copyrightbox

Efficient Runtime-Enforcement Techniques for Policy Weaving *

Richard Joiner[†], Thomas Reps[†,‡], Somesh Jha[†], Mohan Dhawan[$], Vinod Ganapathy[&]

[†]University of Wisconsin-Madison, USA [‡]GrammaTech, Inc., Ithaca, NY, USA
[$]IBM Research, New Delhi, India [&]Rutgers University, Piscataway, NJ, USA
{joiner,reps,jha}@cs.wisc.edu, mohan.dhawan@in.ibm.com, vinodg@cs.rutgers.edu

ABSTRACT

Policy weaving is a program-transformation technique that rewrites a program so that it is guaranteed to be safe with respect to a stateful security policy. It utilizes (i) static analysis to identify points in the program at which policy violations might occur, and (ii) runtime checks inserted at such points to monitor policy state and prevent violations from occurring. The promise of policy weaving stems from the possibility of blending the best aspects of static and dynamic analysis components. Therefore, a successful instantiation of policy weaving requires a careful balance and coordination between the two.

In this paper, we examine the strategy of using a combination of *transactional introspection* and *statement indirection* to implement runtime enforcement in a policy-weaving system. In particular,

- Transactional introspection allows the state resulting from the execution of a statement to be examined and, if the policy would be violated, suppressed.
- Statement indirection serves as a light-weight runtime analysis that can recognize and instrument dynamically generated code that is not available to the static analysis.
- These techniques can be implemented via static rewriting so that all possible program executions are protected against policy violations.

We describe our implementation of transactional introspection and statement indirection for policy weaving, and report experimental results that show the viability of the approach in the context of real-world JavaScript programs executing in a browser.

Categories and Subject Descriptors

D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms

Security, verification

* Supported, in part, by DARPA under cooperative agreement HR0011-12-2-0012. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies. T. Reps has an ownership interest in GrammaTech, Inc. Thanks to M. Zhivich, S. Maddi, and M. Rabe at MIT Lincoln Laboratory for invaluable feedback on our implementation. Thanks to the anonymous reviewers for many good suggestions.

FSE'14, November 16–22, 2014, Hong Kong, China.

http://dx.doi.org/10.1145/

Keywords

Security policy enforcement, dynamic runtime verification, transactional introspection, statement indirection, speculative execution

1. INTRODUCTION

Policy weaving for security-policy enforcement is a program-rewriting approach oriented towards striking a balance between static and runtime analysis techniques [**? ?**]. It is:

- *Sound for policy violations*: Each program trace that is prevented is one that violates the policy.
- *Complete for policy violations*: All program traces that violate the policy are prevented.
- *Transparent*: The rewritten program has the same semantics as the original program, modulo policy violations.

We refer collectively to these properties as *correctness*. The policy-weaving approach is motivated by the acknowledgment that in non-trivial scenarios, static analysis alone cannot prove that a program adheres to a policy [**? ?**]. In contrast, runtime analyses such as inlined reference monitoring (IRM) [**? ? ?**] can be complete for policy violations by testing properties of concrete states at runtime, but at the cost of degraded performance. In addition, they may not be sound because it may be difficult from a pragmatic standpoint to exclude only policy-violating traces. To address these limitations, policy weaving is a hybrid approach that (i) attempts to statically identify sections of the program that can be proven safe, and (ii) rewrites the program to include runtime checks at locations where policy state may be affected. This approach, while harnessing the best of both worlds, also creates a new challenge: that of coordinating the interoperability of—and managing the trade-offs between—the static and runtime analyses. Our goal in this paper is to present and evaluate a runtime policy-enforcement mechanism that is well-suited to be the target of a static weaving algorithm.

Specifically, we investigate the utility of *transactional introspection* applied to a program through static rewriting of source code, and an additional *statement-indirection* transformation to enable just-in-time transactional introspection of code that is generated at runtime. We describe the resulting end-to-end policy-enforcement system, and present experimental data that shows the viability of this approach when applied to real-world JavaScript applications in a browser context.

Transactional introspection is a straightforward and powerful tool for securing untrusted code. Transactions allow speculative execution, meaning that the effects of an execution can be computed and examined prior to the application of those effects to the environment. The ability to *introspect* on actions without committing their effects is necessary when dealing with programs that perform irrevocable actions, such as the initiation of an HTTP request. Rollback of communication and other I/O actions is not possible in general, so a security-policy-enforcement mechanism that aims to mediate such actions must recognize and potentially suppress such events prior to their occurrence, rather than reacting after the fact.

Transactions provide this capability to "peek into the future" and take preventive action to avoid policy violations.

Transactional introspection has been studied in prior security work [**? ? ? ?**], and some difficulties have been identified:

- Performance overhead of speculative execution can be prohibitive [**? ?**].
- Correct placement of transactional instrumentation is a nontrivial task [**? ? ?**].
- Implementation of introspection logic to recognize and prevent policy violations can be error-prone [**?**].

In this paper, we address each of the concerns that have been raised. In particular, the complexity of both manually placing individual transactions and manually constructing introspection code is replaced by the requirement to formulate an explicit security policy as an automaton. Security-policy automata allow programmers to state their intentions and goals explicitly in a typically small and self-contained artifact. Moreover, the use of transactions produces substantial benefits in terms of performance and flexibility when compared to other enforcement mechanisms.

We also incorporate *statement indirection*, a concept introduced by Yu et al. [**?**], as a supplemental enforcement primitive. This mechanism allows us to apply transactional semantics to *generated code*, which is code that is created at runtime and is therefore unavailable to the static analysis. We use statement indirection to recognize *higher-order scripts*, language constructs that generate code (to be run immediately or at a later time), and to ensure that this code is executed inside a transaction. Examples of higher-order scripts abound in modern scripting languages, such as JavaScript (`eval`, `Function`), Python (`eval`, `exec`, `compile`), and Perl (`eval`). The use of statement indirection serves two purposes:

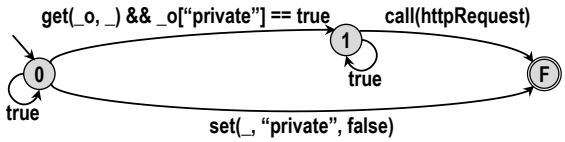
- The static analysis is freed from the generally impossible task of precisely modeling all possible executions of dynamically generated code.
- Indirection serves as a light weight but special-purpose alternative to the use of hard-coded transactions.

Our work makes use of and synthesizes several techniques from prior work:

- Runtime verification [**? ? ? ? ?**]: the monitoring and enforcement of safety properties during program execution
- Policy/aspect weaving [**? ? ? ?**]: the use of static analysis and rewriting for fine-grained placement of instrumentation
- Transactional introspection [**? ? ? ?**]: a powerful monitoring and enforcement tool that allows effects of program execution to be examined before being committed
- Statement indirection [**? ?**]: a light-weight monitoring tool that can apply instrumentation to dynamically generated scripts

However, integrating these diverse techniques is challenging. The specific contributions of our work are as follows:

- We show how a policy-weaving algorithm can be used to automatically weave sound, complete, and transparent transactional instrumentation into a program.
- We describe the automatic translation of a security-policy specification into introspection code, which substantially reduces the number of opportunities for implementation errors.
- We describe how statement indirection can be used to propagate transactional introspection to all dynamically generated scripts.
- We present JAMScript, a simple, general-purpose extension of the JavaScript language that implements transactional semantics with properties suited to security-policy enforcement.
- We present experimental results that demonstrate that our approach performs well compared to manual placement of transactions and other types of enforcement mechanisms that could be targeted by a policy-weaving algorithm.




```
0,1: get(_o, _) && _o["private"] == true
1,F: call(httpRequest)
0,F: set(_, "private", false)
```

Figure 1: Policy automaton (and its textual representation) that specifies a set of disallowed traces, namely, ones that "set the `private` property of any object to `false`" (the bottom path in the automaton) or "read any property of an object whose `private` property is `true` and subsequently make a network request" (the top path). `set`, `get` and `call` are special event predicates that unify their arguments with values. `_o` is a variable constrained to be the same value at each occurrence, and underscore (`_`) matches any value. `httpRequest` references a specific function, and other values are literals. In the textual representation, self-loops with the predicate `true` are implicit.

