CONSCRIPT: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser

Benjamin Livshits
Microsoft Research
livshits@microsoft.com

Leo Meyerovich
University of California, Berkeley
lmeyerov@eecs.berkeley.edu

December 3, 2009

Microsoft Research Technical Report
MSR-TR-2009-158

Abstract

Much of the power of modern Web comes from the ability of a Web page to combine contents and JavaScript code from disparate servers on the same page. While the ability to create such mash-ups is attractive for both the user and the developer because of extra functionality, because of code inclusion, the hosting site effectively opens itself up for attacks and poor programming practices within every JavaScript library or API it chooses to use. In other words, expressiveness comes at the price of losing control. To regain the control, it is therefore valuable to provide means for the hosting page to *restrict* the behavior of the code that it may include.

This paper presents CONSCRIPT, an client-side advice implementation for security, built on top of Internet Explorer 8[a]. CONSCRIPT allows the hosting page to express fine-grained application-specific security policies that are enforced at runtime. In addition to presenting 17 widely-ranging security and reliability policies that CONSCRIPT enables, we also show how policies can be generated automatically through static analysis of server-side code or runtime analysis of client-side code. We also present a type system that helps ensure correctness of CONSCRIPT policies.

To show the practicality of CONSCRIPT in a range of settings, we compare the overhead of CONSCRIPT enforcement and conclude that it is significantly lower than that of other systems proposed in the literature, both on micro-benchmarks as well as large, widely-used applications such as MSN, GMail, Google Maps, and Live Desktop.

[a] The name CONSCRIPT has been chosen to reflect our desire to restrict malicious script.

Abstract

Much of the power of modern Web comes from the ability of a Web page to combine contents and JavaScript code from disparate servers on the same page. While the ability to create such mash-ups is attractive for both the user and the developer because of extra functionality, because of code inclusion, the hosting site effectively opens itself up for attacks and poor programming practices within every JavaScript library or API it chooses to use. In other words, expressiveness comes at the price of losing control. To regain the control, it is therefore valuable to provide means for the hosting page to *restrict* the behavior of the code that it may include.

This paper presents CONSCRIPT, an client-side advice implementation for security, built on top of Internet Explorer 8[a]. CONSCRIPT allows the hosting page to express fine-grained application-specific security policies that are enforced at runtime. In addition to presenting 17 widely-ranging security and reliability policies that CONSCRIPT enables, we also show how policies can be generated automatically through static analysis of server-side code or runtime analysis of client-side code. We also present a type system that helps ensure correctness of CONSCRIPT policies.

To show the practicality of CONSCRIPT in a range of settings, we compare the overhead of CONSCRIPT enforcement and conclude that it is significantly lower than that of other systems proposed in the literature, both on micro-benchmarks as well as large, widely-used applications such as MSN, GMail, Google Maps, and Live Desktop.

[a] The name CONSCRIPT has been chosen to reflect our desire to restrict malicious script.

1 Introduction

Much of the power of modern Web comes from the ability of a Web page to combine HTML and JavaScript code from disparate servers on the same page. For instance, a Yelp! page describing a restaurant uses APIs from Google Maps to show the restaurant's location, jQuery libraries to provide visual effects, and Yelp APIs to obtain the actual review and rating information. While the ability to create such client-side mashups within the same page is attractive for both the user and the developer because of the extra functionality this provides, because of including untrusted JavaScript code, the hosting page effectively opens itself up to attacks and poor programming practices from every JavaScript library or API it uses. For instance, an included library might perform a prototype hijacking attack [2], drastically redefining the behavior of the remainder of the JavaScript code run on the page.

CONSCRIPT, a browser-based *aspect system* for security proposed in this paper, focuses on empowering the hosting page to carefully *constrain* the code it executes. For example, the hosting page may restrict the use of `eval` to JSON only, restrict cross-frame communication or cross-domain requests, allow only white-listed script to be loaded, limit popup window construction, limit JavaScript access to cookies, disallow dynamic IFRAME creations, etc. These constraints take the form of fine-grained policies expressed as JavaScript aspects that the hosting page can use to change the behavior of subsequent code. In CONSCRIPT, this kind of behavior augmentation is done via the script include tag to provide a policy as follows:

```
<SCRIPT SRC="script.js" POLICY="(function () {...}">
```

With CONSCRIPT, the first general browser-based policy enforcement mechanism for JavaScript to our knowledge, at a relatively low cost of several hundred lines of code added to the JavaScript engine, we gain vast expressive power. This paper presents 17 widely-ranging security and reliability policies that CONSCRIPT enables. To collect these policies, we studied bugs and anti-patterns in both "raw" JavaScript as well as popular JavaScript libraries such as jQuery. We also found bugs in and have rewritten many of the policies previously published in the literature [23, 33] in CONSCRIPT. We discovered that in many cases a few lines of policy code can be used instead of a new, specialized HTML tag. Our experience demonstrates that CONSCRIPT provides a general enforcement mechanism for a wide range of application-level security policies. We also show how classes of CONSCRIPT policies can be generated automatically, with static analysis of server-side code or runtime analysis of client-side code, removing the burden on the developer for specifying the right policy by hand. Finally, we propose a type system that makes it considerably easier to avoid common errors in policies.

We built CONSCRIPT by modifying the JavaScript interpreter in the Internet Explorer 8 Web browser. This paper describes our implementation, correctness considerations one has to take into account when writing CONSCRIPT policies, as well as the results of our evaluation on a range of benchmarks, both small programs and large-scale applications such as MSN, GMail, and Live Desktop.

1.1 Contributions

This paper makes the following contributions.

- **Security aspects in the browser.** We present a case for the use of aspects for enforcement of rich application-specific policies by the browser. Unlike previous aspect systems for the Web and dynamic languages, we advocate *deep aspects* that are directly supported by the JavaScript and browser runtimes. Modifying the JavaScript engine allows us to easily enforce properties that are difficult or impossible to fully enforce otherwise.
- **Correctness checking for aspects.** CONSCRIPT proposes static and runtime validation strategies that ensure that aspects cannot be subverted through common attack vectors found in the literature.
- **Policies.** We present 17 wide-ranging security and reliability policies. We show how to concisely express these policies in CONSCRIPT, often with only several of JavaScript code. These policies fall intro the broad categories of controlling script introduction, imposing communication restrictions, limiting dangerous DOM interactions, and restricting API use. To our knowledge, this is the most comprehensive catalog of application-level security policies for JavaScript available to date.
- **Automatic policy generation.** To further ease the policy specificiation burden on developers, we present two strategies for *automatically* producing CONSCRIPT policies through static or runtime analysis.
- **Evaluation.** We implemented technique described in this paper in the context of Internet Explorer 8. We assess the performance overhead of our client-side enforcement strategy on the overall program execution of real programs such as Google Maps and Live Desktop, as well as a set of JavaScript micro-benchmarks previously used by other researchers. We conclude that CONSCRIPT results in runtime enforcement overheads that hover around 1% for most large benchmarks, which is considerably smaller than both time and space overheads incurred by alternative implementations.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 2 provides background on aspect systems. Section 3 gives a description of our implementation. Section 4 talks about challenges of writing correct and secure policies and describes our policy verifier. Section 5 describes concrete policies we express using our aspect language. Section 6 talks about how to automatically generate policies using static or runtime analysis. Section 7 discusses our experimental results. Finally, Sections 8 and 9 describe related work and conclude. Appendix A shows type inference rules ConScript uses for validating policies.

2 Overview

This section presents an overview of the use of advice to enforce security and reliability properties in a browser.

2.1 Enforcement of Application Policies

Many Web security policies are being proposed for both browsers and Web applications [8, 16, 18]. Similarly, corresponding enforcement mechanisms at the browser and script levels are also being advocated. These proposals highlight the diverse nature of Web security policies and suggest that the security concerns of a Web application are often orthogonal from those of the browser.

Currently, when determining how to enforce security policies of a Web application by using browser-level or script rewriting and wrapping approaches, there are large trade-offs in granularity, performance, and correctness [17, 30, 36]. We propose to expose browser mechanisms and to make them accessible through an advice system. Doing so lowers performance and code complexity barriers for current cross-cutting security policies (and those that have been too difficult or onerous to implement). Furthermore, enabling applications to deploy their own policies decreases the reliance upon browser upgrades to mitigate security threats.

2.2 Motivating Policy Example in ConScript

We start our description of ConScript advice by showing a motivating example of how it may be used in practice. One feature of the JavaScript language that is often considered undesirable for security is the eval construct. At the same time, because this construct is often used to de-serialize JSON strings, it is still commonly used. A naïve approach to prevent unrestricted use of eval involves redefining eval as follows:

window.eval = function(){/ * ...safe version... * /};

However, references to the native eval functions are difficult to hide fully. This is because window.eval and window.parent.eval, for instance, are both aliases for the same function in the JavaScript interpreter. Are there other access paths specified by Web standards, or, perhaps, provided by some non-standard browser feature for a particular release? Another issue is that some native JavaScript functions eschew redefinition, as the BrowserShield project experience suggests [36].

```
 1. <SCRIPT SRC="" POLICY="
 2.   var substr = String.prototype.substring;
 3.   var parse = JSON.parse;
 4.   around(window.eval,
 5.     function (oldEval, str) {
 6.       var str2 = uCall(str, substr, 1,
 7.                  str.length - 1);
 8.       var res = parse(str2);
 9.       if (res) return res;
10.       else throw "eval only for JSON";
11.    } );">
```

Figure 1: Disallowing arbitrary eval calls.

These factors combined call for browser-based support for such interposition, which can be implemented with the notion of *aspects* [7]. An aspect combines code (*advice*) to execute at specified moments of execution (*pointcut*). We are among the first to consider the use of aspects in an adversarial environment, as discussed in Section 4. Figure 1 shows how we support eval interception and argument checking. There are several things to point out:

1. Advice registration is done through a *reference* such as window.eval on line 4, pointing to the function closure whose execution will be advised. Section 2.4 talks about the role of references in deciding what to advise in ConScript in more detail.
2. The original advised function is passed into the advice function as the first parameter oldEval on line 5.
3. The argument to the original eval is passed as the second parameter str on line 5.
4. Exceptions may be thrown by advice on line 10; here we throw an exception to prevent eval on non-JSON arguments.
5. We leverage existing JavaScript features like using a closure to make a protected reference parse on line 3 that points to the function initially pointed to by json_parse.
6. Instead of a regular call to str.substr, we use a special construct uCall on line 6 to do the same so that this policy type-checks. Section 4 addresses security considerations that arise when writing advice code.

2.3 Aspects: Binding Pointcuts to Advice

Our modification to the JavaScript runtime introduces so-called *around advice* by providing a new built-in function Object.around. The function parameter is invoked directly before (and instead of) any function call specified by the first parameter (a *pointcut* selects potentially multiple *joinpoints* at which to alter execution). The advised function is no longer called: it is up to the policy (advice) designer whether and how to invoke the function and how to resume the program, i.e., forge a result, throw an exception, etc.

