not the focus of this paper, we report the results of running concolic testing on a small program (shown in Table 2), which has complex string operations involving integers, string length, regular expression matching, and concatenation. This program is a slight variant of the program used as a case study in [5]. In concolic testing, we only use the theory of linear integers of CVC3 [4] and model string operations using this theory. We noticed an average slowdown of 145X during concolic execution with a maximum slowdown of 613X and a minimum slowdown of 1.4X. The recording phases showed a slowdown of 1.2X. The slowdown in the concolic execution phase is mostly due to the calls to the SMT solver.

5.5 Issues Detected by Dynamic Analyses

The likely type inconsistency checker noticed that the function `safe_add(x, y)` (shown below) in `crypto-sha1.js` of the SunSpider benchmark suite is mostly called with both of its arguments set to number, but at one location it was invoked with the second argument set to `undefined`. We believe that this could be an unintended behavior.

```
function safe_add(x, y)
{
  var lsw = (x & 0xFFFF) + (y & 0xFFFF);
  var msw = (x >> 16) + (y >> 16) + (lsw >> 16);
  return (msw << 16) | (lsw & 0xFFFF);
}
```

The likely type inconsistency checker reported that the function `CreateP` (shown below) in `3d-cube.js` of the SunSpider benchmark suite is mostly called as a constructor, but at one location it was invoked as a function. As result of the function call, the program creates an unnecessary `V` field in the global object. We believe that this call is a possible programming error.

```
function CreateP(X,Y,Z) {
  this.V = [X,Y,Z,1];
}
```

The object allocation profiler noticed that the method `getValue(place, _board)` in the Annex game webapp creates a constant object containing at least 64 numbers thousands of times. We believe that such unnecessary creation of the constant object can be avoided by hoisting the object creation outside the method.

Benchmark	LOC	Records	fLoads	SlowR	Slowdown in Replay		
					empty	taint	track
3d-cube	339	3670	0.09	18.33	25.16	28.67	26
3d-morph	56	6	< 0.01	18.2	33.2	35.83	33.6
3d-raytrace	443	79791	2.68	38.17	29.05	30.5	35
b-trees	52	146048	18.26	57.8	40	42.4	42.8
fannkuch	68	246	< 0.01	40.6	76.4	73	80.4
nbody	170	78	< 0.01	19	25.8	25.67	24.16
nsieve	39	5	< 0.01	16.4	23.6	30	24.2
3bit-in-byte	38	1	< 0.01	16.6	29	31	30.2
bits-in-byte	26	1	< 0.01	25	25	51.4	47
bitwise-and	31	1	< 0.01	12.83	21.83	29.2	26.2
controlflow	25	1	< 0.01	20	33.2	34.6	28.33
crypto-md5	288	42	< 0.01	12	18	22.2	22
crypto-sha1	225	52	< 0.01	13.4	19.4	21	21.2
date-tofte	300	32018	1.59	92.16	92.67	92.83	95.5
date-xparb	418	95715	17.81	29.83	21	22.67	25.67
math-cordic	101	8	< 0.01	29.6	35.6	45.4	40.17
partial-sums	33	5	< 0.01	14.6	23.4	22.16	23.8
spectral-norm	51	15	< 0.01	19.8	25.2	29.2	29.4
regexp-dna	1714	42	21	2	4	3.17	3.8
string-fasta	90	56947	2.77	40.17	30.33	34.5	38.6
string-tagcloud	266	117577	16.23	51.42	50.86	44	42.8
string-unpack	67	193057	33.21	29.88	13.25	13.75	17
nsieve-bits	35	3	< 0.01	20	36.6	45.4	40
crypto-aes	425	23926	0.73	19	21	23.67	23
string-validate	90	60	13.27	1.5	1.5	1.4	1.5
string-base64	136	40965	3.38	25	27.2	29.6	29.2
annex	9663	87623	0.86	-	-	-	-
calculator	787	1288	17.64	-	-	-	-
go	10,039	114609	0.97	-	-	-	-
tenframe	1491	4656	28.89	-	-	-	-
shopping	5397	1144	22.79	-	-	-	-

