Poster: Dynamic Analysis Using JavaScript Proxies

Laurent Christophe*, Coen De Roover*, Wolfgang De Meuter*
*Software Languages Lab, Vrije Universiteit Brussel, Brussels, Belgium

***Abstract*—JavaScript has become a popular programming language. However, its highly dynamic nature encumbers static analysis for quality assurance purposes. Only dynamic techniques such as concolic testing seem to cope. Often, these involve an instrumentation phase in which source code is extended with analysis-specific concerns. The corresponding implementations represent a duplication of engineering efforts. To facilitate developing dynamic analyses for JavaScript, we introduce Aran; a general-purpose JavaScript instrumenter that takes advantage of proxies, a recent addition to the JavaScript reflection APIs.**

I. Introduction

JavaScript has become ubiquitous on server and client tiers of contemporary web applications. Recent advances in browser technologies have encouraged the use of JavaScript in Rich Internet Applications. JavaScript offers many dynamic and reflective features (e.g., runtime code evaluation with `eval`, scope chain modification through `with`, a dynamic inheritance hierarchy) which makes it hard to analyze JavaScript programs statically [4], [2]. However, dynamic analyses such as taint analysis [7], concolic testing [9] and performance profiling [3] seem to cope. Existing dynamic analyses each use their own instrumentation mechanism which represent non-negligible engineering efforts. In particular, correctly supporting the whole of JavaScript can be challenging. In this work, we investigate the use of a new JavaScript reflection API, proxies, as the foundation for a general-purpose JavaScript instrumenter. The motivation of our work is to facilitate the process of building dynamic analysis for full JavaScript. General-purpose instrumenters for JavaScript have already been proposed (e.g., [5], [1]), but to the best of our knowledge we are the first to take advantage of the new proxy API.

II. Harmony Proxies

Proxies are a powerful reflection means discussed in the upcoming 6th specification of ECMAScript named Harmony. Based on the work of Van Cutsem et. al [8], they enable intercepting object-related operations such as property accesses without having to resort to metacircular interpretation. Despite still being under discussion, some environments already support Harmony proxies; e.g., Node (with the `--harmony` flag) and Firefox. JavaScript environments supporting proxies feature a constructor named `Proxy` which takes a handler object that provides traps for operations on a given target object. For instance, the below snippet creates a proxy object that will log all property reads and writes:

```
var p = new Proxy(targetObject, {
  get: function (o,k)   { log("get "+k); return o[k];   }
  set: function (o,k,v) { log("set "+k); return o[k]=v; }
})
```

TABLE I
Complementary traps.

Code	Instrumented Code	Trap
`"foo"`	`wrap("foo")`	`wrap(Data)`
`x?y:z`	`tobool($x)?$y:$z`	`tobool(val)`
`eval(x)`	`eval(compile(tostring($x)))`	`tostring(val)`
`{}`	`object($Object.prototype)`	`object(proto)`
`[]`	`array()`	`array()`
`function(){}`	`lambda(function(){})`	`lambda(Fct)`
`!x`	`unary("!", $x)`	`unary(Op,arg)`
`x+y`	`binary("+", $x, $y)`	`binary(Op,l,r)`
		`global(path,val)`
		`unwrap(data)`

Proxies open up interesting possibilities with respect to instrumentation:

1) Object-related operations can be intercepted by simply wrapping every newly created object in a proxy.
2) Accesses to the pre-existing global object can be intercepted by shadowing it consistently with its proxy through the `with` construct (cf. Section IV-A).
3) Certain insights about the behavior of non-instrumented functions (e.g., form third-party libraries) can be obtained by passing them proxied objects as arguments.

However, proxies alone do not suffice for developing a dynamic analysis. Proxies still need to be introduced in the analyzed code. Moreover, the dynamic analysis needs to be notified of calls to JavaScript primitives which cannot be intercepted through proxies. We achieve both through a source-to-source program transformation, accompanied by a runtime that can be customized by developer-provided traps: Aran.