2.4 Deep Advice

Unlike class-based object-oriented languages such as Java or C#, JavaScript does not support a class structure. So, a typical pointcut consisting of a fully-resolved class name and a method name simply does not apply to JavaScript. To refer to a function or an object field, one can use an *access path*, a string of identifiers like `window.location.href`. A function referred to by access path `document.getElementById` is just an object allocated on the JavaScript heap and, as such, it can easily be aliased with a statement `var ge = document.getElementById;`

Previously proposed advice systems in JavaScript [33] generally use *wrapping* of a particular access path to mediate access to it, which is a form of *shallow advice*. The issue is that this form of mediation is not complete; other aliases such as `ge` for the function being advised can be used to access the function directly. It is quite difficult to prove that no reference leaks occur.

ConScript advocates the notion of *deep advice*. The idea behind deep advice is best illustrated with an example: as mentioned before, function `document.getElementById` is referred to by at least two access paths: `document.getElementById` and `ge`, as illustrated in Figure 2. Registering advice on one of these access paths will in fact advise the *function itself*, independently of which access paths is used for the call. Deep advice is the default approach in ConScript.

2.5 Boot Sequence and Attack Model

ConScript attempts to limit the allowed behavior of JavaScript code by using application-level policies. We assume that an uncompromised browser properly initializes the JavaScript runtime, which creates built-in objects like `Array` and `Date` as well as objects pertaining to the browser embedding of the JavaScript engine such as `document` or `window`. Clearly, if the browser has been compromised, ConScript-style enforcement may not provide much protection.

Next in this "bootup sequence", advice registration is performed. An appropriate analogy here is that advice is "kernel-level", trusted code. Advice can be registered by the *hosting page*, which may subsequently proceed to load third-party, potentially untrusted JavaScript. However, the subsequent script's execution will be restricted through advice registered by the hosting page.

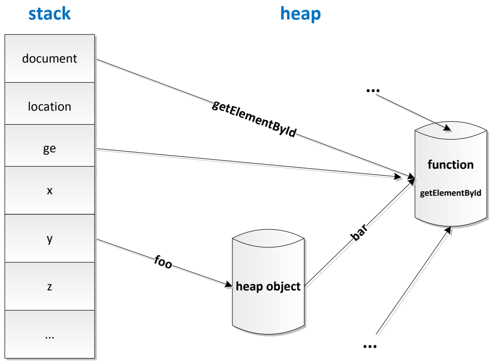


Figure 2: Multiple aliases of function `document.getElementById`.

```
<script src="jQuery.js" policy="
  around($, function ($, expr, ctx) {
    var nodes = $(expr, ctx);
    if (!nodes.length) throw ‘Nothing was selected.’;
    else return nodes;  }); "/>
```

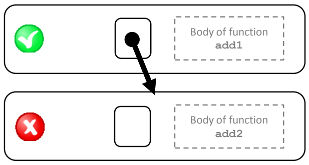
Figure 3: jQuery policy.

Throughout this paper, we assume a powerful attacker who can introduce an arbitrary script into the page. ConScript may be used to limit the scripts that can be injected into the page to a known whitelist, thereby limiting the potential of code injection attacks such as XSS [18]. Alternatively, it may be used to disallow accessing third-party links after cookie access, as explained in Section 5.

An infrequent special case of aspect loading pertains to when we need to load some code *before* registering an aspect. An example of this is a policy for controlling jQuery library behavior from Section 5.4 is shown in Figure 3. The policy is registered around the $ function, which is only available in the global namespeace *after* the jQuery library has been loaded. However, to make sure that `jQuery.js` is not changing the environment in undesirable ways, we need to make sure that `jQuery.js` only *declares* new code and does not *execute* anything as part of being included. This can be achieved through either a static analysis [3, 13] or by observing library loading at runtime. It is our assumption that in the future a ConScript-like system will be integrated with a library loading mechanism that will ensure that the loaded library is not trying to do anything other than registering new code. This is similar to some recent module proposals for JavaScript [31]. An alternative to this solution would be to use a "context" object to store away trusted references before the library is loaded to be later used in policy code. We have prototyped this solution as well, finding it a little more verbose to use in practice.

3 Techniques and Implementation

A design goal for our implementation has been to make minimal changes to the JavaScript engine in Internet Explorer 8. All of our modifications are within the scripting engine; we did not need to modify any other browser subsystems such as the HTML rendering engine. We discuss advising functions and script introduction in Sections 3.1 and 3.3. Section 3.2 focuses on optimizing advice.



```
var add1 = function (x) { return x + 1 }
var add2 = function (_, x) { return x + 2}
adviseFunc(add1, add2)
```

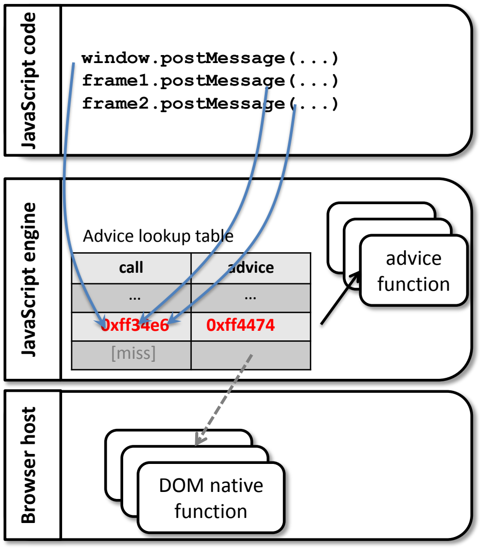
Figure 4: Heap representation of a closure with advice.

3.1 Advising Functions

Our implementation changes the handling of the three types of JavaScript function pointers, as described below.

User-defined functions. Within the Internet Explorer's JavaScript engine, JavaScript functions are represented with heap-allocated and garbage-collected closures. For ConScript, we modified the closure object representation to contain 1) an optional pointer to an advice function pointer and 2) a bit to represent whether it is temporarily disabled. Upon initialization, the advice pointer is `NULL`. Binding advice to a closure is implemented within the runtime by setting the advice pointer on the closure to point to the function that should be run instead (Figure 4). Note that these extra fields are not exposed as JavaScript object fields; they are only visible within the C++ interpreter.

We modified the execution of a user-defined function to first check whether advice was registered and enabled. If so, execution proceeds by running the advice function. This interpositioning is fast in practice because the function to jump into has been resolved at registration time and the stack is already set up for a function call, with the exception of the function being advised being passed as a parameter on the stack.



```
window.postMessage(...)
frame1.postMessage(...)
frame2.postMessage(...)
```

call	advice
...	...
0xff34e6	0xff4474
[miss]	

Figure 5: Foreign function (e.g., DOM) interpositioning.

Native functions. JavaScript supports a standard set of functions, like `eval` and its math libraries, that might be handled more efficiently than more general user-defined functions. As with user-defined functions, there is an explicit object in the interpreter for every such function: interpositioning is analogous to that for user-defined functions.

Foreign functions. A JavaScript engine is typically embedded within a larger hosting application, like a browser, and the host provides functions to the interpreter, which, in turn, exposes them to scripts. For example, Internet Explorer 8 provides COM functions to the JavaScript interpreter for cross-frame communication, which are reachable through the `window` and `document` objects, such as `window.postMessage`.

While such a function is still perceived by the script developer as a JavaScript closure, the hosting environment actually manages the underlying representation. The problem is that Internet Explorer 8's JavaScript interpreter simply represents such functions with a single pointer; there is no object to which we can directly bind advice.

Our solution is to build a *translation table* on demand. As shown in Figure 5, whenever a script binds advice to an external function, the mapping from a function pointer to the corresponding advice function is added to the table. Once a function call is resolved to a foreign function, a check is first made whether advice has been registered: a hit causes the advice to be called, while a miss continues the regular flow. The size of the table is bounded by the number of registered external functions to advice. Compared to alternative implementation strategies, such as using fat pointers, our solution involves minimal instrumentation.

3.2 Blessing and Advice Optimizations

Consider simple "pass-through" advice that attempts to resume the originally invoked function:

```
function add1 (x) { return x + 1; }
function ok (f, x) { return f(x); }
adviseFunc(add1, ok);
var three = add1(2);
```

Upon the initial call to `add1`, because advice is registered for it, `ok` will be called instead. Executing `ok` will call `f`, which is bound to `add1`, leading to infinite recursion.

To address this issue, we provide two functions, `bless` and `curse`, that temporarily manipulate the advice-set bit. A status bit is associated with every closure. Calling `bless` disables advice and calling an advised function checks it, and if the bit is disabled, re-enables the advice bit for the next call, but does not dispatch to the advice function for the current one. The above advice function would be rewritten:

```
function ok(f, x) { bless(); return f(x); }
```

However, in our experimentation with ConScript, we discovered that requiring an explicit call to `bless`, beyond being verbose, introduces an extra script-level function call for the typical case of a policy passing, which incurs a performance penalty. We perform *auto-blessing* by default: we assume advice will dispatch to the raw function and thus disable the advice upon dispatch. For the typical case, the advice code no longer needs to call `bless`.

Automatically flipping the advice bit upon advice invocation introduces a new concern. If the raw function is not called, such as for throwing an exception in response to a policy violation, the advice must be reenabled. We provide the function `curse` to turn the bit back on. For example, to only permit calls to `add1` with numeric parameters, one would write:

```
function onlyNum (f, x) {
  if (typeof x == 'number') return f(x);
  else { curse(); throw 'exn'; } }
around(add1, onlyNum);
```

Auto-blessing also results in a much lower performance overhead than the alternatives. We assess the performance of blessing and auto-blessing in Section 7.

3.3 Advising Script Introduction

Controlling the way new scripts are added to the Web application is paramount to application security. For the specific pointcut of *script introduction*, we modified the engine to support a different form of `around`. Before sending the source of a script to the parser etc., if script advice is registered, it is sent to the advice function:

```
var glbl = this;
around(script, function (src) {
      return (glbl == this) ? "" : src; });
```

In this case, the code about to be introduced is referred to by parameter `src`. As shown above, to determine whether the source is associated with a new `<SCRIPT>` tag or code inlined into an HTML tag (e.g., `<a onclick ="alert('hello')"/>`, advice must check the object to which the `this` is bound.

The string *returned* by the advice function will be passed through the parser instead. This simple advice mechanism could be used to completely change the way scripts are interpreted: for instance, Caja-style rewriting [30] or AdSafe-style subset checking [4] could be applied to the script before being passed to the JavaScript engine.

4 Securing Advice

In this section, we consider attacks against ConScript advice policies. Auditing policies published by other researchers, we found that they are quite tricky to get right. This is true even for policies consisting *of only a few lines of JavaScript* [23, 33]. While the idea of aspects is by no means new [22], in an adversarial environment, aspects are subject to a host of difficult issues.

In our attack model, we distinguish between *kernel* code (code loaded before an untrusted library) and *user* code (untrusted libraries that may execute after the loading sequence). It is our intention to protect against *advice tampering*, i.e. user code that attempts to interfere with the way advise is applied and followed at runtime by tampering with code or data. Our approach is to slightly modify the interpreter to enable isolated reasoning about policies. In Section 4.3, we discuss a custom static analysis to verify policies are safe against common attacks.