Table 1: Results: "Records" column reports number of values of recorded, "fLoads" reports % of loads that were recorded, "SlowR" reports slowdown during recording compared to normal execution.

```
function isValidQuery(str)
{
// (1) check that str contains "/" followed
// by no "/" and containing "?q=..."
 var slash = str.lastIndexOf('/');
 if (slash < 0){
   return false;
 }
 var rest = str.substring(slash + 1);
 if(!(RegExp('\\\?q=[a-zA-Z]+')).test(rest)){
   return false;
 }
// (2) Check that str starts with "http://"
 if (str.indexOf("http://")!==0){
   return false;
 }
// (3) Take the string after "http://"
// strip the "www." off if present
 var t=str.substring("http://".length,slash);
 if (t.indexOf("www.")===0){
   t = t.substring("www.".length);
 }
// (4) Check that the rest is either
// "live.com" or "google.com"
 if(t !=="google.com" && t!=="live.com"){
   return false;
 }
 // str survived all checks
 return true;
}
```

Table 2: Sample code for evaluating performance of concolic testing

6. RELATED WORK

There is a large body of work on record-replay systems (see [10, 11] for survey of this area). In this section, we discuss the papers that are closely related to JALANGI.

JSBench [26] is a technique for creating JavaScript benchmarks using record-replay mechanisms. JSBench captures the interactions of an web application with its surrounding execution environment. It then creates a replayable packaged JavaScript benchmark which can execute in the absence of the surrounding environment. JSBench does not capture all memory loads or memory loads that could potentially be modified by eval or un-instrumented code. Therefore, JSBench could function improperly in the presence of un-instrumented code. JALANGI alleviates this problem by maintaining shadow memory. Selective record-replay techniques have been proposed for Java [15, 28]. Unlike JALANGI, these techniques statically identify the program locations where values need to be recorded.

PinPlay [22], built on top of dynamic instrumentation framework PIN [16] for x86, uses ideas similar to shadow memory [19] to reduce the number of memory logs. PinPlay keeps shadow memory, which they call UserMem, in sync with the actual memory at the byte and word level. In JavaScript it is not possible to keep track of memory at byte and word level. JALANGI uses a novel technique based on unique identifiers to record and sync objects and functions and uses mock objects to mimic behaviors of objects created outside instrumented code.

Mugshot [18] is another record-replay system for JavaScript that captures all events in a JavaScript program and allows developers to deterministically replay past executions of web applications. Ripley [33] replicates execution of a client-side JavaScript program on a server side replica to automatically preserve the integrity of a distributed computation. DoDOM [23] records user interaction sequences with web applications repeatedly, executes the application under the captured sequence of user actions and observes its behavior. WaRR [2] records user interactions with an web application and uses the recorded interaction trace to perform high-fidelity replay of the web application. Contrast to these techniques, JALANGI performs selective record-replay instead of targeting full record-replay of an web application.

The idea of shadow values in the context of x86 binaries has been previously proposed in [20, 36] and has been used in several analysis tools [36, 21, 6, 8]. Instead of creating a separate address space for shadow values, JALANGI wraps each JavaScript value in an object of type `AnnotatedValue`. This simple technique is possible due to the dynamic nature of JavaScript.

In the recent years, several static [35, 14, 1, 12, 34, 32] and dynamic analyses [24, 27, 3, 17] tools for JavaScript have been proposed. Richards et al. [27] observed that dynamic features are widely used in JavaScript programs. These dynamic features make static analysis of JavaScript applications hard and previous research efforts have either ignored or made incorrect assumptions regarding these dynamic features. Dynamic analysis tools developed for JavaScript include tools for testing [3, 29], race detection [24], and security analysis [33]. However, there exists no dynamic analysis framework for JavaScript similar to valgrind [20], PIN [16], DynamoRIO [7] for x86. JALANGI tries to fill this gap by providing a dynamic analysis framework in which one could easily prototype and build sophisticated browser-independent dynamic program analyses for JavaScript.

7. REFERENCES

[1] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for javascript. In *19th European conference on Object-Oriented Programming*, ECOOP'05, pages 428–452, 2005.