Organization: §2 presents a specification of the program-rewriting algorithm, the semantics of transactional introspection, the mechanism of indirection, and how these techniques fit together conceptually. §3 discusses implementation challenges encountered when applying our approach to JavaScript, and the solutions we devised. §4 evaluates the performance of JAMScript on a set of real-world applications. §5 examines related work and discusses various design choices we made that differ from other systems that share the same goals.

2. TECHNICAL OVERVIEW

In this section, we examine the properties that a runtime enforcement mechanism must satisfy to meet the requirements of a policy-weaving system. We identify transactional introspection and statement indirection as two key primitives of an enforcement strategy that meet these requirements. We then incorporate this two-part mechanism as the targeted enforcement platform used by a language-independent program-rewriting algorithm.

2.1 Enforcement Primitives

Three principles to be considered when choosing a runtime enforcement mechanism are as follows:

- Runtime overhead induced by the mechanism should be minimal to maintain an acceptable user experience.
- The mechanism should enable correct policy enforcement (i.e. sound complete, and transparent).
- The mechanism should be general-purpose and extensible, so that it can be used in arbitrary runtime environments.

Our thesis is that a policy-weaving strategy can, in effect, bring the enforcement mechanism closer to satisfying each of these principles, compared to manually placing instrumentation or relying on an automated modular partitioning strategy as suggested in other work [**? ?**]. When used with an enforcement mechanism based on transactional introspection, policy weaving produces the following benefits:

- Fewer program statements are run within transactions, thereby reducing the performance overhead of enforcement (as shown in §4).
- The policy automaton provided as input to the static analysis is automatically translated to introspection code that implements policy monitoring and enforcement at runtime. Examples are presented in §3, and the benefits are quantified in §4.
- By placing transactions via static rewriting, policy weaving can ensure certain restrictions on the semantics of introspection, such as sequential execution and a lack of side-effects. (We say more about these benefits later in this section.)

```
1 config = { request: "standard" };
2
3 fun getRemoteConfig() {
4   param = config.request
5   url = "http://config.example.com?" + param;
6   reply = httpRequest(url);
7   config.remote = eval(reply);
8 }
9
10 fun getUserConfig() {
11   opt, val = readConfigFromUser();
12   config[opt] = val;
13   config.done = askIfDone();
14 }
15
16 while (!config.done) {
17   getRemoteConfig();
18   getUserConfig();
19 }
   ...
```

(a) Original code for a program that reads configuration data from an untrusted server and from the user. It is vulnerable to code injection by the server, and to data leakage to the server.

```
1 config = { request: "standard" };
2
3 fun getRemoteConfig() {
4   introspect(policy0_1) { param = config.request; }
5   url = "http://config.example.com?" + param;
6   reply = indirect(CALL, [httpRequest, [url]], policy1_F);
7   config.remote = indirect(CALL, [eval, [reply]], null);
8 }
9
10 fun getUserConfig() {
11   opt, val = readConfigFromUser();
12   introspect(policy0_F) { config[opt] = val; }
13   config.done = askIfDone();
14 }
   ...
```

(b) Secured code. Gray highlights indicate code introduced by rewriting. Instrumentation consists of two primitives: (i) calls to `indirect` (lines 6 and 7) wrap statements that may invoke higher-order scripts, and (ii) `introspect` blocks (lines 4 and 12) are parameterized by a function that examines actions that occur during speculative execution. Implementation of the instrumentation functions is shown in Figure ??.

Figure 2: (a) A program that may violate the policy depicted in Figure 1, and (b) a secured version.

```
1 fun indirect(type, parts, ispect) {
2   if (type == CALL) {
3     fun = parts[0]; args = parts[1];
4     if (fun == eval)
5       ispect = policyFull;
      ... cases for other higher-order scripts ...
7     if (ispect != null)
8       introspect(ispect) { ret = fun.apply(args); }
9     else
10      ret = fun.apply(args);
11    return ret
12  }
    ... cases for other expression types ...
14 }
15
16 fun policy0_1(tx) {
17   for (r in tx.getReadSequence()) {
18     if (!policyStates[1] && r.object.private == true)
19       policyStates[1] = true;
20     commitAction(r);
21   }
22 }
23
24 fun policy1_F(tx) {
25   for (c in tx.getCallSequence()) {
26     if (policyStates[1] && c.target == httpRequest)
27       throw new ViolationException();
28     commitAction(c);
29   }
30 }
31 policyStates = [true, false, false];
32
33 fun policy0_F(tx) {
34   for (r in tx.getWriteSequence()) {
35     if (r.property == "private" && r.value == false)
36       throw new ViolationException();
37     commitAction(r);
38   }
39 }
40
41 fun policyFull(tx) {
42   for (a in tx.getActionSequence()) {
43     if (!policyStates[1] && a.type == GET && r.object.private == true)
44       policyStates[1] = true;
45     if (policyStates[1] && a.type == CALL && a.target == httpRequest)
46       throw new ViolationException();
47     if (a.type == SET && r.property == "private" && r.value == false)
48       throw new ViolationException();
49     commitAction(a);
50   }
51 }
52
53 fun commitAction(a) {
54   if (a.type == CALL && a.target == eval)
55     a.args[0] = "introspect(policyFull)"
56       + "{" + a.args[0] + "}";
     ... cases for other higher-order scripts ...
58   a.target.apply(a.args);
59 }
```

Figure 3: Implementation of the functions used to enforce the policy at runtime. The `policy*` functions examine actions that occur within an `introspect` block to potentially (i) update the policy state (maintained in the global `policyStates` list) or (ii) throw an exception to prevent a violation. The parameter `tx` provides access to sequences of actions recorded during speculative execution. `policyFull` implements introspection for the entire policy, while the other `policy*` functions are specialized to examine a subset of policy transitions. Each action is passed to `commitAction`, which (i) recognizes higher-order scripts, (ii) applies introspection to generated code, and (iii) applies the effects of the action to the program state. The `indirect` function (lines 1–14) dynamically determines the level of introspection to apply to a statement. If a higher-order script is detected (such as `eval`), the `policyFull` introspector is used. Otherwise, the input parameter `ispect`, if non-null, is applied when the original statement needs specialized introspection (as with line 6 of Figure ??(b)).