4.1 Motivating Example: A Whitelist Policy

Consider the policy in Figure 6a that attempts to use a whitelist to limit which frames may be messaged. Using ConScript's deep `around` advice eliminates concerns about alternate aliases for `postMessage`. However, there are further exploitable attack vectors:

1. `toString` redefinition: The `target` parameter is expected to be a string but this is never checked, so the attacker may foil the whitelist check with a clever use of a custom `toString` method:

```
var count = 0;
frame1.postMessage("1",
  {toString: function () {
    count++;
    return count == 1 ? "http://www.google.com"
      : "evil.com" });
```

2. `Function.prototype` poisoning:
`Function.prototype` may be modified to have method `call` invoke the auto-blessed function:

```
Function.prototype.call =
  function () { window.postMessage("1", "evil.com"); }
frame1.postMessage("1", "http://www.google.com");
```

3. `Object.prototype` poisoning: New entries may be added to `Object.prototype`, including one whitelisting the URL `"evil.com"`:

```
Object.prototype["evil.com"] = true;
frame1.postMessage("1", "evil.com");
```

4. Malicious getters: Combining poisoning attack 3 with a syntactically-invisible malicious getter function, an attacker may even gain access to the whitelist object and edit it:

```
Object.prototype.__defineGetter__("evil", function () {
  delete this["http://www.google.com"]; });
frame1.postMessage("1", "evil");
```

```
var okOrigin = {"http://www.google.com": true};
around(window.postMessage,
  function (post, msg, target) {
    if (!okOrigin[target]) {
      curse(); throw ’err’;
    } else return post.call(this, msg, target); });
```

```
let okOrigin = {"http://www.google.com": true };
around(window.postMessage,
  function (post, msg, target) {
    let t = toPrimitive(target);
    if (!hasProp(okOrigin, t)) {
      curse(); throw ’err’;
    } else return uCall(this, post, msg, t); });
```

Figure 6: Vulnerable **(a)** and secure version **(b)** of the same intended whitelisting policy. Policy (b) passes the type checker.

While aspects eliminate a common source of error in securing APIs — targeting the appropriate functionality — as these examples show, writing correct policy logic is still very tricky. Our solution is to make interpreter-level modifications that enable isolated reasoning and then provide a static analysis for policies. Figure 6b shows a version of the original policy that passes our checker and is also not vulnerable with respect to the attacks listed above.

4.2 New and Removed Features

To enable modular reasoning, we slightly modify JavaScript. Just like the ES5's standard's strict mode [6] we eliminate dynamic constructs `with` and `eval`, as they make static reasoning quite difficult by allowing user code to manipulate seemingly encapsulated policy code. For instance, JavaScript exposes a limited form of stack inspection: if a policy calls an external helper function, that function may use field `caller` to access and *modify* arguments on the stack and call functions on the stack. In ConScript, we disallow `caller` access[1].

We added a new secure calling form `uCall` to avoid prototype poisoning attack 2 on `call`, which we can also use to build further calls. In attack 2, the policy writer wants to invoke `call` on `post`, but invocations of `post.call(...)` are subject to prototype poisoning attacks. Figure 6b demonstrates our new primitive `uCall` that may be used to invoke functions with custom `this` objects (`post` in this example) but without prototype poisoning. Similarly, while we might try to avoid the poisoning in attack 3 of `okOrigin[target]` by writing `okOrigin.hasOwnProperty(target)` to check direct (non-inherited) fields of `okOrigin`, we must avoid using a poisoned `hasOwnProperty`. Our solution is to use `uCall` to encode the safely encapsulated `hasProp` function:

```
var h = {}.hasOwnProperty();
function hasProp (o, fld) {return uCall(o, h, fld);}
```

Finally, to avoid type forgery attacks as in attack 1, we provide function `toPrimitive` to perform the conversion from a potentially poisoned object to a primitive type.

4.3 Statically Validating Policies

In this paper, we propose a static verifier to check for common security holes in policies. The basic of this verifier is a type system in which traditional ML-style types, like arrows for functions and records for objects, are annotated with security labels. We use established techniques to check for our properties [12, 32]. Due to the unfortunate lack of a usable semantics or base type system for JavaScript, providing formal proofs is beyond the scope of this work. Instead, we expand upon the intuition for our approach.

Our type system makes some environmental assumptions. Our restrictions for general JavaScript programs described above, like disabling stack inspection, are required for our type system, and apply both to policy and non-policy code.

It is worth noticing that we do not require non-policy code to pass the type checker: our type system must simply track unknown foreign values. To enable such isolated reasoning by the type system, we perform the preceeding (small) interpreter modifications that limit the encapsulation-breaking abilities of non-policy code.

We found two primary classes of attack against which policies should be verified with our analysis:

1. **Reference isolation:** *Kernel objects should not flow to user code*. E.g., the whitelist in Figure 6 may only be referenced by policy code.
2. **Access path integrity of explicitly invoked functions**[2]: *When a policy invokes a function, that function should be known at time of policy loading*. Otherwise, the call may be subject to prototype poisoning, as with `call` in Figure 6a.

The rest of this section is organized as follows. Our system tracks these properties using labels. After introducing the labels (Section 4.3.1), we summarize the underlying ML-like type system (Section 4.3.2), and examine some representative rules (Section 4.3.3). We briefly describe type inference (Section 4.3.4).

To provide a more formal approach, Figure 8 describes our core language syntax and Figures 10 and 11 presents the corresponding type judgments. An extended language's syntax and type judgments are in Figures 12 and 13, respectively.

[1] This feature is deprecated in the upcoming JavaScript language standard [6]. In particular, Section 15.3.5.4 notes: "If `P` is `caller` and `v` is a strict mode `Function` object, throw a `TypeError` exception." Similar `eval` restrictions needed for encapsulation are given in Section 10.4.2.1.

[2] A stronger property may be to disallow any user function invocation except for the advised function, but this would rule out setters and getters on user objects, which we found to be too draconian.

Label	Policy-only	Invocable
(**u**)ser object		
(**k**)ernel env. function		✓
protected (**o**)bject	✓	✓

Figure 7: Label properties.

4.3.1 Security Labels

All terms are labeled with one of three levels of privilege levels, u, k, and o. The privilege level determines whether either of the following two capabilities is valid for a particular label, as summarized in Figure 7:

- **Policy-only.** The policy-only property enables reference isolation by signaling which values user code cannot directly reference (and, implicitly, stating user code might have access to any other). For example, an object representing a whitelist defined in policy code should not leak out (Figure 6) and thus should be policy-only. In CONSCRIPT, the opposite of being policy-only is being a potential sink for capability leaks. For example, if an object is not policy-only, it might be accessible to user code, as would any of its fields. These fields act like an escape sink towards user code; they should not be assigned a policy-only value like a whitelist. Label o values are policy-only (Figure 7). Only special CONSCRIPT primitives or closures and object literals defined in a policy are labeled o.
- **Invocable.** The invocable property marks the access path integrity of functions that are invoked. For example, function `window.postMessage` in Figure 6 is accessed at policy definition time and thus should be marked as invokable without concern for hijacking. Policy-only terms are labeled o are never leaked and thus not hijacked; they may always be invoked. Environment values accessed in the top-level (policy definition time, akin to boot loading), such as `window.postMessage` above, are labeled k and are invocable. The distinction of top-level vs. function bodies (potentially policy execution time) is somewhat unusual. As the position is syntactically defined, we can define some CONSCRIPT type rules to act differently depending on a term's position. For example, rule (u stat get)'s return type label is shown as k when in the top-level but is u otherwise.

```
T ::= (<script [src= URL ] [policy= S] />)+
S ::= S ; S
    | let ID = E; S
    | E
E ::= P
    | { [" STR " : E [, " STR " : E ]* ] }
    | E . E
    | ID = E
    | E . ID = E
    | [new] E ( A )
    | function ( A ) { S }
A ::= [E (, E)*]
```

Figure 8: Core language syntax.

Base types:	$\star$
Type constructors:	$\texttt{ref}, (), \rightarrow$
Labels:	$\texttt{u}, \texttt{k}, \texttt{o}$
Flows-to:	$L_1 \rhd L_2 \overset{def}{\equiv} (L_1 = \texttt{k} \wedge L_2 = \texttt{u}) \vee (L_1 = L_2)$

Figure 9: Type language and helpers.

Our properties form a partial order of labels. If we kept the two above properties distinct, we could envision a cross-product *unleakable* × *invocable* of labels, though we found only the three above combinations to be relevant. The combination of unleakable and not invocable did not occur in our policies. Furthermore, our labels are ordered by the lattice:

$$(\bot < \texttt{k} < \texttt{u} < \top) \times (\bot < \texttt{o} < \top)$$

The partial ordering of policy-only terms from non-policy-only ones emerges when preserving reference isolation: o terms cannot be substituted for u or k terms in assignment expression nor vice versa in function calls. The subtyping relationship between invocable k and non-invocable u non-policy values is that k terms may be substituted for u terms as strictly less interactions are performed with u terms. We capture the substitutions with flow relation $L_1 \rhd L_2$ (Figure 9), read as L_1 may flow to (or substitute for) L_2, such as $\texttt{k} \rhd \texttt{u}$ and $\texttt{o} \rhd \texttt{o}$.

4.3.2 An ML-like Core Language

Our core language uses an ML-like subset of JavaScript. The base term for every type (e.g., T_1 in $T_1^{L_1}$) is the base type $\star$ or an ML-like type constructor: $(\ldots; fld_n : T_n^{L_n})$ for a record type, $\ldots \times T_n^{L_n} \rightarrow T_o^{L_o}$ for a function type, and T^L `ref` for a reference type (Figure 9). Note that if T_1 is a type constructor, its component types will be labeled. We use the above type constructors to provide structure — otherwise, labels would be too imprecise.

Primitive types are uninteresting in terms of our security properties: we describe them with our single base type $\star$. CONSCRIPT's use of a JavaScript-like language induces several type rules not seen in simplified kernel languages.

- **An open environment.** We assume an open environment of kernel APIs. E.g., rule (global var) imports a variable as a global variable if the name has not been shadowed. Such external values have base type $\star$. We use the ML-like type system to keep track of them: combined with our reference isolation property, we inductively know such foreign calls will return foreign or primitive values (and thus are also base type $\star$).
- **Implicit reference types.** As typical [12], we desugar variable introductions as first-class reference [12] types, as seen in rules (k abstr) and (k obj lit).
- **Primitive functions.** We bootstrap some of our primitives as initial functions in the environment in rule (script

Value construction:

$$\frac{}{\Gamma \vdash i : \star^{\mathtt{k}} \ \mathtt{where}\ i \in \mathbb{R} \cup \mathbb{STRING} \cup \{\mathtt{null}, \mathtt{undefined}\}}\ \text{(prim)}$$

$$\frac{\Gamma \vdash v_n : T_n^{L_n} \quad i \in \{\ldots,\ n\}}{\Gamma \vdash \{\ldots, f_n : v_n\} : (\ldots;\ f_n : {T_n}^{L_n}\ \mathtt{ref}^{L_n})^{\mathtt{o}}}\ \text{(k obj lit)}$$

$$\frac{\begin{array}{c}\Gamma[\ldots,\ a_n \mapsto {T_n}^{L_n}\ \mathtt{ref}^{L_n}][\mathtt{this} \mapsto ()^{\mathtt{o}}\ \mathtt{ref}^{\mathtt{o}}] \\ [\mathtt{arguments} \mapsto ()^{\mathtt{o}}\ \mathtt{ref}^{\mathtt{o}}] \vdash s : T_0^{L_0}\end{array}}{\Gamma \vdash \mathtt{function}\,(\ldots,\ a_n : T_n^{L_n}) : T_0^{L_0}\{s\} : \ldots \times T_n^{L_n} \rightarrow^{\mathtt{o}} T_0^{L_0}}\ \text{(k abstr)}$$

$$\frac{\mathtt{hasProp} \notin \Gamma \quad \Gamma \vdash o : (r)^{\mathtt{o}} \quad \Gamma \vdash i : T^L \quad L \rhd \mathtt{u}}{\Gamma \vdash \mathtt{hasProp}(o, i) : \star^{\mathtt{k}}}\ \text{(k hasProp)}$$

Top-level vs. inner-level value introduction:

$$\frac{x \notin \Gamma \cup \{\mathtt{around},\ \mathtt{hasProp},\ \mathtt{uCall}\}}{\Gamma \vdash x : \star^{\mathtt{k\ or\ u}}\ \mathtt{ref}^{\mathtt{k\ or\ u}}}\ \text{(global var: top-level or inner-level)}$$

$$\frac{\Gamma \vdash o : \star^{L_1} \quad L_1 \rhd \mathtt{u}}{\Gamma \vdash o.f : \star^{\mathtt{k\ or\ u}}}\ \text{(u stat get: top-level or inner-level)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash f : \star^{L_f} & L_f = \mathtt{k} \\ \Gamma \vdash a_i : T_i^{L_i} \quad L_i \rhd \mathtt{u} & i \in \{\ldots,\ n\}\end{array}}{\Gamma \vdash [\mathtt{new}]\ f(\ldots,\ a_n) : \star^{\mathtt{k\ or\ u}}}\ \text{(u f app: top-level or inner-level)}$$

Figure 10: Core language judgments (part 1).

env). Other primitive functions are inexpressible as normal arrow types, like `around` calls, so we utilize special judgments such as rule (around) and make them second-class, enforced by not allowing their aliasing by rule (global var).