[2] S. Andrica and G. Candea. Warr: A tool for high-fidelity web application record and replay. In *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 403–410, 2011.

[3] S. Artzi, J. Dolby, S. H. Jensen, A. Møller, and F. Tip. A framework for automated testing of javascript web applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 571–580. ACM, 2011.

[4] C. Barrett and C. Tinelli. CVC3. In 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302, 2007.

[5] N. Bjørner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In *15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '09, pages 307–321. Springer-Verlag, 2009.

[6] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley. Tracking bad apples: reporting the origin of null and undefined value errors. In *ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 405–422, 2007.

[7] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, 2003.

[8] M. Burrows, S. Freund, and J. Wiener. Run-time type checking for binary programs. In *Compiler Construction*, volume 2622 of *LNCS*, pages 90–105. Springer, 2003.

[9] J. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *International symposium on Software testing and analysis*, ISSTA '07, pages 196–206. ACM, 2007.

[10] F. Cornelis, A. Georges, M. Christiaens, M. Ronsse, T. Ghesquiere, and K. D. Bosschere. A taxonomy of execution replay systems. In *International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, 2003.

[11] C. Dionne, M. Feeley, and J. Desbiens. A taxonomy of distributed debuggers based on execution replay. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 203–214, 1996.

[12] A. Feldthaus, M. Schaefer, M. Sridharan, J. Dolby, and F. Tip. Efficient construction of approximate call graphs for javascript ide services. In *International Conference on Software Engineering*, ICSE '13, 2013.

[13] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *PLDI'05*, June 2005.

[14] S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *17th international conference on Static analysis*, SAS'10, pages 320–339, 2010.

[15] S. Joshi and A. Orso. Scarpe: A technique and tool for selective capture and replay of program executions. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 234–243, 2007.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200. ACM, 2005.

[17] A. Mesbah and A. van Deursen. Invariant-based automatic testing of ajax user interfaces. In *31st International Conference on Software Engineering*, ICSE '09, pages 210–220. IEEE, 2009.

[18] J. Mickens, J. Elson, and J. Howell. Mugshot: deterministic capture and replay for javascript applications. In *7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 11–11, 2010.

[19] S. Narayanasamy, C. Pereira, H. Patil, R. Cohn, and B. Calder. Automatic logging of operating system effects to guide application-level architecture simulation. In *Proceedings of the joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '06/Performance '06, pages 216–227. ACM, 2006.

[20] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100. ACM, 2007.

[21] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium*, NDSS '05, 2005.

[22] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 2–11, 2010.

[23] K. Pattabiraman and B. Zorn. Dodom: Leveraging dom invariants for web 2.0 application robustness testing. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*, ISSRE '10, pages 191–200, 2010.

[24] B. Petrov, M. Vechev, M. Sridharan, and J. Dolby. Race detection for web applications. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 251–262. ACM, 2012.

[25] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. Jsmeter: comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 3–3, 2010.

[26] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 677–694. ACM, 2011.

[27] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12. ACM, 2010.

[28] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 114–123, New York, NY, USA, 2005. ACM.

[29] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 513–528. IEEE, 2010.

[30] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE'05*, Sep 2005.

[31] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Annual conference on USENIX Annual Technical Conference*,

ATEC '05, pages 2–2, 2005.

[32] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip. Correlation tracking for points-to analysis of javascript. In *26th European conference on Object-Oriented Programming*, ECOOP'12, pages 435–458, 2012.

[33] K. Vikram, A. Prateek, and B. Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *16th ACM conference on Computer and communications security*, CCS '09, pages 173–186. ACM, 2009.

[34] S. Wei and B. G. Ryder. A practical blended analysis for dynamic features in javascript. Technical report, Department of Computer Science, Virginia Tech., 2012.

[35] D. Yu, A. Chander, N. Islam, and I. Serikov. Javascript instrumentation for browser security. In *ACM symposium on Principles of programming languages*, POPL '07, pages 237–249, 2007.

[36] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and scalable memory shadowing. *8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 22–31, 2010.