We now introduce an example program and a security policy, and explain how the policy can be enforced via transactional introspection and statement indirection. The policy shown in Figure 1 asserts that network communication should not occur after reading from objects marked "private," and that the "private" property cannot be set to false. The program shown in Figure **??**(a) potentially exhibits behaviors that violate the policy. We use a conservative static analysis to identify statements that may cause policy transitions. (The details of this analysis are outside of the scope of this paper.) This information is used to rewrite the statements, resulting in a transformed program, shown in Figure **??**(b), having the following properties:

- All statements that may directly affect the policy state—not through invocation of a higher-order script—are enclosed in *transaction blocks*, indicated by the keyword `introspect`.
- An *introspector* function is passed as the parameter to each transaction block; these functions are defined in Figure **??**.
- Each introspector evaluates a unique combination of policy predicates, and they each have read and write access to the global `policyStates` array.

This rewriting results in the following runtime behavior:

```
1  RewriteIntrospect(P, Φ, Ψ_Φ):
   Data: P: source code of the program being analyzed
         Φ: policy automaton as a set of transitions
         Ψ_Φ = {(s_0, τ_0), ..., (s_m, τ_n)}: a policy-violating witness
   Result: P rewritten to prevent Ψ_Φ
2  foreach (s, τ) ∈ Ψ_Φ do
3      Let T_s be the set of policy transitions for which s is already
         instrumented in P
4      Let s_orig be the original, uninstrumented statement corresponding to s
5      if τ is not null then
6          T_s ← T_s ∪ {τ}
7      if T_s = ∅ then
8          ispect_s ← null
9      else
10         Generate or retrieve ispect_s, the introspector function that
             implements Algorithm 1 for all τ' ∈ T_s
11     if s_orig potentially invokes a higher-order script then
12         P ← RewriteIndirect(P, Φ, s_orig, ispect_s)
13     else
14         Let s' be introspect(ispect_s){s_orig}
15         Replace s with s' in P
16 return P
```

Algorithm 1: RewriteIntrospect **specifies the static-rewriting step of a policy-weaving algorithm that applies direct transactional introspection or calls** RewriteIndirect **(see Algorithm 3) for statements that may invoke higher-order scripts. Each element (s, τ) of the witness Ψ_Φ consists of a program statement s and a policy transition τ. τ is either** `null` **(indicating that the statement does not directly induce a transition in this witness, but may invoke a higher-order script that does) or a triple $(\varphi_{pre}, \varphi, \varphi_{post})$ consisting of a policy state φ_{pre}, a policy state φ_{post}, and a predicate φ that induces the transition.**

```
1  Introspect(T_s, Actions, Reached, ispect_all):
   Data: T_s: sequence of policy transitions potentially induced by the
           introspected statement(s)
         Actions: sequence of recorded actions
         Reached: set of policy states reached during the current execution
         ispect_all: introspector function that enforces all policy transitions
   Result: Terminate execution if a policy violation is detected, or update
           Reached and commit each a ∈ Actions otherwise
   /* Examine each action in the order it was recorded        */
2  foreach a ∈ Actions do
       /* Evaluate each policy transition                     */
3      foreach (φ_i, φ, φ_j) ∈ T_s do
           /* If the policy-transition pre-state has been
              reached and the post-state has not              */
4          if φ_i ∈ Reached ∧ φ_j ∉ Reached then
               /* If the policy-transition predicate holds   */
5              if a ⊨ φ then
                   /* If the post-state is final             */
6                  if φ_j is final then
7                      Throw an exception and quit
8                  else
9                      Reached ← Reached ∪ {φ_j}
       /* Apply introspection to dynamically generated code  */
10     if a invokes a higher-order script then
11         Identify the text "t" of the code being generated
12         Replace "t" with "introspect(ispect_all){t}"
13     if a can affect the program state then
14         Commit a
```

Algorithm 2: Introspect **examines an action sequence at runtime and either commits or suppresses the effects. The condition $a \models \varphi$ is true if action a satisfies φ. This algorithm also identifies higher-order scripts and encloses the generated code within a transaction block.**

- During execution of a transaction block, all *actions* (reads, writes, and calls) are recorded sequentially, and the effects of writes are postponed. Calls to native functions may trigger transaction suspension (described in §**??**).
- When the closing brace of the transaction block is reached, the introspector is invoked with a *transaction-object* argument, through which all security-relevant information about the recorded actions can be accessed.
- The introspector examines the action sequence and determines whether to commit or suppress the recorded actions.

In addition to the direct weaving of transaction blocks described above, we define a second type of rewriting, called *statement indirection*, by which any statements that potentially invoke higher-order scripts are transformed so that transactional introspection can be selectively applied at runtime. Statement indirection is achieved by statically decomposing a statement that may invoke a higher-order script so that the individual constituents can be examined to determine their runtime values. The static analysis may identify statements that are candidates for both transactional introspection and statement indirection, such as line 6 in Figure **??** (assuming that the identifier `httpRequest` can target a higher-order script, in addition to the function of that name specified in the policy in Figure 1). In this case, introspection is subsumed into the indirection mechanism. The `indirect` function, shown in lines 1–14 of Figure **??**, determines whether to use the specialized introspector (`policy1_F` in this example) or the full-policy introspector (`policyFull`) if a higher-order script is detected.

Delaying the decision to perform full introspection on these statements serves as an important optimization in languages (including each of those previously mentioned) in which higher-order scripts can be invoked through indirect function calls or writes to object properties. In moderately large systems written in languages with these features, construction of a conservative call graph can be very imprecise. The use of statement indirection allows a rewriter to avoid enclosing a large percentage of all statements in transaction blocks, in favor of performing a runtime check to determine whether a given statement needs to be evaluated within a transaction.

2.2 A Formalization of Rewriting

The rewriting that occurs between Figure **??**(a) and Figure **??**(b) is accomplished via a modified and extended version of the rewriting step of the SafetyWeave algorithm formalized by Fredrikson et al. [**?**]. While that paper focuses primarily on the static policy-weaving algorithm, it assumes that the enforcement mechanism is an inlined reference monitor that evaluates the pre-image $pre(s, \varphi_{violation})$ of the subsequent statement s and policy predicate $\varphi_{violation}$ with respect to the current program state σ_{pre}, and terminates the program if $[\![pre(s, \varphi_{violation})]\!](\sigma_{pre}) = true$ (i.e., the execution of the statement could cause a policy violation).

A key insight that led us to integrate SafetyWeave with a transactional enforcement mechanism is that the semantics of introspection allows predicates constituting the policy to be directly evaluated in the context of the (speculative) program state. Viewed another way, a code fragment that consists of a transaction block containing a statement s implements the post-state operator $post$, which maps the program state σ_{pre} encountered at transaction entry and the enclosed statement s to the post-state σ_{post} that results from the execution of s: $post(\sigma_{pre}, s) = \sigma_{post}$. A policy predicate $\varphi_{violation}$ can then be directly evaluated in this post-state. Formally, the condition for policy violation is $post(\sigma_{pre}, s) \models \varphi_{violation}$.