- **Correct embedding.** CONSCRIPT is intended for embedding, meaning a JavaScript interpreter, with our proscribed instrumentation, may run CONSCRIPT code alongside typical JavaScript code. We statically prevent some undesired JavaScript features from subverting policy code while allowing non-policy code to still use them. Of note, in rule (k abstr), JavaScript's special identifiers of `this` and `arguments` are typed as empty records, limiting interactions with them, and labeled with o, preventing their leakage.

For clarity of presentation, the type system we describe deviates from CONSCRIPT in our typing of statements in the standard way [12]. To be discussed, our policies are written with a more concise (and optional) annotation language that is described in Section 5. Otherwise, the two are consistent.

$$\frac{\begin{array}{c}\Gamma[\mathtt{bless} \mapsto \star^{\mathtt{k}} \rightarrow^{\mathtt{o}} \star^{\mathtt{k}}][\mathtt{toPrimitive} \mapsto \star^{\mathtt{u}} \rightarrow^{\mathtt{o}} \star^{\mathtt{k}}] \\ [\mathtt{curse} \mapsto \star^{\mathtt{k}} \rightarrow^{\mathtt{o}} \star^{\mathtt{k}}][\mathtt{autobless} \mapsto \star^{\mathtt{k}} \rightarrow^{\mathtt{o}} \star^{\mathtt{k}}] \\ \vdash s_i : T_i^{L_i} \qquad\qquad i \in \{\ldots,\ n\}\end{array}}{\Gamma \vdash \ldots \mathtt{<script}\ [\mathtt{src} = \text{“}url\text{”}]\ [\mathtt{policy} = \text{“}s_n\text{”}]\mathtt{>} : \star^{\mathtt{u}}}\ \text{(script env)}$$

$$\frac{\Gamma \vdash o : (f : T^L\ \mathtt{ref}^L;\ r)^{\mathtt{o}}}{\Gamma \vdash o.f : T^L}\ \text{(k stat get)}$$

$$\frac{\Gamma \vdash o : (f : T^L;\ r)^{\mathtt{o}} \quad \Gamma \vdash v : T^L}{\Gamma \vdash o.f = v : T^L}\ \text{(k stat set)}$$

$$\frac{\Gamma \vdash o : \star^{L_1} \quad \Gamma \vdash v : \star^{L_2} \quad L_1, L_2 \rhd \mathtt{u}}{\Gamma \vdash o.f = v : T_2^{\mathrm{L}_2}}\ \text{(u stat set)}$$

$$\frac{\Gamma \vdash o : T^{L_1}\ \mathtt{ref}^{L_1} \quad \Gamma \vdash v : T^{L_2} \quad L_2 \rhd L_1}{\Gamma \vdash o = v : T^{L_2}}\ \text{(asgn)}$$

$$\frac{\begin{array}{cc}\Gamma \vdash a_i : T_i^{A_i} & i \in \{\ldots,\ n\} \\ \Gamma \vdash f : \ldots \times T_n^{L_n} \rightarrow^{\mathtt{o}} T_o^{L_o} & A_i \rhd L_i \ \ i \in \{\ldots,\ n\}\end{array}}{\Gamma \vdash [\mathtt{new}]\ f(\ldots,\ a_n) : T_o^{L_o}}\ \text{(k f app)}$$

$$\frac{\begin{array}{c}\Gamma \vdash p : \star^{L_p} \qquad L_p \rhd L_1 \qquad L_r \rhd L_p \\ \Gamma[\mathtt{this} \mapsto \star^{\mathtt{u}}\ \mathtt{ref}^{\mathtt{u}}][\mathtt{arguments} \mapsto \star^{\mathtt{u}}\ \mathtt{ref}^{\mathtt{u}}] \\ [\ldots,\ a_i \mapsto T_n^{L_n}\ \mathtt{ref}^{L_n}] \vdash s : T_r^{L_r}\end{array}}{\Gamma \vdash \mathtt{around}(p,\ \mathtt{function}\,(\ldots,\ a_n : T_n^{L_n})\ : T_r^{L_r}\{s\}) : \star^{\mathtt{u}}}\ \text{(around)}$$

$$\frac{\Gamma \vdash s_1 : T_1^{L_1} \quad \Gamma \vdash s_2 : T_2^{L_2}}{\Gamma \vdash s_1; s_2 : T_2^{L_2}}\ \text{(seq)}$$

$$\frac{\Gamma \vdash e : T_1^{L_1} \quad \Gamma[x \mapsto T_1^{L_1}\ \mathtt{ref}^{L_1}] \vdash s : T_2^{L_2}}{\Gamma \vdash \mathtt{let}\ x = e; s : T_2^{L_2}}\ \text{(let)}$$

Figure 11: Core language judgments (part 2, value manipulation).

```
T ::= ...
S ::= ...
    | if ( E ) { S }
    | if ( E ) { S } else { S }
    | while ( E ) { S }
    | try { S } catch ( ID ) { S }
    | throw E
E ::= ...
    | E [ E ]
    | E [ E ] = E
    | E B E
    | U E
    | [new] E . ID  ( A )
    | [new] E [ E ] ( A )
A ::= ...
B ::= + | - | * | / | && | ||
U ::= ! | +
```

Figure 12: Extended language syntax.

4.3.3 Intuition and Sample Inference Rules

This section examines several examples of how labels are used in our type system.

Calling trusted foreign functions: The first rule we discuss below would be used to check the invocation of $\texttt{uCall}(\texttt{this}, \texttt{post}, \texttt{msg}, \texttt{t})$ in Figure 6b. `uCall` may invoke non-policy functions, but the rule must ensure that the non-policy function has not been hijacked and that it is not leaked a reference to policy objects:

$$\dfrac{\begin{array}{c} \texttt{uCall} \notin \Gamma \\ \Gamma \vdash o : T_o^{L_o} \qquad\qquad L_o \rhd \texttt{u} \\ \Gamma \vdash f : \star^{L_f} \qquad\qquad L_f = \texttt{k} \\ \Gamma \vdash a_n : T_n^{L_n} \quad L_n \rhd \texttt{u} \quad i \in \{\ldots,\ n\} \end{array}}{\Gamma \vdash \texttt{uCall}(o, f\ [, \ldots,\ a_n]) : \star^{(\texttt{k or u})}}$$

First, we cannot use an ordinary arrow type for `uCall`: the first antecedant checks that it is truly the environment's `uCall` and a rule not shown here checks that it is never used in a first-class way. The fourth antecedant checks that `post`'s base type is $\star$: `uCall` is not intended for policy functions. The fifth antecedant requires `post` to have label `k`, meaning it could not have been hijacked and can thus be invoked. Using flow relation $\rhd$, the third and seventh antecedants check that `this`, `msg`, and `t` values have low enough privilege to be allowed to flow to user code (label `u`). As f is not a policy function and external code is not given policy values, we know the result of `uCall` is either a primitive value or an external non-primitive value, denoted by return type $\star$.

Our system distinguishes top-level code, executed when advice is registered and thus has access to kernel APIs in the global environment, from code in function bodies, which might run in response to user code that has poisoned the global environment. If `uCall` is used in the top-level, we label the return value as `k`, meaning it can be used to load kernel APIs. However, if it is not in the top-level, we cannot

$$\dfrac{\begin{array}{c} \Gamma \vdash o : T_1^{L_1} \quad \Gamma \vdash i : T_2^{L_2} \quad \Gamma \vdash v : T_3^{L_3} \\ L_1, L_2, L_3 \rhd \texttt{u} \end{array}}{\Gamma \vdash o[i] = v : T_3^{\mathrm{L}_3}} \quad \text{(dyn set)}$$

(dyn get: top-level or inner-level)

$$\dfrac{\Gamma \vdash o : T_1^{L_1} \quad \Gamma \vdash i : T_2^{L_2} \quad L_1, L_2 \rhd \texttt{u}}{\Gamma \vdash o[i] : \star^{\texttt{k or u}}}$$

$$\dfrac{\begin{array}{c} \Gamma \vdash o : \star^{L_o} \qquad\qquad L_o \rhd \texttt{u} \\ \Gamma \vdash a_i : T_i^{L_i} \quad L_i \rhd \texttt{u} \quad i \in \{\ldots,\ n\} \end{array}}{\Gamma \vdash [\texttt{new}]\ o.f(\ldots,\ a_n) : \star^{\texttt{k}}} \quad \text{(u m app: top-level)}$$

$$\dfrac{\begin{array}{c} \Gamma \vdash o : \star^{L_o} \quad \Gamma \vdash i : \star^{L_i} \quad L_o, L_i \rhd \texttt{u} \\ \Gamma \vdash a_i : T_i^{L_i} \quad L_i \rhd \texttt{u} \quad i \in \{\ldots,\ n\} \end{array}}{\Gamma \vdash [\texttt{new}]\ o[i](\ldots,\ a) : \star^{\texttt{k}}} \quad \text{(u d m app: top-level)}$$

(uCall: top-level or inner-level)

$$\dfrac{\begin{array}{c} \texttt{uCall} \notin \Gamma \quad \Gamma \vdash o : T_o^{L_o} \quad L_o \rhd \texttt{u} \\ \Gamma \vdash f : \star^{L_f} \qquad\qquad L_f = \texttt{k} \\ \Gamma \vdash a_n : T_n^{L_n} \quad L_n \rhd \texttt{u} \quad i \in \{\ldots,\ n\} \end{array}}{\Gamma \vdash \texttt{uCall}(o, f\ [, \ldots,\ a_n]) : \star^{(\texttt{k or u})}}$$

$$\dfrac{\begin{array}{c} \Gamma \vdash a_i : T_i^{A_i} \quad A_i \rhd L_i \quad i \in \{\ldots,\ n\} \\ \Gamma \vdash o : (m : \ldots \times T_n^{L_n} \rightarrow^{L_f} T_r^{L_r}\ \texttt{ref}^{L_r}; r)^{L_o} \end{array}}{\Gamma \vdash o.m(\ldots,\ a_n) : T_r^{L_r}} \quad \text{(k m app)}$$

$$\dfrac{\Gamma \vdash t : T^L \quad \varnothing \in \{!, +\}}{\Gamma \vdash\ \varnothing\, t : \star^{\texttt{k}}} \quad \text{(unop)}$$