III. The Aran Instrumenter

The Aran instrumenter and runtime, available on GitHub[1], can be steered through three kinds of traps: (i) traps from the Harmony proxy API [8] (ii) complementary traps listed in Table I (iii) generic observer traps that expose information about each statement or expression that is about to be executed.

The trap `global(path, value)` provides a means for the analyst to customize the handling of built-in functions. Within these handlers, developers can either return an object or a primitive value. Our framework takes care of automatically wrapping the return value using an analysis-specific proxy or datastructure. The constructor for the former is constructed through a recursive descent of the value's properties. The constructor for the latter can be specified through the `wrap` trap. The default handling of built-in functions corresponds to the following:

[1]https://github.com/lachrist/aran

```
globalObjectShadow[wrap("parseInt")] = lambda(function (
    string, radix) {
  return wrap(global.parseInt(unwrap(string), unwrap(radix)
    ))
}))
```

Here, the handler delegates to the built-in after unwrapping its arguments. Note that our design enables constructing heavy-weight dynamic analysis such as concolic testing. For instance, the shadowing of the global object enables implementing the symbolic counterpart of primitive functions while the `toBool` trap enables accumulating conditionals to the path constraint.

IV. Instrumentation Pitfalls and Workarounds

A. Shadowing the Global Object

In JavaScript, the pre-existing global object is the common root of all scope chains. Among others, it provides access to the DOM that is rendered by the browser. Access from the instrumented code to the global object should be controlled:

1) The analyst might want to intercept global object functionality (e.g., to record the number of times DOM elements are fetched by class name).
2) The inner mechanisms of the instrumenter (e.g., traps, parsers and stack accessors) present in the global object should be protected from tampering.
3) Having the global object completely shielded from the instrumented code enables multiple instrumentations to be active within the same JavaScript execution instance.

In other words, a sandbox needs to be constructed by shadowing the global object consistently with a proxy. To this end, we wrap the instrumented code in a `with` construct:

```
var globalObjectShadow = ...
var proxy = new Proxy(globalObjectShadow, {
  has: function (sb, key) {
    if (key in sb) { return true; }
    throw new Error("Undefined reference: "+key);
  }
});
with (proxy) {
  // Sandboxed execution, outer variables are unaccessible.
}
```

The `with` construct enables using a specific object as the starting scope for identifier lookup. For instance, evaluating `with({a:1}) {console.log(a+2)}` will print out `3` to the console. In the above example, accesses to undeclared identifiers within the `with` body (e.g., DOM-related built-ins) will trigger the `has` handler of the sandbox and cause an exception to be thrown.

The only remaining door to the underlying global object is the `this` keyword. Our source-to-source transformation therefore replaces its occurrences with a reference to the sandbox. An identifier mangling, as evidenced by the `$`-signs in Table I, separates the identifiers of our traps from user-defined ones.

B. Sound desugaring of complex expressions

The traps depicted in Table I have been designed as a minimal complement to the existing proxy API. Rather than registering a trap for intercepting post-increment expressions, for instance, it suffices to register a trap for the corresponding addition on the right-hand side of assignments. This requires a desugaring from complex expressions to a composite of simpler expressions in a manner that is true to the actual JavaScript semantics. We have used a stack-based memory for storing the composite's intermediate results. Some example desugarings are listed below:

- Post/pre increment/decrement expressions such as `x++` become assignments `($x=binary("+",push($x),1),pop())`. Note that the generic trap for assignments will notify the analysis when their execution is about to begin.
- Object literals become a call to the object trap, followed by calls to property definition traps: e.g., `{a:1}` becomes `(push(object($Object.prototype)), get()[wrap("a")]=wrap("1")),pop())`.