The observations above allow the rewriting step of SafetyWeave to be replaced by a transactional-enforcement mechanism. Moreover, despite their similar goals, the implementations of the two approaches are quite different. Computation of the pre-image involves the static construction of a potentially complicated symbolic predicate to characterize the set of dangerous program states. In contrast, the post-state operator that we describe in this paper is available as a built-in feature of the interpreter, and it produces a single program state in which $\varphi_{violation}$ is evaluated.

```
1  RewriteIndirect(P, Φ, s, ispect_s):
   Data: P: source code of the program being analyzed
         Φ: policy automaton as a set of transitions
         s: a program statement
         ispect_s: introspector function that implements Algorithm 1
           for all policy transitions that s may directly affect
   Result: P with s rewritten by statement indirection
     over all policy transitions to higher-order scripts invoked by s
3  Let e be the first subexpression of s (according to execution order) that
     (i) may affect the policy state, directly or by invoking a higher-order script
     (ii) is not already within an indirection expression
4  if e is not null then
5      Let type_e be the expression type of e
6      Let parts_e be an array of subexpressions comprising e
7      Retrieve ispect_all, the introspector function that implements
         Algorithm 1 for all transitions τ ∈ Φ
8      Let e' be Indirect(type_e, parts_e, ispect_all, ispect_s)
9      Let s' be the result of replacing e with e' in s
10     Replace s with s' in P
11     return RewriteIndirect(P, Φ, s', ispect_s)
12 return P
```

Algorithm 3: RewriteIndirect **specifies the transformation that applies indirection to statements that potentially invoke higher-order scripts. Each subexpression that potentially causes a policy transition (as determined by the static analysis) is replaced by a runtime call to** `Indirect`**, which is defined in Algorithm ??.**

```
1  Indirect(type_e, parts_e, ispect_all, ispect_s):
   Data: type_e: expression type of e
         parts_e: array of values of the subexpressions that comprise e
         ispect_all: introspector function that enforces all policy transitions
         ispect_s: introspector function (possibly null) originally applied to s
   Result: Value of expression e, or terminate execution in case of a policy
     violation
2  if e invokes a higher-order script (determined by type_e and parts_e) then
3      Let ispect_effective be ispect_all
4  else
5      Let ispect_effective be ispect_s
6  Reassemble expression e from parts_e
7  if ispect_effective is null then
8      Evaluate expression r = e
9  else
10     Evaluate expression introspect(ispect_effective){r = e}
11 Return r
```

Algorithm 4: Indirect **is the runtime target of expressions rewritten by Algorithm 3. It examines** $\texttt{type}_e$ **and** $\texttt{parts}_e$ **to recognize higher-order scripts, and determines the function** $\texttt{ispect}_{\texttt{effective}}$ **to use for introspection. By construction,** $\texttt{ispect}_{\texttt{all}}$ **subsumes** $\texttt{ispect}_s$**, so the detection of a higher-order script effectively results in a security upgrade.**

A policy-weaving algorithm may utilize arbitrary (conservative) static-analysis techniques to investigate a program's behavior with respect to a security policy Φ. However, in a departure from traditional static verification methodology, policy weaving uses a rewriting step, which serves as the interface between the static and runtime analyses. Rewriting is invoked in two situations.

- A valid execution trace that violates the policy is identified.
- An invalid execution trace is identified, and a configurable resource bound on how much effort is to be expended on refining the program abstraction has been met.

Here, a "valid" execution trace is one that is realizable in the concrete program, as determined via symbolic execution, and an "invalid" trace is one that cannot occur in the concrete program, but is extracted from the abstract model of the program due to imprecision. In the first case, rewriting the program allows the policy-weaving algorithm to convert a program that can generate policy violations into a safe, instrumented program. In the second case, rewriting allows the algorithm to terminate in bounded time and to use as much information as can be acquired by static analysis while staying within the resource bound. (The worst-case complexity of the static analysis would normally preclude its use if the results of a "full" static analysis were required.)

The nature of the runtime-enforcement mechanism (the primitive inserted into the program by the SafetyWeave rewriting step) is discussed in formal but generic terms in [**?**]. We now develop precise specifications of both the static rewriting step, which consists of calls to routine Algorithm 2 and subroutine Algorithm 3, and the procedures Algorithm 1 and Algorithm **??**, which implement transactional introspection and statement indirection at runtime.

In Algorithm 2, the "policy-violating witness" Ψ_Φ represents an execution trace along which the policy may be violated in an abstract model of the (partially instrumented) program.[1] The program is rewritten by Algorithm 2 to include transactional introspection for each statement s in the trace that can directly induce a policy transition τ. If s can invoke a higher-order script, the statement-indirection transformation is applied by Algorithm 3 so that the introspector (if any) can be chosen as needed at runtime. As the static analysis proceeds, witnesses that are not prevented by previously-applied instrumentation are identified and the constituent statements are rewritten. Eventually, no more witnesses will be found because, in the degenerate case, every statement in the program will be guarded by instrumentation that monitors the full policy. When no witnesses can be found, the analysis system has proven that no policy-violating execution trace is realizable in the program.

In contrast to the rewriting step specified in past work on policy weaving [**?**], which requires an implementation of the pre-image operator to determine when s will cause a policy transition, the formulation described here relies on the inherent properties of transactional introspection to effectively apply the post-state operator in the form specified in Algorithm 1 (i.e., as a sequence of action commits).

2.3 Transaction Suspension

While the term "transaction" often implies atomic execution of a block of code, our goal of security-policy enforcement does not impose this requirement. In fact, we find that maintaining both atomicity and security—in the presence of actions that have externally visible or unpredictable results—is impractical at best. In an imperative language with a built-in I/O interface, for example, it may be impossible to execute I/O actions speculatively while allowing for the possibility of suppression of such actions. Therefore, we use *transaction suspension* [**?**] as a means of escaping execution of the transaction block into the introspection code where the speculative state can be evaluated and committed prior to resumption. Due to this process, a transaction may end up being committed as a succession of partial commits, rather than as an atomic unit.

The introduction of transaction suspension does not require alteration of Algorithm 1. Upon suspension, the action sequence available through the transaction object will be a consistent prefix of the full action sequence for the entire transaction that would be available in the absence of suspension. Therefore, these actions can be examined, and the policy state updated, just as before.

We restrict the application of transaction suspension to predictable circumstances; in particular, it is useful for suspension to occur at all invocations of native or externally-linked functions. A whitelist of functions that are free of side-effects—and therefore

[1] More precisely, Ψ_Φ represents a *collection* of traces. It is a sequence of (statement, policy-transition) pairs and thus does not include the statements in the concrete trace that do not affect the policy state. In general, there will be many ways in which the "gaps" between policy-transitions could be filled in to create a specific concrete trace. The rewriting step serves to prevent all of these traces.

do not need to trigger a suspension—can be maintained to reduce overhead. We discuss specific scenarios in which transaction suspension is useful in §3, but we maintain that some form of suspension will be necessary to support transactions in any imperative language that incorporates external interfaces.

2.4 Correctness

The specification of indirection developed by Yu et al. [**?**] is accompanied by proofs of completeness for policy violation (which that paper refers to as soundness) and transparency with respect to a formal semantics of CoreScript, a language subset of JavaScript. Proofs for the JAMScript framework with respect to the full JavaScript language would follow a similar trajectory, but are outside the scope of this paper.

Intuitively, completeness—prevention of all policy violations by the top-level script and any generated scripts—can be established by first noting that any higher-order script invoked within an `introspect` block will trigger a suspension of the currently executing transaction, at which point the generated script is extracted and wrapped in another transaction block (see lines 10–12 of Algorithm 1). Likewise, all higher-order scripts will be instrumented with statement indirection by the rewriting specified in Algorithm 3 so that transactional introspection over the full policy is applied at runtime to the generated script. In this way, introspection is propagated transitively to all scripts that are executed directly or indirectly in an environment.