(binop)

$$\dfrac{\Gamma \vdash t_1 : T_1^{L_1} \quad \Gamma \vdash t_2 : T_2^{L_2} \quad \varnothing\ \in \{\&\&, ||, ==, -, *, /\}}{\Gamma \vdash t_1 \varnothing\, t_2 : \star^{\texttt{k}}}$$

$$\dfrac{\Gamma \vdash t_1 : T_1^{L_1} \quad \Gamma \vdash t_2 : T_2^{L_2} \quad L_1, L_2 \rhd \texttt{u}}{\Gamma \vdash t_1\ +\ t_2 : \star^{\texttt{k}}} \quad \text{(add/concat)}$$

$$\dfrac{\Gamma \vdash s_1 : T^L \quad \Gamma[e \mapsto \star^{\texttt{u}}\ \texttt{ref}^{\texttt{u}}] \vdash s_2 : T^L}{\Gamma \vdash \texttt{try}\ \{s_1\}\ \texttt{catch}\ (e)\ \{s_2\} : T} \quad \text{(try)}$$

$$\dfrac{\Gamma \vdash e : T^L \quad L \rhd \texttt{u}}{\Gamma \vdash \texttt{throw}\ e} \quad \text{(throw)}$$

$$\dfrac{\Gamma \vdash e : T_1^{L_1} \quad \Gamma \vdash s_1 : T_2^{L_2} \quad [\Gamma \vdash s_2 : T_2^{L_2}]}{\Gamma \vdash \texttt{if}\ (e)\ \{s_1\}\ [\texttt{else}\ \{s_2\}] : T_2^{L_2}} \quad \text{(if)}$$

$$\dfrac{\Gamma \vdash e : T_1^{L_1} \quad \Gamma \vdash s : T_2^{L_2}}{\Gamma \vdash \texttt{while}\ (e)\ \{s\} : T_2^{L_2}} \quad \text{(while)}$$

Figure 13: Extended language judgments

1. $\llbracket \texttt{K} \rrbracket = \star^{\texttt{k}}$
2. $\llbracket \texttt{U} \rrbracket = \star^{\texttt{u}}$
3. $\llbracket \{ \ldots , \; fld_n : T_n \} \rrbracket = (\ldots ; \; fld_n : \llbracket T_n \rrbracket \; \texttt{ref}^{label(\llbracket T_n \rrbracket)})^{\circ}$
4. $\llbracket \ldots \; \texttt{x} \; T_n \rightarrow T_o \rrbracket = \ldots \times \llbracket T_n \rrbracket \rightarrow^{\circ} \llbracket T_o \rrbracket$

where $label(T^L) = L$.

Figure 14: Interpretation $\llbracket \cdot \rrbracket$ of policy annotations as type annotations.

trust that its result has not been hijacked nor that its fields are impervious to hijacking: we label it as u.

Dynamic field sets: The second rule applies to the syntactic form $o[i] = v$. Intuitively, if i is not a direct field of o, o's prototype chain will be checked for i, which might resolve to a field on Object.prototype. In the case of poisoning with a setter, similar to attack 4 (Section 4.1), o may leak. As we do not statically know the value of i, such a call is only allowed if o's type has a privilege label that states it is acceptable to leak it to user code ($L_1 \rhd \texttt{u}$).

$$\frac{\Gamma \vdash o : T_1^{L_1} \qquad \Gamma \vdash i : T_2^{L_2} \qquad \Gamma \vdash v : T_3^{L_3} \qquad L_1, L_2, L_3 \rhd \texttt{u}}{\Gamma \vdash o[i] = v : T_3^{\mathrm{L}_3}}$$

If term i is an object, it will be dispatched to the toString method. toString might be poisoned to leak object i, so we must check that i's label allows it to flow to user values ($L_2 \rhd \texttt{u}$). Due to these same attacks, we must also ensure that it is acceptable to leak v to user code: $L_3 \rhd \texttt{u}$. The other antecedents simply check that the terms are well-typed.

Static field sets with records. The third rule applies to form $o.f = v$ where o was defined within the policy code, such as if we wanted to modify the whitelist. In this case, type inference reveals that o is a record type:

$$\frac{\Gamma \vdash o : (f : T^L ; \; r)^{\circ} \qquad \Gamma \vdash v : T^L}{\Gamma \vdash o.f = v : T^L}$$

Unlike with dynamic field sets, we can check that the desired field actually exists in the record, in which case there is no prototype poisoning attack to leak o. If the record's field and assignment's right-hand side labels and types match, the assigned value will not be leaked either.

4.3.4 Label Inference

Label inference follows base type inference. If labels are ignored, the base types may be inferred using ML-style unification algorithm. Concrete label o is introduced for type constructors and labels k and u for top-level and inner-level global variables, respectively. The remaining labels are variables to be inferred as Foster et al. [12] describes for general type qualifier labels.

5 Policies

In this section we present a variety of fine-grained policies we expressed with CONSCRIPT. To collect these policies, we studied bugs and anti-patterns in both "raw" JavaScript as well as popular JavaScript libraries such as jQuery. We also investigated and rewrote some of the policies published in the literature [23, 33] in CONSCRIPT. We discovered that in some cases, a few lines of policy code can replace a new specialized HTML tag. This demonstrates that CONSCRIPT provides a general enforcement mechanism for a wide range of application-level security policies. Crucially, our type system helped us avoid many errors found in previously proposed policies and even a few of our own.

To help demonstrate the value our type system described in Section 4.3 provides, we manually annotate variable declarations and function parameters with security labels. Due to some invariants of CONSCRIPT's use of labeled types, our policy annotations (Section 5) use a simpler language than that of our type annotations (Figure 9). Figure 14 defines a syntax-directed translation from policy annotations to more verbose but canonical labeled type annotations. The intuition is that terms with label u or k have base type $\star$ (rules 1 and 2, exercised in policy 2) and that terms whose base types are constructors will have label o (rules 3 and 4, exercised in policies 2 and 4, respectively). As these redundancies may be reconstructed in a simple, direct manner, our type annotation language elides them.

In the remainder of this section, the policies are grouped as controls on vectors for dynamic code introduction (Section 5.1), communication restrictions (Section 5.2), document object policies (Section 5.3), and API and library reliability guidelines (Section 5.4).

5.1 Script Introduction Policies

We start with an important class of policies are used for controlled dynamic introduction of code. Recall that CONSCRIPT is designed to protect the hosting page from either malicious or poorly-written third-party code and libraries. As such, the hosting page may choose to enforce properties of the introduced code and this section explores some of these possibilities.

As an extreme, one might write a CONSCRIPT policy to parse dynamically injected code and perform static analysis on it, rejecting everything that does not match a static analysis policy [3, 13]. Static analysis techniques currently struggle with mechanisms for intercepting dynamically injected code [3, 14], which CONSCRIPT more cleanly exposes, as show below. Recall that in the case of script interception advice, the policy is designed to return the code that the JavaScript interpreter will proceed to run instead of the code being introduced.

1. No dynamic scripts

The simplest policy is to simply disallow any code from being introduced after a certain point, such as after the main library loads. We encode disabling scripts by returning the empty string back to the JavaScript interpreter.

```
<script src="main.js" policy="
  around(script, function () { return ‘’; }); "/>
```

2. No string arguments to `setInterval`, `setTimeout`

The functions `setInterval` and `setTimeout` run callbacks in response to the passing of time. A closure is typically passed in for the callback parameter. Surprisingly, and even commonly proscribed in introductory tutorials, string arguments to be evaluated may also be accepted. This attack vector may be easily dealt with.

```
let onlyFnc : U x U -> K =
  function (setWhen : K, fn : U, time : U) {
    if ((typeof fn) != "function") {
      curse();
      throw "The time API requires functions as inputs.";
    } else return setWhen(fn, time); };
around(setInterval, onlyFnc);
around(setTimeout, onlyFnc);
```

3. No inline scripts

Previous code injection attacks such as the Samy worm would often try to attach malicious script to DOM elements. The policy below aims to prevent inline scripts: if a script's context is not the global object, it is an inline script, so the empty string is returned to the interpreter.

```
let glbl : K = this;
around(script, function (src) {
    return glbl == this ? src : ""; });
```

4. Script tag whitelist

We can ensure that statically loaded script tags have a source listed in whitelist `w`. When the advice function is invoked, the script tag has been created, but the script has not yet been executed. Therefore, for statically loaded scripts, checking the `src` attribute of the last one in the document tree suffices. A more direct interface would be to modify our system to also pass in the node or script context as a parameter to the script advice function [8]. Issues of correctness pertaining to whitelist use are covered in Section 4.2.

```
let glbl : K = this;
let getScripts : K = document.getElementsByTagName;
let doc : K = document;
let w : {"good.js": K} = {"good.js": true};
around(script, function (load : K, src : U) {
    if (this == glbl) {
      let scripts : U = uCall(doc, getScripts, "script");
      return hasProp(w, scripts[scripts.length - 1].src) ?
        load(src) : "";
    } });
```

5. NOINLINESCRIPT tag

BEEP [18] advocates the introduction of a <`noscript`> tag. Fortunately, CONSCRIPT is general enough to implement this tag in the form of a policy. As a finer-grained version, the following policy prevents inline scripts from being loaded as descendants of a <`noinlinescript`> tag:

```
let glbl : K = this;
let getScripts : K = document.getElementsByTagName;
let doc : K = document;
around(script, function (load : K, src : U) {
    if (this == glbl) return src;
    else {
      let n : U = this;
      while (n)
        if (n.tagName == "NOINLINESCRIPT") return "";
        else n = n.parentNode;
      return src; } });
```

The significance of this example is that we can securely previously proposed HTML tags using our primitives, separating the slow process of defining new standards and upgrading browsers from securing applications.

5.2 Communication Restrictions

Our trust model, reflecting modern application design, expands an application's trust boundary to include the client. By trusting client-side computations, Web applications are now sensitive to how a client communicates with untrusted principals. Developers should now, for example, more carefully restrict how messages are passed to frames belonging to untrusted origins or what RPC calls are made to third-party servers. Browser policies are coarse and may be ignored; we show how applications may enforce fine-grained policies on communication.