Note that the configuration of the stack before and after the evaluation of a desugared expression should be the same. This invariant has to be upheld in the presence of exceptional control flow. To this end, we push a marker at the beginning of the execution in a `try` and pop all items above the last marker in its corresponding `finally`. For instance, `try {f()} catch (e) {g()}` becomes `try {mark(),apply(f)} catch ($e) {apply(g)} finally {unmark()}`.

V. Conclusion and Future Work

We have presented Aran, a general-purpose instrumenter based on the new proxy API for JavaScript. We have already used Aran to implement different kinds of tracing. In the future, we plan to further test the instrumenter in heavy-weight dynamic analyses such as concolic testing. We will also conduct experiments to properly assess the performance overhead of our proxy-based approach.

References

[1] James W Mickens, Jeremy Elson, and Jon Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, volume 10, pages 159–174, 2010.

[2] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G Zorn. Jsmeter: Comparing the behavior of javascript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 3–3. USENIX Association, 2010.

[3] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of javascript benchmarks. *ACM SIGPLAN Notices*, 46(10):677–694, 2011.

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

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

[6] Bastian Steinert, Lauritz Thamsen, Tim Felgentreff, and Robert Hirschfeld. Object versioning to support recovery needs: Using proxies to preserve previous development states in lively. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 113–124, New York, NY, USA, 2014. ACM.

[7] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44(6):87–97, 2009.

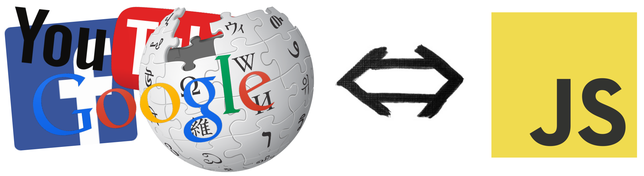
[8] Tom Van Cutsem and Mark S Miller. Proxies: design principles for robust object-oriented intercession apis. In *ACM Sigplan Notices*, volume 45, pages 59–72. ACM, 2010.

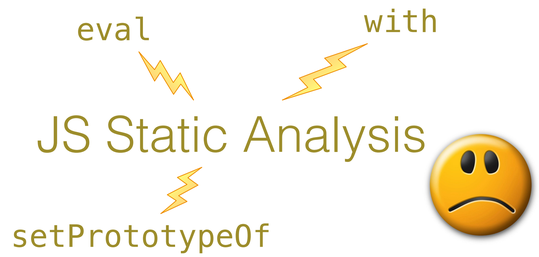
[9] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 249–260. ACM, 2008.

Dynamic Analysis Using JavaScript Proxies

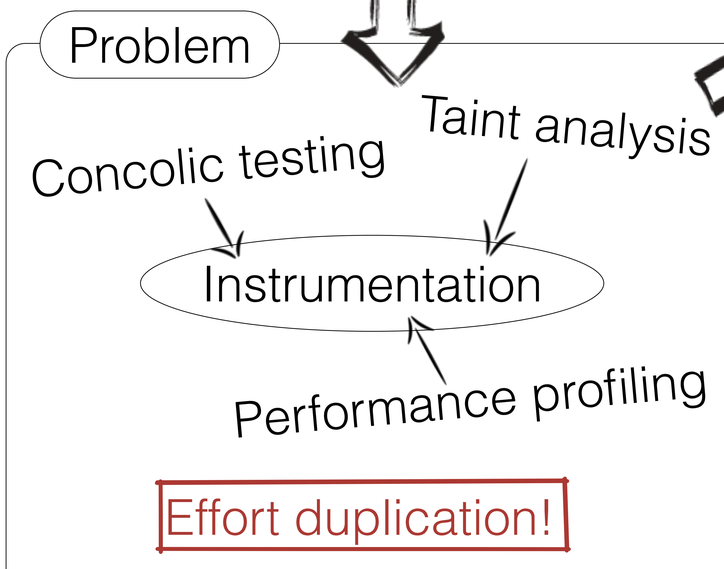
Laurent Christophe Coen De Roover Wolfgang De Meuter