Soundness and transparency—maintenance of correct program semantics in the absence of policy-violating execution traces—can be established informally by inspecting the definitions of transactional introspection and statement indirection. Transactional introspection implemented by Algorithm 1 and woven by Algorithm 2 is guaranteed, by construction, to (i) have no visible side-effects, and (ii) maintain sequential execution up to policy violation. That is, the execution of a transaction block always produces the same results as a prefix of the original unprotected code. Statement indirection implemented by Algorithm **??** and woven by Algorithm 3 is a simple syntactic transformation that delays execution of an expression to allow inspection of its constituents. Its only function is to apply transactional introspection, which results in sound, complete, and transparent policy enforcement, as shown above.

3. IMPLEMENTATION

This section discusses the implementation of the JAMScript extension to the JavaScript language, and the integration of its primitives as the target of a policy weaver. Source code and documentation for the two components are available online: (i) the JAM policy weaver is at `https://github.com/blackoutjack/jamweaver` and (ii) the JAMScript enforcement mechanism is at `https://github.com/blackoutjack/jamscript`.

3.1 A Strawman Approach

An initial attempt we made to implement speculative execution, which we refer to as ForkIsolate, failed to have acceptable performance to serve as a policy-weaving enforcement mechanism. As the name indicates, each time speculative mode was entered, the browser process was duplicated via a fork. The protected code was then executed in the new branch, and the resulting state was reported back to the original process. Despite copy-on-write semantics for a forked process, the overhead incurred by this mechanism caused some applications to run 3 orders-of-magnitude slower than unprotected execution. As an example, decrypting a block of text in the unprotected `snote` application takes about 13ms; with a version monitored by ForkIsolate instrumentation, this same action took 16659ms! (With our current implementation, decryption takes around 17ms.)

Our experience with ForkIsolate also showed the difficulty of developing a speculative-execution framework that can achieve correct enforcement of JavaScript embedded in the browser environment. We found that due to the underlying mechanism of isolation used by ForkIsolate, several classes of statements, for example those involving calls to DOM methods that expected access to a GUI (such as `window.alert`), resulted in non-transparent behavior, including freezes and crashes. A wide range of such scenarios would have required special handling to avoid all unexpected and unpredictable behaviors.

The task of performing complete enforcement in ForkIsolate was also a struggle. As constructed, the mechanism was incomplete because only the final state of the speculative execution was considered, so it was possible for the analyzed code to violate the policy and then revert to a non-violating state, in which case the policy violation would be undetected. Completeness, like transparency, was also hindered by environment-specific complications. For example, if the `HTMLElement.prototype.appendChild` DOM method is called during speculative execution, it is necessary to identify all HTML `script` elements contained in the argument and instrument the JavaScript code within. Given the architecture of the ForkIsolate system, these cases had to be recognized and handled by hard-coded logic.

Consequently, converting ForkIsolate from a prototype into a fully correct enforcement mechanism for policy weaving would have required (i) special handling for a wide range of scenarios, and (ii) a finer granularity of state monitoring. In response to these challenges, we developed JAMScript, an enforcement mechanism that (i) is extensible by allowing specialized enforcement capabilities to be loaded as necessary for each embedding system, and (ii) uses transactions to record a sequence of all actions that occur during speculative execution.

3.2 Description of JAMScript

With JAMScript, we adopt transactional introspection as the key enforcement component for static policy weaving. We have implemented the mechanism as an extension to the JavaScript language in accordance with the formalization described in §2. The extension consists of one new keyword, `introspect`, to indicate the opening of a transaction block that protects a given fragment of code (delimited by enclosing curly braces). A function value is passed as a parameter to the block to provide the introspection logic.

Additional utilities for runtime enforcement, such as the implementation of the `indirect` function described in §2, are provided as part of the `JAM` library. This code is written in JavaScript and is loaded prior to loading the rewritten program, and after loading the policy logic that is generated at rewriting time by Algorithm 2. Because the `JAM` library methods are accessible as source code, rather than being built into the interpreter, they can be extended in a straightforward manner to handle constructs provided by embedding systems (such as the browser). We have developed a fully functional version of this library for the core JavaScript language, and a prototype implementation that accounts for DOM and other Web API constructs provided by a browser environment.

In accordance with §**??**, invocation of methods provided by systems outside of the core JavaScript language, for example the DOM, causes suspension of a running transaction to allow the ambient memory state to "catch up" to the speculative state. In light of this approach, external systems that utilize our modified JavaScript interpreter do not need to make special accommodations for the potentially transactional nature of what is occurring in the JavaScript heap. A complementary consequence is that the JavaScript interpreter does not need to be aware of the systems into which it is integrated. JAMScript's simple model of suspension decouples the transactional semantics of the JavaScript interpreter from the semantics of the embedding system.

To achieve correct enforcement of scripts generated by calls to `document.write`, our implementation includes a few modifica-

```
1 Object.defineProperty(this, "JAM", { value: (function() {
2     var _eval = eval;
3     var _call = Function.prototype.call;
4     var _document_write = HTMLDocument.prototype.write;
5     var _Array_slice = Array.slice;
6
7     return {
8       policy: policy,
9
10      commitAction: function(a) {
11        if (a.type === "call") {
12          var fun = a.target, args = a.args, obj = a.object;
13          if (fun === _eval) {
14            args[0] = "introspect(JAM.policy.full)"
15              + "{" + args[0] + "}";
16          } else if (fun === _document_write) {
17            JAM.pushDynamicIntrospector(JAM.policy.full);
18            var ret = fun.apply(obj, args);
19            JAM.popDynamicIntrospector();
20            return ret;
21          }
22          return fun.apply(obj, args);
23        }
24      },
25
26      call: function(f, rec, args, ispect) {
27        if (f === _call) {
28          f = rec; rec = args[0];
29          args = _Array_slice(args, 1);
30          return JAM.call(f, rec, args);
31        }
32
33        if (f === _eval || f === _document_write)
34          ispect = JAM.policy.full;
35
36        if (ispect !== undefined)
37          return introspect(ispect) { f.apply(rec, args); }
38        else
39          return f.apply(rec, args);
40      },
41    }     /* end JAM object literal */
42  })()    /* end anonymous function call */
43 });       /* end call to Object.defineProperty */
44 Object.freeze(JAM);
```

Figure 4: Initialization of the `JAM` library and definition of the `call` and `commitAction` methods that implement statement indirection and instrumentation of code generated by expressions that invoke `eval`, `HTMLDocument.prototype.write`, and `Function.prototype.call`. (Cases for other higher-order scripts, error cases, and additional library methods are elided to conserve space.) Lines 2–5 define private references to native objects. The `policy` object (defined in Figure 4) is incorporated (line 8), and the `JAM` library object is frozen (line 44) so that its properties cannot be overwritten.

tions to source files that implement DOM functionality. As noted by Yu et al. [**?**], `document.write` (or `writeln`) can be used to output an HTML `script` element in pieces over multiple calls. Therefore, the full text of the generated code may not be available to the JAMScript instrumentation at any point. We address this issue with two new native functions, `JAM.pushDynamicIntrospector` and `JAM.popDynamicIntrospector`, that respectively add to and remove function values from a stack. The modified DOM implementation associates these functions with new scripts as they are created. When such a script begins execution, the JavaScript interpreter runs it as if it were enclosed in (nested) transaction blocks with the associated functions as the introspectors. Lines 16–21 in Figure **??** show how the `JAM` library makes use of this facility.