6. Restrict `XMLHttpRequest` to secure connections

`XMLHttpRequest` enables communication with an application's server without reloading a page. An instance of the `XMLHttpRequest` object provides the method `open(mode, url, sync, username, password)`, where the last two parameters are optional. A program that specifies a username and password has heightened security concerns. The following policy ensures, if a username and password is supplied, that the connection is over HTTPS.

```
let substr : K = String.prototype.substr;
around((new XMLHttpRequest()).open,
  function (o : K, m : U, u : U, a : U, nm : U, pw: U)
  {
    let name : K = toPrimitive(nm);
    let password : K = toPrimitive(pw);
    let url : K = toPrimitive(u);
    if ((name || password)
       && uCall(url, substr, 0, 8) != "https://") {
      curse(); throw "Use HTTPS for secure a XHR.";
    } else
      return uCall(this, o, m, url, a, name, password);
    });
```

7. HTTP-only cookies

Servers often store state on the client to avoid costs associated with maintaining session state between calls to the server. Cookies may therefore contain valuable state information that should only be read and written by the server. We therefore might want to disable JavaScript access to cookies.

```
let httpOnly : K -> K = function (_ : K) {
    curse(); throw "HTTP-only cookies"; };
around(getField(document, "cookie"), httpOnly);
around(setField(document, "cookie"), httpOnly);
```

8. Whitelist cross-frame messages

The `postMessage` function may transmit primitive values between frames of differing origins. While a developer may specify the intended origin of the receiving frame during a particular call, which prevents man-in-the-middle attacks [1], the developer is not obligated to. The following requires such a specification and, further, limits communication to a whitelist of URIs.

```
let okOrigins : {"http://www.google.com": K}
  = {"http://www.google.com":  true};
around(window.postMessage,
  function (p : K, msg : U, target : U) {
    let t : K = toPrimitive(target);
    if (!hasProp(okOrigins, t)) {
      curse(); throw ’err’;
    } else return p(msg, t); });
```

9. Whitelist cross-domain requests

The push to have more application functionality run in the browser has led to new primitives like `XDomainRequest` for communicating with foreign servers without requiring a server-side proxy (which might have performed its own access checks). Similar to the `postMessage` example, we introduce a check against a whitelist of URIs before allowing cross-domain server requests.

```
let w : {"http://www.google.com": K}
  = {"http://www.google.com":  true};
around((new XDomainRequest()).open),
  function (x : K, a1 : U, url : U) {
    let u : K = toPrimitive(url);
    if (!hasProp(w, u)) {
      curse(); throw ’err’;
    } else return x(a1, u); });
```

A subtlety of the `XMLHttpRequest` and `XDomainRequest` examples are that we advise the function attached as method `open` on request object instances. Each request object calls the same function, one for `XMLHttpRequest` calls and a different one for `XDomainRequest` calls. Using the `rqst.open`(...) form just passes in `rqst` to the advised function as the `this` object; despite advising a function reached through one instance of a request object, we are indeed advising the function shared by all of them. This is analogous to advising `eval` by using just one alias.

5.3 DOM Interactions

DOM interaction are a common source of security flaws. This section shows how ConScript policies can help reign in some of these issues.

10. No foreign links after a cookie access

The following policy, proposed by Kikuchi et al. [23], is intended to prevent links from being used for cookie access. The first advice function represents eliminating side-channels. The second advice function, after being triggered, enables a stricter policy mode. The third advice function attaches a policy to `src` attributes of dynamically generated nodes: it avoids `toString` rewriting attacks and, whenever the strict policy is enabled, whitelists target domains.

```
around(document.setAttribute, function () {
    curse(); throw ’err’; });
let ok : K = true;
around(getFld("cookie", document), function (g : K) {
    ok = false;
    return g(); });
let slice : K = Array.prototype.slice;
around(document.createElement, function (c : K, t : U) {
    let elt : U = uCall(document, c, t);
    if (elt.nodeName == "A")
      around(setFld("href", elt),
        function (setter : K, v : U) {
          let str : K = toPrimitive(v);
          if (ok ||
              uCall(str, slice, 12) == "http://g.com/"))
            setter(str);
          else {
            curse(); throw ’err’; } });
    return elt; });
```

We show how this policy can be implemented in ConScript with only a few lines of policy code.

11. Limit popup window construction

Below we show the implementation of another policy proposed by Kikuchi et al. [23], we can limit the number of attempts to open a popup window by counting the number of invocations. Further, we can restrict the dimensions of the popup window.

```
let split : K = String.prototype.split;
let toLower : K = String.prototype.toLowerCase();
let match : K = String.prototype.match;
let toInt : K = parseInt;
let count : K = 0;
around(window.open,
  function (w : K, url : U, name : U, features : U) {
    if (count++ > 2) {
      curse(); throw ’err’;
    } else if (features) {
      let f = toPrimitive(features);
      let a = uCall(f, split, ",");
      let i = 0;
      while (i < a.length) {
        var o = uCall(a[i], split, "=");
        var prop = uCall(o[0], toLower);
        if (uCall(prop, match, "width|height"))
          if (toInt(o[1]) < 100) {
            curse(); throw 'err'; }
        i++; }
      return w(url, name, f); } });
```

To further prevent click-jacking, a similar policy might also be used to restrict where the window may be moved.

12. Disable dynamic `IFRAME` creation

Phung et al. [33] introduce a policy to prevent the construction of IFRAME elements using `createElement`. Note that unlike the original policy, ours is safe from attacks like running `delete` on the attribute or accessing the `createElement` function from other aliases.

```
around(document.createElement,
  function (c : K, tag : U) {
    let elt : U = uCall(document, c, tag);
    if (elt.nodeName == "IFRAME") throw 'err';
    else return elt; });
```

13. Whitelist URL redirections

Phung et al. [33] advocate checking programmatic URL redirections against a whitelist. Note in the following policy the common theme of not leaking the whitelist:

```
let whitelist : {"http://microsoft.com": K}
  = {"http://microsoft.com": true};
around(setFld(document, "location"),
  function (setter : K, url : U) {
    let to : K = toPrimitive(url);
    if (hasProp(whitelist, to)) setter(to);
    else { curse(); throw 'err'; } });
```

14. Prevent resource abuse

A common policy is for preventing abuse of resources like modal dialogs. These may be disabled simply:

```
let err : K -> K = function () {
                  curse(); throw 'err'; });
around(prompt, err);
around(alert, err);
```

5.4 API and Reliability Guidelines

A key use case for advice is to introduce additional constraints such as pre- and post-conditions on the use of important APIs. In the following examples, also note how the structured nature of advice allows us to install upgrades to third-party libraries like jQuery without needing to manually reinstrument or otherwise specially handle the new versions in our policies.

15. Simple and fast jQuery selectors

$ is a core operator in the popular jQuery library that, given a selector expression, returns the matching document elements. For code style or, more commonly, performance concerns about selectors, a simple pre-condition is to disallow selectors with slow composition operators.

```
<script src="jQuery.js" policy="
  let match : K = String.prototype.match;
  let r : K = /^[a-zA-Z0-9.#:]+(( > | )[a-zA-Z0-9.#:]+)+$/;
  around($, function ($ : K, selStr : U) {
      let s : K = toPrimitive(selStr);
      if (!uCall(s, match, r)) {
        curse();
        throw ‘Compose selectors only with ( > ) or ( ).’;
      } else return $(s); });"/>
```

16. Explicit jQuery selector failure

An anti-pattern by jQuery is to silently fail when no elements are returned by $, allowing a library user to attach behavior to the null-set. An application may choose to add the post-condition that $ return values should not be empty.

```
<script src="jQuery.js" policy="
  around($, function ($ : K, expr : U, ctx : U) {
    let nodes : U = $(expr, ctx);
    if (!nodes.length) throw ‘Nothing was selected.’;
    else return nodes;  }); "/>
```

17. Staged `eval` restrictions

A common implicit invariant in JavaScript applications is that they use `eval` [40] but only in restricted ways. This might more precisely appear as a staged precondition. For example, we might allow the trusted jQuery library to initialize itself using `eval` but, for all subsequent code, we might then restrict usage of `eval` to deserializing JSON objects.

```
<script src="jQuery.js" policy="
  let parse : K = JSON.parse;
  around(eval : K, function (_ : K, evalStrArg : U) {
      curse();
      return parse(evalStrArg); }); }); "/>
```

6 Automatically Generated Policies

Writing policies by hand places the burden on the developer to “get things right”. Fortunately, relatively few modern large-scale Web applications are constructed in isolation, without using a framework or a toolkit of some kind, such as GWT [11], Volta [28], Java J2EE, ASP.NET, etc. These frameworks can bring in policies of their own that extend to applications written on top of them.

Application-specific policies can also be inferred through static analysis or runtime training. In this section we explore these options through two case studies. The first described in Section 6.1 uses a very simple form of static analysis on the server to restrict possible behavior on the client. The second described in Section 6.2 uses runtime training to determine the space of “expected” benign behavior and then rejects behaviors outside of the training set. In both cases, once policies have been generated, CONSCRIPT is used for policy enforcement. We assess the performance of both in Section 7.

6.1 Private Methods in Script#

Script# is a tool that translates C# code into JavaScript [24]. This tool is used in a variety of large-scale commercial projects such as Live Maps to simplify and quicken the development process. As part of its C#-to-JavaScript translation, Script# takes a set of C# classes and translates them into JavaScript. However, these two languages are really quite different. One area of distinction is that C# supports access qualifiers such as `internal`, `private`, `protected`, and `public`, and JavaScript does not. After the translation takes place, a previously `private` method is effectively accessible as a `public` one to any piece of code that is loaded before and after the code that has been translated with Script#. This issue is sometimes referred to as failure of full abstraction [20].

Fortunately, ConScript makes this deficiency simple to rectify by generating a policy as part of the translation process. We traverse the original C# program source offline, identifying `private` methods and, for each, also identity public entry points to the classes (`public` methods) in which the private methods are found. Note that this information is readily available to a C# language compiler.

We automatically generate policies from this list: an `enabled` status bit is allocated for every class, where entry through a public point is modified to enable the corresponding class bit, exit resets it, and access of a private method checks it. The privileged policy bits are encapsulated within the set of policies and the (anonymous) policies are associated with the method objects; private methods may only be called when public ones are on the stack, akin to *cflow* pointcuts in other aspect systems [22]. If we exposed `arguments.caller` to policies, we could match the exact access modifier semantics and would only need to instrument the `private` methods.

6.2 Intrusion Detection of Client-Side Exploits

In practice, many JavaScript applications only exercise a small subset of browser capabilities, but this subset varies between applications. According to the principle of least authority, if an application does not need a capability, it should not have it. Instead of using a preset list, which may be too lax, or manually generating the policy list of acceptable functionality, which may be error-prone, here we demonstrate the potential for automatically restricting browser functionality to a subset. Such a subset can be synthesized through runtime training. This is similar to intrusion detection techniques that train on valid runs observing system calls or their sequences, proceeding to flag all other possibilities as suspicious [9, 14, 38]. An interesting observation is that ConScript may be used to apply logging aspects to a large number of functions to see which ones are used at the time of training. ConScript can also be used to enforce the blacklist at the time of detection.

This general approach may be used to harden many Web sites and applications. Consider the popular Web applications GMail and Google Calendar. As a Web-based program designed to display untrusted HTML email, GMail is a particularly good example for this style of intrusion detection. If a maliciously crafted message "breaks out" of GMail sanitization, which is what happened in the case of the Yamanner worm [37], our aspects will flag attempts to execute previously unseen dangerous method calls. A Google Calendar user might similarly attempt to circumvent sanitization by creating a meeting request with a malicious body and sending it to others.

In our experiments, we blacklisted `XDomainRequest`, `XMLHttpRequest`, `postMessage`, `setTimeout`, `setInterval`, `eval`, `alert`, `prompt` and several other potentially dangerous methods not seen during training. We did not encounter any intrusion detection alarms.