Context





Problem

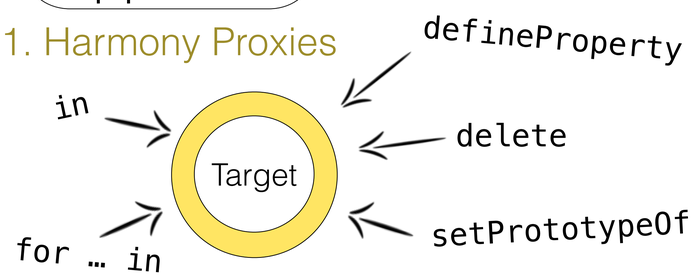


Effort duplication!

JavaScript dynamic analysis tools often use their own specific source-to-source code transformation.

Approach

1. Harmony Proxies



2. Complementary Traps

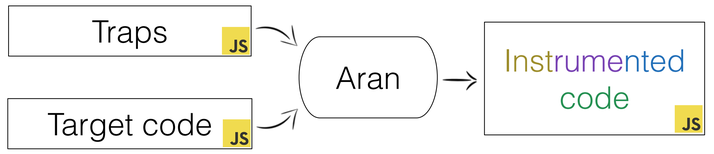
```
wrap(Data)
unwrap(data)
global(path,val)

object(proto)
array()
lambda(Fct)

toBool(val)
toString(val)
unary(Op,arg)
binary(Op,l,r)
```

3. Global Object Shadowing

```
var proxiedShadow = new Proxy(shadow, {
  get: function (s,k) {return s[k.substring(1)]},
  set: function (s,k,v) {return s[k.substring(1)]=v},
  delete: function (s,k) {return delete s[k.substring(1)]},
  has: function (s, k) {
    if (k.substring(0,1)===“$”) {return false}
    if (k.substring(1) in s) {return true}
    throw new Error(“Reference error”);
  }
})
with (proxy) { /* global shadowed */ }
```

4. Stack-based Memory

```
f() || g()
        ↘ toBool(push(f()) ? pop() : (pop(),g())
```



Example

Traps for Symbolic Execution

```
var i = 0;
var traps = {};
traps.wrap = function (data) { return {raw:data, sym:null} }
traps.unwrap = function (data) { return data.raw }
traps.toBool = function (val) {
  if (val.sym) { console.log("Path constraint: "+val.sym) }
  return Boolean(val.raw);
}
traps.object = function (proto) {
  return {raw:Object.create(proto), sym:null}
}
traps.global = function (path) {
  if (path.join()==="valprompt") {
    return function (msg, def) {
      console.log("New symbol: x"+(++i));
      return {raw:prompt(msg.raw, def.raw), sym:"x"+i}
    }
  }
}
traps.binary = function (op,l,r) {
  var sym = (l.sym||l.raw)+op+(r.sym||r.raw)
  return {
    raw: eval(l.raw+op+r.raw),
    sym: (l.sym||r.sym) ? sym : null
  }
}
```

Incomplete trap set for gathering symbolic path constraints

```
var msg = "Enter your birthdate";
var user = {birthdate:prompt(msg)};
var age = 2015 - user.birthdate;
if (age > 18) {
  alert("Welcome");
} else {
  alert("Go away");
}
```

Instrumentation

```
with (proxy) {
  var $msg = wrap("Enter your birthdate");
  var $user = (
    push(object($Object.prototype)),
    get().birthdate=$prompt($msg),
    pop());
  var $age = binary("-", wrap(2015), $user.birthdate)
  if (toBool(binary(">", $age, wrap(18)))) {
    $alert(wrap("Welcome"));
  } else {
    $alert(wrap("Go away"));
  }
}
```

more info & download: **www.github.com/lachrist/aran**

lachrist@vub.ac.be

Software Languages Lab

Vrije Universiteit Brussel