3.3 Protecting the Instrumentation

A concern when developing a runtime security technique is to protect the instrumentation itself from being modified or bypassed. The JAMScript framework leverages JavaScript's lexical scoping and closures to create immutable and self-enclosed enforcement code. All native objects that will be referenced during policy monitoring are statically known, so the instrumentation is packaged as objects with methods that close over these references when created.

```
1 var policy = (function() {
2   var _HTMLImageElement = HTMLImageElement;
3
4   function policy0_F(tx) {
5     var commit = true;
6     var ws = tx.getWriteSequence();
7     for (var i=0; i<ws.length; i++) {
8       var a = ws[i];
9       if (a.type === "write" && a.id === "src"
10          && a.object instanceof _HTMLImageElement)
11        commit = false; break;
12    }
13    if (commit) JAM.commit(tx);
14    else JAM.prevent(tx);
15  }
16
17  return {
18    policy0_F: policy0_F,
19    full: policy0_F
20  };
21 }());
22 Object.freeze(policy);
```

Figure 5: Definition of a single-transition `policy` object preventing writes to the `src` property of `HTMLImageElement` objects. `_HTMLImageElement` on line 2 is a private reference needed for consistent evaluation at line 10. If no violation is detected, `JAM.commit` is called (line 13, definition not shown) to commit the effects of write actions and instrument higher-order scripts with the `JAM.commitAction` method (defined in Figure ??).

Both the `JAM` library (Figure **??**) and the `policy` object (Figure 4) are initialized by an anonymous function with local variables and nested functions that reference these variables. The nested functions become methods of the returned enforcement object, and JavaScript's lexical scoping ensures that the private state is accessible only from within these trusted methods.

We must also ensure that the methods of the `JAM` library cannot be reassigned by untrusted code as a means of subverting the enforcement. Therefore, we define the `JAM` property of the global object with the `Object.defineProperty` method in Figure **??** to set the `writable` and `configurable` attributes to `false`, and use `Object.freeze` to render all properties immutable (line 44). JavaScript also allows variable shadowing, which is another potential way of subverting the instrumentation: a malicious guest program can declare its own `JAM` variable whose value overrides, or "shadows," the instrumentation itself. This possibility is addressed by a simple static rewriting step (not shown) that renames declared variables that could shadow the global `JAM` identifier.

The instrumentation-protection methodology described above requires that the enforcement code be initialized before the execution of any untrusted JavaScript. This requirement is easily met in the browser context by including the scripts that define the `policy` and `JAM` objects prior to any others.

3.4 Statement Indirection in JAMScript

The problem of applying a security-enforcement mechanism to code that is interpreted by the `eval` function or `Function` constructor has led many security researchers to limit their tools to language subsets [**?**]. However, surveys have shown that higher-order scripts are commonly used in existing JavaScript programs [**?**]. Indeed, the browser context provides many additional methods of generating dynamic scripts, such as the `setTimeout` function and assignment to the `HTMLElement.innerHTML` property. In this section, we describe statement indirection, a program transformation that propagates transactional introspection to scripts generated at runtime. We focus on indirection applied to invocations of `eval`, but the technique is also applicable to function calls and property writes that target the higher-order scripting mechanisms mentioned above, among others.

```
1 var ret = obj.meth(arg);
```

(a) A statically-indeterminate callsite in the original program.

```
1 var ret = JAM.call(obj.meth, obj, [arg]);
```

(b) Transformed callsite after statement indirection is applied.

Figure 6: Example of statement indirection to extend the policy to dynamically generated code. If `obj.meth` references a function that can generate code dynamically, such as `eval`, the `JAM.call` method (lines 26–40 in Figure ??) applies the security policy at runtime.

Consider the callsite in Figure 5(a). If static analysis cannot exclude that `obj.meth` references the `eval` function, we must consider the possibility that the value of `arg` may be executed as code. The statement-indirection transformation passes the function, receiver, and arguments for each relevant callsite to the `JAM.call` method, which implements Algorithm **??**. (Rather than using a `type` parameter, as in line 1 of Figure **??**, the expression type is encoded into the method that is called: we also implement analogous `JAM.new` and `JAM.set` methods for the corresponding types.) The call produces the same effects as the original method-invocation expression, except that it is instrumented to monitor the policy state and prevent violations.

The core JavaScript language defines the `call`, `apply`, and `bind` methods of function objects, which allow indirect invocation of functions. `JAM.call` must also detect and handle these constructs in case they are used to invoke higher-order scripts indirectly. When one of these methods is detected at runtime within `JAM.call`, we rearrange the arguments in a way that maintains correct semantics, and recursively call `JAM.call`. (Lines 27–31 of Figure **??** show how `Function.prototype.call` is handled.) This recursive approach protects against (presumably malicious) attempts to obfuscate a call to a higher-order script within multiple nested invocations of `call` and `apply`. Additionally, the `defineProperty`, `__defineGetter__`, and `__defineSetter__` methods of the `Object` class may be used to indirectly invoke a higher-order script that is assigned as a property getter or setter. In a typical, well-intentioned JavaScript program, this situation is likely to be very rare, but such a technique could be used by malicious code to attempt to subvert a policy. JAMScript handles the cases mentioned above along with many additional constructs provided by the Web API in the browser.

4. EXPERIMENTAL RESULTS

In this section, we present an evaluation of the performance of JAMScript on a suite of real-world applications. Our experiments were designed to answer the following questions:

- How well does transactional introspection perform when applied to real-world applications? Is the overhead acceptable?
- Does the policy-weaving approach provide performance benefits over a naive placement of transactional instrumentation?
- What are the savings to the user in terms of conceptual complexity? In particular, how does the size of a security-policy automaton compare to the generated introspection code?

To validate the correctness of our implementation, we worked with an independent team at MIT Lincoln Laboratory, who probed the system from an attacker's point of view and provided invaluable feedback.

4.1 Setup and Methodology

To answer the questions posed above, we collected a set of 27 distinct JavaScript applications, and an additional 51 sub-applications of the SMS2 DNA analysis suite (found at `http://www.bioinformatics.org/sms2/`). A list of applications is shown in Figure 8. With the goal of increasing the signal-to-noise ratio of our performance measurements, we intentionally sought applications that perform computationally intensive tasks, such as decoding a QR code or processing a DNA sequence. We developed policies ranging from simple properties, like preventing calls to `window.open`, to compound policies that reflect higher-level goals, such as preventing all external network communication. The benchmark applications and policies were paired for one of two general reasons: (i) to test the scalability and robustness of the system, and (ii) to demonstrate effectiveness of the system in preventing real exploits. A few cases (`jsqrcode`, `snote`, and `mwwidgets`) were developed and contributed by the team at MIT Lincoln Laboratory.

Application	AST Nodes	# of transaction blocks woven by Algorithm 2	# of statements rewritten by Algorithm 3
squirrelmail	110	0	0
portscanner	195	1	5
doubleclick-loader	271	0	0
userprefs	375	15	15
sunspider	407	6	2
kraken	414	8	4
beacon	787	35	33
plusone	1195	34	35
imageloader	3991	47	68
jswidgets-menu	6288	74	84
sms2-*	6656.7	317.2	321.9
snote	6852	146	139
piwik	7132	229	224
mwwidgets	7504	150	144
midori	9018	19	250
greybox	9914	346	426
googiespell	11603	63	398
ga	13236	388	398
jsqrcode	16407	361	355
hulurespawn	30269	1496	1501
colorpicker	32653	413	1048
adsense	37709	268	431
flickr	38018	1122	1139
jsbench-amazon	59192	32	6966
puzzle	104486	332	391
jsbench-facebook	112823	149	5786
jsbench-google	144453	145	9556
phylojive	166473	8542	8344

Figure 7: Catalog of the applications and policies provided as input to the analysis, and counts of the different types of instrumentation inserted. The "sms2-*" item represents the average over a set of 51 applications from the Sequence Manipulation Suite for DNA analysis.