7 Evaluation

This section presents an evaluation of ConScript in the context of Internet Explorer 8. Our primary focus is on the runtime overhead introduced with ConScript instrumentation compared to alternative techniques. Section 7.1 talks about our experimental setup. Section 7.2 evaluates micro-benchmarks and Section 7.3 focuses on applying ConScript advice to large AJAX sites and applications such as MSN, GMail, and Live Desktop; a summary of information for these applications is given in Figure 16. In addition to measuring the performance overhead, we also compare ConScript's *space* overhead to that induced by JavaScript code rewriting systems Caja [30] and WebSandbox [17] and the advice system proposed in Kikuchi et al. [23] in Section 7.4.

7.1 Browser Modifications for ConScript

As explained in Section 3, we modified the JavaScript interpreter in Internet Explorer 8 to support advising functions and dynamic script introduction. In addition, for the integrity of policies, we had to disable features like `arguments.callee`, as suggested by ECMAScript 5 [6], and introduce more secure calling forms for primitive functions like `around`, `uCall`, etc., as discussed in Section 4.2.

Overall, our changes are small and we believe similar augmentations can be made to scripting engines of other browsers. In total, we have added 969 lines to the JavaScript interpreter over 60 different code locations, which constitutes a small fraction of the overall interpreter size. About half of these changes were boilerplate for exposing new functionality as JavaScript functions: once discounted, the average instrumentation point was only 8 lines of code. In contrast, Caja [30], a source rewriting tool, currently has over 181,000 lines of code in its main source directory. All measurements reported in this section were performed on a Dual Core 3GHz

Task	raw	wrapping	blessing	auto-blessing
USER-DEFINED FUNCTIONS				
`function(){}`	1	3.86	1.06	1.02
`function(){return + 1; }`	1	4.04	1.07	1.03
NATIVE FUNCTIONS				
`Math.tan(5)`	1	1.37	2.16	1.27
`eval(”1”)`	1	1.53	1.47	1.36
`eval(”if(true)true; false; ”)`	1	2.93	1.79	1.72
FOREIGN FUNCTIONS				
`getElementsByTagName(”div”)`	1	5.57	1.31	1.20
`createElement(”div”)`	1	4.63	1.2	1.10
Average	1	3.42	1.44	1.24

Figure 15: Runtime overhead of applying aspects to micro-benchmarks.

Pentium 2 machine running Windows Vista.

7.2 Runtime Overhead on Micro-benchmarks

Potentially thwarting our goal of fine-grained policy support is interpositioning cost. Language-level support, in static languages, has been shown to take away much of the instrumentation cost of an aspect system. In particular, the cost of additional function dispatches introduced by a naïve syntactic desugaring of an aspect might be eliminated by approaches like inlining. We find similar benefits for a dynamic language. In this section we study the overhead of 1) mediating a function call with advice, and 2) of lesser concern, the initialization overhead of associating an advice policy with a function to protect.

7.2.1 Overhead of Advice Interpositioning

We consider the cost of running advice that simply proxies calls with no side-effect beyond the seemingly inherent performance cost of an indirected call. This removes the policy cost, leaving only the advice mechanism and the original function. Further refining previous benchmarks of a proposed wrapping system [33], we distinguish between advising user defined functions, native functions provided by the JavaScript interpreter (at low cost), and external native DOM functions exposed to the JavaScript interpreter through a COM interface (at a high cost). A summary of our micro-measurements is shown in Figure 15.

To collect out measurements, for every benchmark, we start a new JavaScript runtime, run 10,000 invocations of the advised function, and normalize over the cost of running 10,000 invocations of the function unadvised.

Application	URL	JavaScript files	size (KB)
GMail	`mail.google.com`	32	421
Google Calendar	`calendar.google.com`	5	360
Live Desktop	`www.mesh.com`	3	178
MSN	`www.msn.com`	3	17

Figure 16: Macro-benchmark information summary.

We also measure the overhead of the wrapping approach discussed earlier. Our measured performance of wrapper-based advice is 2–3x better than reported by others [33], perhaps due to using a different browser or our care in not inserting extraneous calls nor conflating policy logic with advice mechanisms. In almost all benchmarks, even naïve language-level support of advice with explicit `bless` calls (column “bless”) performs 2.7x faster than wrapping (column “wrapping”). Introducing further optimizations, like auto-blessing (column “auto-bless”), always outperforms wrapping with an average 2.9x speedup over wrapped invocation speed. While the benefits of language-based support were largely expected for user-defined functions (rows 1–2), speedup in advising native interpreter functions (tests 3–5) and DOM functions (tests 6–7), which are often privileged, is not as obvious.

7.2.2 Initialization Overhead

A survey of aspect literature shows that advice registration can be quite expensive: an unoptimized DOM wrapper approach takes 9 ms to initialize [27] and therefore can be as expensive as 100 ms on a mobile device [19].

In CONSCRIPT, starting a new application session creates a new interpreter session with globally available advice functions. A new local environment is created that aliases these advice function objects and global references to them are deleted, making the local environment privileged. Note that these manipulations only pertain to generally small advice functions, not the large set of DOM and JavaScript library functions that would be necessary for complete mediation in other approaches. Policies are then run, not in the global environment, but the privileged local one.

The initialization overhead is quite small in our experiments. Based on a trial of 10,000 runs, creating the privileged environment and removing global access costs only 24 μs. The remaining costs for loading policies are analogous to the optimized process of handling source attributes for a `<SCRIPT>` tag.

7.3 Runtime Overhead on Macro-benchmarks

While the experiments on micro-benchmarks above show superiority of CONSCRIPT compared to other techniques, the real test of our system is when it comes to advising large existing applications. To this end, we automatically generate policies, as discussed in Section 6, and apply them to large, JavaScript-heavy sites and applications such as MSN, Google

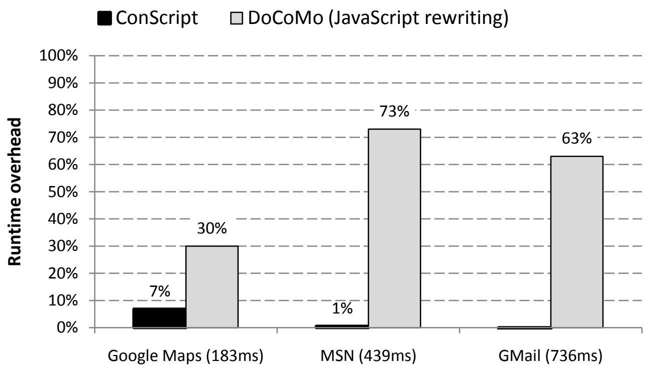


Figure 17: Macro-benchmarks: enforcing policies in Section 6.2 using ConScript has low overhead, compared to rewriting techniques.

Maps, and Google Calendar, etc. to measure the performance overhead in normal use.

For every experiment, we locally cached all resource requests and performed dynamic rewriting of Web pages to add our advice to the <HEAD> section using the Fiddler proxy [25]. For performance measurements of large highly interactive Web applications, deciding *how* to measure the overhead presents a difficulty. Our strategy has been to find two runtime events that do not exhibit much runtime variance such as the `onload` event and the first `XmlHttpRequest` being issued. This approach for experimenting with third-party sites was previously advocated in the AjaxScope project [21].

7.3.1 Overhead of Private Methods in Script# Policy

For the policy in Section 6.1, our protection of Script# private methods exhibits two kinds of runtime costs: instantiation overhead during application loading and then runtime monitoring overhead. Of concern, to protect a private method, all public entry points in the same class are also instrumented. This is unlike our other policies as the cost may be linear in the program size.

For this experiment, we applied the policy to the Live Desktop application that is part of the application suite located at `www.mesh.com`. We instrumented two core file and folder manipulation class files of the Live Desktop shared desktop application to evaluate these overheads, spanning 5% of the main library (23 private methods and 32 public ones out of 1,327 total functions). We have only automated policy generation given the namespace information: while the C# compiler generates this information, we manually extracted it for our benchmark. We averaged our overhead numbers over 20 trials, with most network resources cached locally.

There is no statistically significant impact on loading the application when measured from beginning to end (and forcing caching). The entire set of 55 method policies was installed in 1 ms or less — our JavaScript timer does not provide finer granularity — representing at most 0.3% of the processing time (and our micro-benchmarks suggest much less). For the task of opening a folder, 2 instrumented private calls are made, with 40 invocations of other methods in the same classes, accounting for 1.4% of the calls. We detected no statistically significant overhead: the average 0.9% slowdown is below the 3.5% range of experimental error.

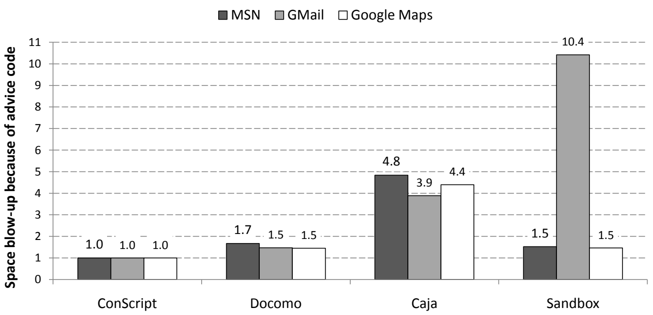


Figure 18: Code size increase for different instrumentation technologies.

7.3.2 Overhead of Intrusion Detection Policy

For the intrusion detection policy in Section 6.2, to perform our experiments, we created a representative list of 15 common attack vectors. For example, `postMessage` and `XDomainRequest` can be used to circumvent the single origin policy and functions are typically coded without considering the possibility of `defineProperty` changing the behavior of field access. Next, we monitored scripts sent to a browser from the server, checking against a master list of privileged DOM functions. Whatever did not occur was added to the final blacklist. As we cannot directly access the code of third-party sites, we used Fiddler to rewrite pages received by our test browser to load our blacklist advice, mimicking our suggested deployment approach.

We report the performance impact as part of our overall performance analysis, as shown in Figure 17. We ran 30 trials of uninstrumented and instrumented versions. We compare measured slowdown to that reported for JavaScript rewriting overhead [23], though, unfortunately, error information is unavailable. Fundamentally, when blacklisting functions under benign scenarios, the only cost is in instrumenting blacklist functions at initialization time. For most applications we tested (including Google Calendar, which is not shown), the standard deviation of the slowdown was 5%, with the average slowdown being negligible (0%). While we show an average 7% overhead on Google Maps, as its slowdown deviation is 46%, the slowdown is not statistically significant. Note that we have not observed any intrusion detection alarms while testing these applications.

7.4 Code Size Increase

JavaScript code *size* is a major concern for Web application performance [26], which becomes especially acute for mobile devices with limited storage capacity and for Web applications in general where resources are transferred over the network. Our timing benchmark measurements were performed on locally cached files. However, we must consider the file size increases related to verbose policies and rewriting, as initial network transfer time is crucial to fast application loading.

Figure 18 shows that our advice system has small and constant space overhead relative to other approaches, which, in contrast, have a cost linear in the size of application. We compare CONSCRIPT with the size blowup of running Docomo [23][3], Caja [30], and WebSandbox [17].

We selected JavaScript files from MSN, GMail, and Google Maps and ran them through existing rewriting tools [17, 30], or used previously reported results when no tool was available [23]. Reflecting best practices, we then run both the input and output through a suite of JavaScript compressors and pick the smallest file size.