We do not present data related to the strawman implementation described in §3.1, because it performed too poorly to serve as a reasonable point of comparison. (Instrumented code ran 3–4 orders-of-magnitude slower than the original, uninstrumented code.) Instead, we compare the performance of the applications secured by policy weaving (referred to as the "fine-grained" approach, because individual statements are speculatively executed) to that of applications secured in a whole-program fashion (referred to as the "coarse-grained" approach) in which transaction blocks are used to secure entire scripts. The latter approach approximates prior work [**? ?**] in which transaction boundaries are determined by individual `script` tags or by the browser's same-origin policy. Additionally, to test the absolute performance of transactions as an enforcement mechanism for policy weaving, we compare the running time of the secured applications to that of the original program.

The static analysis was performed by the JAM policy weaver [**?**], modified to use the rewriting technique presented in Algorithm 2 and Algorithm 3. JAMScript is implemented in the SpiderMonkey JavaScript interpreter, version 1.8.5, which is embedded in the Firefox browser, version 17.0.5esr. Experiments were run on a Dell Inspiron E6520 laptop computer with an 8-core Intel Core i5-2540M 2.60GHz CPU with 8GB RAM, running the 64-bit Ubuntu 12.04 LTS operating system.

4.2 Runtime Performance

The plot shown in Figure 9 shows that in most cases, and particularly for test cases that involve computationally-heavy processes, a program woven with fine-grained transactions outperforms the corresponding program protected by a coarse-grained strategy. This result was not a foregone conclusion, because the two approaches

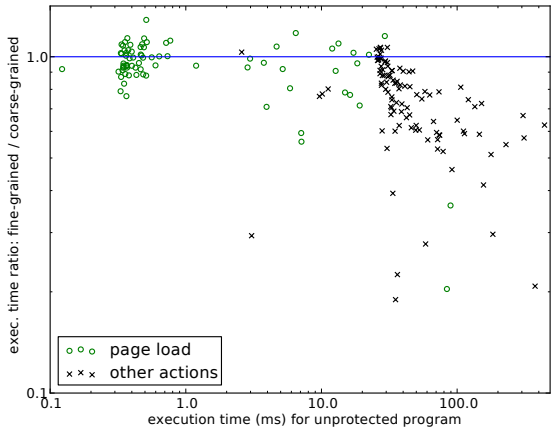


Figure 8: Log-log plot of the execution times of programs with fine-grained transactional enforcement applied through weaving compared to coarse-grained whole-program transactions. The ratio of the two approaches is plotted against the execution time of the original unprotected program. Most points are below the line $y = 1$ when the original execution time is non-trivial, which indicates an overall performance benefit from policy weaving, compared to the coarse-grained approach.

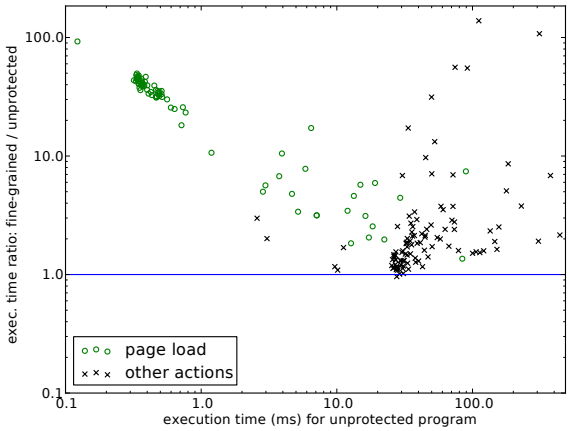


Figure 9: Log-log plot of the ratio of the execution times of programs with fine-grained transactional enforcement applied through weaving compared to unprotected execution, plotted as a function of the execution time of the original unprotected program. Page-load actions include the time to initialize the policy object and JAMScript library, which becomes less of a factor as the overall page-load time increases. The overall trend decreasing to the right indicates that the percentage overhead of security-policy enforcement is less dramatic for more computationally-intensive applications.

represent opposite ends of a performance trade-off between transaction start-up time and in-transaction processing time. A weaving strategy that uses many fine-grained transactions for policy enforcement relies on the assumption that most of the overhead of speculative execution is incurred during the course of the transaction, rather than in the initialization of the transaction. Our experiments bear out this assumption: we measured an overall 23% speed-up for program actions instrumented with fine-grained transactions versus coarse-grained transactions, when summarized by the geometric mean. The speed-up breaks down into a 7% speed-up for page-load actions, which includes the time taken to load the JavaScript policy object and the `JAM` library, and a 33% speed-up for other program actions. Moreover, there are no prospects for reducing the observed overhead with the coarse-grained approach, whereas policy weaving provides the opportunity to exert additional effort during static analysis to rule out spurious instrumentation to further reduce the runtime overhead.

Similarly, Figure 10 shows the ratio of the execution time for programs protected by woven transactions versus the original unprotected program, plotted as a function of the original execution time. When summarized by the geometric mean, the measured execution time for actions other than the initial page load protected by fine-grained transactions is 231% of the time for the unprotected program. (This number can be compared to 343% for coarse-grained transactions.) Loading the page took 20.8 and 22.3 times longer for the instrumented programs (fine-grained and coarse-grained, respectively), which must load the policy object and the `JAM` library prior to the program itself. However, the absolute time represented by this slowdown ranges from 8.5ms to 24.2ms, which is imperceptible to human users.

Application	Description of Applied Policy	Input Policy (nodes)	Baseline Policy (nodes)	Specialized Policy (nodes)
googiespell	disallow document.write	7	103	196
imageloader	disallow document.write	7	103	196
jsbench-facebook	disallow XMLHttpRequest.open	11	103	196
sunspider	disallow XMLHttpRequest.open	11	103	196
squirrelmail	disallow access to src property	11	103	105
jsbench-amazon	disallow XMLHttpRequest.open	11	103	196
jsbench-google	disallow XMLHttpRequest.open	11	103	196
kraken	disallow XMLHttpRequest.open	11	103	196
puzzle	prevent creation of script elements	13	143	256
hulurespawn	disallow local storage access	13	146	331
greybox	prevent creation of script elements	13	143	256
jswidgets-menu	prevent creation of script elements	13	143	256
midori	prevent modification of cookie	13	139	330
flickr	prevent external pop-up creation	16	135	240
portscanner	disallow setting src property	18	126	226
doubleclick-loader	prevent navigation	18	128	130
phylojive	isolate document from cookie	19	162	356
piwik	isolate document from cookie	19	162	356
beacon	isolate document from cookie	19	162	356
userprefs	disallow appendChild and eval	22	121	229
ga	prevent script creation, document edits	24	140	356
plusone	prevent script creation, document edits	24	140	356
adsense	isolate document from cookie	29	163	690
colorpicker	prevent navigation, src/cookie access	33	151	574
snote	certain elements write-once/read-only	92	285	1329
mwwidgets	certain elements write-once/read-only	92	285	1247
sms2-*	prevent all network communication	457	953	1981.9
jsqrcode	prevent all network communication	457	953	2566

Figure 10: Size of the input policy versus the automatically generated enforcement code, given by abstract-syntax-tree nodes. "Input policy" refers to the automaton provided to the static analysis (see Figure 1). The "baseline policy" is a single introspector that checks all policy transitions (e.g., `policyFull` in Figure ??) needed for coarse-grained protection of the full program. The "specialized policy" represents introspection code produced for fine-grained protection of the woven program (e.g., all of the `policy*` functions in Figure ??), and is generally larger because multiple introspector functions are needed to evaluate different combinations of policy predicates.