For all, we compare the initial file size to the size of the secured one. Our policies add an average 0.7 KB to the compressed file size. As applications grow in size, relative cost decreases because the policy size is constant in most of our examples. In contrast, variable in the source rewriter and the application, a cost linear in the source size was incurred. We show the average linear cost per rewriter; we suspect the importance of considering the application is that different source generation tools (e.g., minifiers or tier-splitters) were used on a per-application basis.

8 Related Work

We implement much of the vision previously proposed by Erlingsson and Livshits [8] and examine the unaddressed problem of writing secure programmatic policies. There have been significant advances since the original proposal:

Static analysis. Policies might be phrased as properties to be statically verified. On benchmarks for a control-flow analysis, Guha et al. found large JavaScript applications need context sensitivity prohibitively higher than that for applications written in more static languages [14]. Evaluating a points-to analysis, Guarnieri et al. [13] found JavaScript widgets (that are typically between 50-250 lines) utilize a more tractable language subset. It is still unclear, however, how to apply such a static analysis to large, expressive Web applications.

Type systems. Our type system in Section 4.3 provides a form of fully-static checking and considers the subtleties of JavaScript, unlike Chugh's [3]. It is derived from label-based information flow type systems like Myer's [32] and Pottier's [35]. Non-interference was too strict of a property for our domain; under the object capability threat model, we must simply prevent references to policy heap objects from leaking. Inference is well-studied for such systems; further aiding usability, due to our safety properties and interpreter modifications, we only require policy code to pass the checker.

Browser tags. There are several proposals for modifying browsers to support coarse tag-based policies. For example, BEEP [18] introduces both a <noscript> tag to disallow scripts in descendant nodes and an application meta-tag for hash-based whitelisting of dynamically loaded scripts. Our more general script pointcuts enable encoding these primitives by supporting context-sensitive advice at the point in which a script enters the interpreter. MashupOS [16] proposes open and closed sandbox tags. These enable an application to load a script and manipulate the script's content while preventing the script from manipulating the application. Section 5 lists examples of CONSCRIPT policies that largely obviate the need for specialized tags.

Isolation languages. ADSafe [4], FBJS [10], Caja [30], and WebSandbox [17] are JavaScript variants designed to run untrusted gadgets in isolation from the rest of a page without modifying browsers. ADSafe syntactically checks that a gadget does not use many JavaScript language features such as the `this` object; this is analogous to our heavily restricted policy language subset except the entire program must be written in it. The rest support larger JavaScript subsets by rewriting gadget source to perform dynamic checks and lookup translations.

Deep wrapping. The above isolation systems share DOM API access between gadgets by providing a proxy API that relays commands. They are *deep* or recursive in the sense that function return values that are functions or objects, such as for document content, must also be wrapped in order to control interactions with them. Guha et al. [15] similarly attempt to restrict the input and output values passing through a function given a developer's specification of the desired type by wrapping and monitoring the function and any values reached through it. These enforcement mechanisms are realizations of the membrane pattern [27, 29] for JavaScript.

Shallow wrapping. Instead of the pervasively wrapping and rewriting, an option is to only advise particular method calls [33, 34] by dynamically reassigning an object's method to point to a wrapped, instrumented verison. While this may be acceptable for some use cases like debugging [33] that tolerate error, it is inappropriate for securing large APIs. For example, there are many aliases to the function `eval` that must be manually enumerated, and, unlike the rewriting and deep wrapping systems, there are no controls against inadvertant escaping of aliases. As a result, we found these systems to be susceptible to our policy integrity attacks.

Weaving aspects into source code. A traditional technique for implementing aspects that solves the above aliasing problem is, instead of forcing the developer to rewrite all aliases to a function, to just rewrite the function. Kikuchi et al. [23] and Washizaki et al. [39] demonstrate this idea for JavaScript by introducing a serverside proxy to rewrite outgoing pages.

[3] The measurements of Kikuchi et al. [23] are copied from the reported ones as there was no public way to reproduce them.

Unfortunately, the server cost is not negligible. Furthermore, JavaScript is dynamic: traditional aspect weavers consume type-based pointcuts, which is too imprecise for JavaScript. Using references for finer pointcuts currently requires pervasive rewriting to pinpoint enforcement locations at runtime. Next, the DOM API contains many privileged functions: there is no source code available to rewrite. We avoid these problems by instrumenting the interpreter.

Aspect interfaces for dynamic languages. How aspects are exposed to developers is crucial. As in the above example, pointcuts are too imprecise if specified with type signatures. We do not grant ambient authority to aspects [29]: instead of accepting a variable or function name as a pointcut as Washizaki et al. do [39], we advocate *requiring* a reference to a function in order to be allowed to advise it. Kikuchi et al. [23] suggest developers control JavaScript applications using a new XML-based language with code template and state machine tags. In contrast, we propose the single succinct construct `around`. This construct in conjunction with libraries may be used to develop other forms of advice (*before*, etc.) and to provide pointcut combinators.

Secure aspects. While a traditional use case for aspects has been for enforcing cross-cutting security (safety) properties, discussion of securing aspect systems is more recent. Dantas et al. [5] explore how to provide a non-interference property: a program may not be behaviorally modified by a malicious aspect. We weaken this property to abide by the object capability model: advice may only apply to a function given a reference to the function. We primarily focus on an opposite threat model: aspects should be protected from subsequent code. For example, we show how the locally declared objects of a policy, like a whitelist, may be verified for inaccessibility outside of the policy system.

9 Conclusions

This paper presents ConScript, a system that implements client-side *deep* advice for security. ConScript has been implemented by extending the Internet Explorer 8 JavaScript engine. To demonstrate the expressive power of ConScript, we presented 17 security and reliability policies that are specific to an application and are drawn from literature, practice, and analysis. We applied a type system to these policies to ensure that, once they type-check, they are free from a range of common bugs that were found in JavaScript policies before. We further presented two strategies for *automatically* producing ConScript policies through static and runtime analysis, demonstrating policies need not be created by hand, and that ConScript provides expressive primitives that simplify the implementation of policy generation tools.

We conducted a range of experiments with ConScript, both using micro-benchmarks and large, popular Web applications. In our extensive experiments, both the time and space overhead of ConScript has been demonstrated to be negligible for most large applications, hovering around 1%, which is often orders of magnitude smaller than what had been shown by previously published techniques.

References

[1] A. Barth, C. Jackson, and J. C. Mitchell. Securing Frame Communication in Browsers. In *Proceedings of the Conference on Security Symposium*, pages 17–30, 2008.

[2] B. Chess, Y. T. O'Neil, and J. West. JavaScript Hijacking. `www.fortifysoftware.com/servlet/downloads/public/JavaScript_Hijacking.pdf`, Mar. 2007.

[3] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged Information Flow for JavaScript. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 50–62, 2009.

[4] D. Crockford. ADsafe. adsafe.org.

[5] D. S. Dantas and D. Walker. Harmless Advice. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 383–396, 2006.

[6] ECMA International. ECMA-262: ECMAScript language specification, version 3.1, Sept. 2009.

[7] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[8] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end Web Application Security. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[9] E. Eskin, S. J. Stolfo, and W. Lee. Modeling System Calls for Intrusion Detection with Dynamic Window Sizes. *DARPA Information Survivability Conference and Exposition,*, 1:0165, 2001.

[10] Facebook. FBJS - Facebook Developer Wiki, July 2007. http://wiki.developers.facebook.com/index.php/FBJS.

[11] Google Web Toolkit. `http://code.google.com/webtoolkit`.

[12] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for java. *SIGPLAN Notices*, 42(10):321–336, 2007.

[13] S. Guarnieri and B. Livshits. Gatekeeper: Mostly Static Enforcement of Security and Reliability Policies for

JavaScript Code. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.

[14] A. Guha, S. Krishnamurthi, and T. Jim. Using static analysis for AJAX intrusion detection. In *Proceedings of the International Conference on World Wide Web*, pages 561–570, 2009.

[15] A. G. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi. Relationally-parametric Polymorphic Contracts. In *Proceedings of the Symposium on Dynamic Languages*, pages 29–40.

[16] J. Howell, C. Jackson, H. J. Wang, and X. Fan. MashupOS: Operating System Abstractions for Client Mashups. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, May 2007.

[17] S. Isaacs and D. Manolescu. Web Sandbox - Microsoft Live Labs. `http://websandbox.livelabs.com/`, 2009.

[18] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Embedded Policies. In *Proceedings of the International World Wide Web Conference*, 2007.

[19] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodk. Parallelizing the web browser. In *Workshop on Hot Topics in Parallelism*, March 2009.

[20] A. Kennedy. Securing the .NET Programming Model. *Theor. Comput. Sci.*, 364(3):311–317, 2006.

[21] E. Kıcıman and B. Livshits. AjaxScope: a Platform for Remotely Monitoring the Client-Side Behavior of Web 2.0 Applications. In *Proceedings of Symposium on Operating Systems Principles*, Oct. 2007.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 327–353, 2001.

[23] H. Kikuchi, D. Yu, A. Chander, H. Inamura, and I. Serikov. JavaScript Instrumentation in Practice. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, pages 326–341, 2008.

[24] N. Kothari. Script#. `http://projects.nikhilk.net/ScriptSharp/`, 2008.

[25] E. Lawrence. Fiddler: Web debugging proxy. `http://www.fiddlertool.com/fiddler/`, 2007.

[26] B. Livshits and E. Kiciman. Doloto: Code splitting for network-bound Web 2.0 applications. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, Sept. 2008.

[27] L. A. Meyerovich and A. P. Felt. Object Views: Fine-Grained Sharing in Browsers, November 2009.

[28] Microsoft Corporation. Microsoft Live Labs Volta. `http://labs.live.com/volta/`, 2007.

[29] M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[30] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - Safe Active Content in Sanitized JavaScript, October 2007. http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf.

[31] Mozilla Foundation. Using JavaScript code modules, Sept. 2009.

[32] A. C. Myers. JFlow: practical mostly-static information flow control. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[33] P. H. Phung, D. Sands, and A. Chudnov. Lightweight Self-protecting JavaScript. In *Proceedings of the International Symposium on Information, Computer, and Communications Security*, pages 47–60, 2009.

[34] R. Porotnikov. AOP fun with JavaScript, July 2001. http://www.jroller.com/deep/date/20030701.

[35] F. Pottier and S. Conchon. Information flow inference for free. *SIGPLAN Notices*, 35(9):46–57, 2000.

[36] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the Operating System Design and Implementation Conference*, 2006.

[37] The JS.Yamanner worm. `http://www.f-secure.com/v-descs/yamanner_a.shtml`, 2006.

[38] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions using System Calls: Alternative Data Models. In *IEEE Symposium on Security and Privacy*, volume 0, pages 1–33, 1999.

[39] H. Washizaki, A. Kubo, T. Mizumachi, K. Eguchi, Y. Fukazawa, N. Yoshioka, H. Kanuka, T. Kodaka, N. Sugimoto, Y. Nagai, and R. Yamamoto. AOJS: Aspect-Oriented JavaScript Programming Framework for Web Development. In *Proceedings of the Workshop on Aspects, Components, and Patterns for Infrastruc-*

ture Software, pages 31–36, 2009.

[40] C. Yue and H. Wang. Characterizing Insecure JavaScript Practices on the Web. In *Proceedings of the International Conference on World Wide Web*, pages 961–970, 2009.