4.3 Policy Complexity

A substantial benefit of the automated policy-weaving approach to program security is that it permits policies to be specified declaratively. The rewriting framework converts the declarative policy into imperative code that makes use of the introspection capabilities of JAMScript to enforce the policy at runtime. An example of this translation appears in the relationship between the input policy in Figure 1 and the `policy*` functions in Figure **??**. These functions constitute a "specialized policy" in which each introspector evaluates a unique combination of policy transitions to correspond to the fine-grained instrumentation placed by policy weaving. The `policyFull` function in Figure **??** is an example of a "baseline policy" that monitors all transitions of the input policy and can be used with a coarse-grained transaction strategy. The policy weaver that we have integrated with JAMScript provides a language for specifying a policy automaton as a list of edges labeled by predicates.

Each predicate is written in a dialect of JavaScript that can reference calls to native functions (and corresponding arguments), reads and writes to properties of native objects, reference (in)equality, and string matching with regular expressions.

To quantify the benefits provided by the automated production of introspection code, Figure 11 compares the size of the input policy to the size of the generated code. Summarized by the geometric mean, the code for the baseline policy is 7.4x larger than the input policy for our benchmarks, and the code for the specialized policy is 15.8x larger. The size of the representation is admittedly only an indirect measure of implementation complexity, but we also believe that the input policy automaton provides an intuitive interface for specifying a set of disallowed execution traces. These factors together provide strong support for the usability of the policy-weaving approach to security-policy enforcement.

5. RELATED WORK

Aspect Weaving and Runtime Verification. Runtime verification systems [**?** **?** **?**] monitor safety properties over execution traces produced by an instrumented runtime environment. Runtime enforcement is additionally charged with preventing violations of safety properties during the execution. We have described a formulation of runtime enforcement suited for securing untrusted programs written in interpreted languages that provide a mechanism for evaluating dynamically generated code.

Oftentimes, aspect weaving is the methodology used to place the instrumentation that a runtime-verification system uses. This approach may be limited to interposition on predefined threshold events, such as system calls or library method calls [**?** **?** **?**]. In contrast, our work pursues a more invasive but powerful technique: policy weaving through program rewriting. This approach enables enforcement at a finer granularity (for example, individual property reads and writes). Policy weaving also departs from standard aspect weaving by enabling semantics-based—rather than syntax-based—rewriting.

Runtime Enforcement for JavaScript. A number of prior works investigate the enforcement of security policies for JavaScript in a browser. In general, the solution for runtime enforcement that we describe in this paper is distinguished from prior work by the choice of transactional introspection as the enforcement mechanism, and the resulting ability to handle safely the full JavaScript language, rather than a restricted subset.

Fredrikson et al. [**?**] describe the JAM policy weaver for JavaScript as an implementation of their general technique. That paper focuses primarily on the static-rewriting methodology, and leaves the nature of the runtime-enforcement mechanism largely unspecified. In a complementary manner, we investigate the desirable properties of the runtime component, while allowing for the possibility of integrating a wide range of static-analysis techniques.

Yu et al. [**?**] present a rewriting scheme that uses indirection to apply security policies to dynamically generated JavaScript code. Their method actively rewrites this code as it is encountered at runtime. In contrast, we avoid structural manipulation of code at runtime by applying transactional introspection to the generated code as it is encountered. Their system is built entirely in JavaScript without interpreter modifications, and they express concern that this approach can allow a malicious script to alter the instrumentation during execution. While we implement much of the functionality of transactional introspection in the interpreter, we also introduce JavaScript code that must be protected. Our method of doing so, based on lexical closures, could be employed in their implementation.

Several systems, including Caja [**?**], ADsafe [**?**], ConScript [**?**], and Maffeis et al. [**?**], use wrappers or object-capability models (often in addition to rewriting) as a means of enforcing policies on untrusted JavaScript code. In these schemes, references to native objects are redirected to secure implementations that can filter unsafe runtime behaviors. Wrappers and statement indirection both enable intervention on security-sensitive actions, but statement indirection is a rewriting technique and wrappers are an alteration to the runtime environment. Therefore, indirection can be used at a per-statement granularity, while wrappers are typically employed per-module. For this reason, statement indirection is a more natural target for policy weaving, which aims to apply instrumentation precisely where it is needed. Each of the wrapper-based approaches limits the analyzed programs to some subset of the JavaScript language, often to prohibit higher-order scripts, thereby sacrificing transparency. Transactional introspection in JAMScript is crucial to provide complete enforcement for dynamically generated code.

Transaction-Based Policy Enforcement. The use of transactional introspection for security-policy enforcement can be viewed as an evolution of inlined reference monitoring, which was developed around 2000 by Schneider and Erlingsson [**?** **?** **?**]. The primary difference is that the semantics of introspection enables a more direct examination of the effects of the monitored program statements, rather than relying on a calculation of the effects. When higher-order scripts (or just complicated statements) are used in the analyzed program, this ability to observe concrete effects is needed for sound and complete policy enforcement.

Transcript [**?**] is an extension to the JavaScript language that implements speculative execution with introspection for security-policy enforcement. As presented in [**?**], each untrusted "guest" module, delineated as different source-code files included in a web page via HTML `script` tags, is executed within a transaction. In contrast, JAMScript was developed to be the target of a static weaving process that results in fine-grained transactions applied to the full program, including the host code. Also, while Transcript relies on a complex conflict-resolution system to maintain the consistency of the DOM during a transaction, JAMScript uses a relatively simple suspend mechanism that is oblivious to the context in which the core JavaScript interpreter is running.

Richards et al. [**?**] apply speculative-execution semantics (referred to as "delimited histories") to untrusted code in an automatic but coarse-grained fashion based on the browser's same-origin policy. In their approach, the entirety of the third-party code is executed speculatively, and the host code is trusted to execute without protection. In contrast, we argue for an approach in which *all* code is subjected to the security policy, and the scope of instrumentation is focused by using the results of a conservative static analysis. Trusting the host code may be reasonable in some contexts, but this assumption leaves open the possibility for clever attackers to manipulate the environment and coerce the host code into violating the intended policy. Another difference between their work and ours is that the introspection code in [**?**] is written manually on a case-by-case (albeit reusable) basis in C++ modules, with the intent of shielding the instrumentation from manipulation. In contrast, our system automatically produces introspection code as a translation of the policy automaton, and we make use of a relatively simple scheme for protecting the integrity of the instrumentation.

Various systems use transactional introspection for policy enforcement in other contexts. Transactional Memory Introspection (TMI) [**?**] applies transactional instrumentation to multithreaded server software. The JavaScript language does not exhibit true multithreaded behavior, so TMI addresses issues that are orthogonal to the ones addressed in this paper. TxBox [**?**] uses transactions to enforce operating-system-level policies that limit the access of applications to system resources. In contrast, JAMScript enforces application-level policies and can restrict internal program behavior. Our approach is also distinguished by the use of static analysis

for runtime-performance benefits, as well as by the techniques we use for securing dynamically generated scripts.