Efficient Runtime Enforcement Techniques for Policy Weaving

Richard Joiner†, Thomas Reps‡, Somesh Jha‡, Mohan Dhawan§, Vinod Ganapathy∗

†University of Wisconsin-Madison  ‡GrammaTech, Inc.  §IBM Research New Delhi  ∗Rutgers University

{joiner, reps, jha}@cs.wisc.edu, mohan.dhawan@in.ibm.com, vinodg@cs.rutgers.edu

Abstract

Policy weaving is a program-transformation method 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 power and flexibility of policy weaving arises from its ability to blend the best aspects of the static and runtime components. Therefore, a successful instantiation requires careful balance and coordination between the two.

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

• Transaction-based introspection allows the state resulting from the execution of a statement to be examined and, if the policy would be violated, suppressed.
• Callsite 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 transaction-based introspection and callsite indirection for policy weaving, and report experimental results that show the viability of the approach in the context of real-world JavaScript programs running in a browser.

1. Introduction

Policy weaving for security-policy enforcement is a program-rewriting approach oriented towards striking a balance between static and runtime analysis techniques [7, 8]. It is

• Sound, meaning that all program traces that violate the policy are prevented.
• Transparent, meaning that the rewritten program has the same semantics as the original program, modulo policy violations.

The policy-weaving approach is motivated by the acknowledgment that in nontrivial scenarios, static analysis alone cannot prove that a program adheres to a policy [4, 10]. In contrast, runtime analyses such as inlined reference monitoring (IRM) [6, 15, 17] can soundly evaluate concrete states at runtime, but at the cost of degraded performance. 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 gives rise to 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 explore and demonstrate the utility of transactional introspection applied to a program through static rewriting of source code, and an additional callsite-indirection transformation to enable just-in-time sandboxing 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.

We find that transactions provides a straightforward and powerful tool for securing untrusted code in the manner required by policy weaving. Transactions are a form of speculative execution, meaning that the effects of an execution can be computed and examined prior to the application of those effects to the execution environment. This 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. Because rollback of communication and other I/O actions is not possible in general, a security-policy-enforcement mechanism that aims to mediate such actions needs the ability to recognize and potentially suppress such events prior to their occurrence, rather than reacting after the actions have occurred. The transactions we describe provide this capability to “peek into the future” and take preventive action to avoid policy violations.

We also extend the methodology of policy weaving to incorporate the use of callsite indirection to dynamically apply transactional semantics to code that is dynamically generated. This strategy serves two purposes.

• The static analysis is freed from the generally impossible task of precisely modeling all possible executions of dynamically-generated code.
• Callsite indirection serves as a light-weight but special-purpose alternative to the use of hard-coded transactions.
Transactional-introspection schemes have been studied in other papers in the security literature [2, 5, 9, 14], and various difficulties have been identified:

- Performance overhead of speculative execution can be prohibitive [3, 11].
- Correct placement of transactional instrumentation is a nontrivial task [9, 14, 16].
- Implementation of introspection logic to recognize and prevent policy violations can be error-prone [14].

We address each of these concerns via the formulation described in this paper. The complexity of both manually placing individual transactions and constructing introspection code is replaced by the requirement to formulate an explicit security policy as an automaton. A security-policy automaton allows a programmer to state his intentions and goals explicitly in a typically small and self-contained artifact. Moreover, we find that the use of a transactional paradigm serves as a natural and intuitive runtime enforcement platform for policy weaving, and produces substantial benefits in terms of performance and flexibility when compared to other enforcement mechanisms.

To summarize, our contributions are:

- We show how a policy-weaving algorithm can be used to automatically and soundly weave transactional instrumentation into a program to enforce security invariants.
- 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 callsite indirection, a program-transformation technique that, in conjunction with transactions applied directly by policy weaving, ensures that the policy is also applied to all dynamically generated code.
- We present JAMScript, a general-purpose but simple 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 to other types of enforcement mechanisms that could be targeted by a policy-weaving algorithm.

**Organization:** §2 presents a specification of the program-rewriting algorithm, the semantics of transactional introspection, the mechanism of callsite indirection, and how these techniques conceptually fit together. §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 motivates various design choices that differ from other systems that share the same goals. §6 concludes.

### 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 callsite indirection as two key primitives of an enforcement strategy that meet these conditions. We then incorporate this combined 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.

Figure 1: Security-policy automaton and textual representation specifying a set of disallowed traces: “read a password value and subsequently make a network request.” The identifier F is reserved for the final state.

- The mechanism should precisely evaluate the policy at runtime, without spuriously blocking traces.
- The mechanism should be general-purpose, flexible, and straightforward to apply.

Because the latter two concerns are directly at odds with each other, and both may indirectly hinder the first goal of acceptable runtime performance, it may be difficult or impossible to arrive at an ideal solution that satisfies all three principles simultaneously. Our thesis is that utilizing a policy-weaving strategy can, in effect, bring the enforcement mechanism closer to achieving each of these principles, compared to manually placing instrumentation or relying on an automated modular partitioning strategy as suggested in other work [5, 14].

When used with an enforcement mechanism based on transactional introspection, as advocated in this paper, 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 described 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.)

We now introduce an example program and a security policy that we would like to enforce via transactional introspection and callsite indirection. The policy shown in Fig. 1 asserts that network communication should not occur after certain kinds of accesses to local information. The program shown in Fig. 2(a) may have behaviors that violate the policy. The use of a conservative static analysis allows the rewritten program, shown in Fig. 2(b), to have the following properties:

- Any statement that potentially affects the policy state is enclosed in a transaction block, indicated by the keyword introspect.
- An introspector function is passed as the parameter to each transaction block; these functions are defined in Fig. 3(a).
- Each introspector implements the runtime evaluation of 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:

- During execution of the transaction block, all actions (reads, writes, and calls) are recorded sequentially, and the effects of writes are postponed.
- When the closing brace of the transaction block is reached, the introspector is invoked with a transaction-object argu-
SECRET_PASSWORD = "supersecret";
user_pwd = "";
cfg = {};

fun getUserInput(prompt) {
    stdout.write(prompt + " ");
    input = stdin.readLine();
    return input;
}

fun getConfig() {
    config = ";
    return input;
}

fun indirect(f, args) {
    f = indirect(eval, [configString]);
    f();
}

while (true) {
    opt = getUserInput("Enter option");
    if (opt == "config") {
        getConfig();
    } else if (opt == "run") {
        user_pwd = getUserInput("Enter password");
        if (match) {
            break;
        } else {
            print("Wrong password!");
        }
    } else if (opt == "quit") {
        exit;
    }
}

// The rest of the program...

(a) Original code for a program intended to read configuration data from an untrusted server and subsequently perform some computation guarded by a secret password. It is vulnerable to code injection by the untrusted server and accidental password leakage by the user.

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

(b) Secured code. The gray highlighting indicates code that was introduced by the rewriting step. Some unchanged sections are elided to save space. introspect is a control-flow keyword parameterized by a function that contains the introspection logic. indirect is a function that wraps the configString argument within an introspect block, and returns a reference to the eval function bound to the instrumented argument. Fig. 3 shows the implementation of the introspector functions and Fig. 3(b), shows indirect.

(a) Functions that perform introspection on the actions that occur within an introspect block. The parameter tx is an object that holds information about the reads, writes, and calls that have been recorded during transaction execution. A collection of policy states reached during the current global execution is maintained in the policyState list, and an exception is thrown when a policy violation is detected.

(b) Implementation of the indirect function that (i) wraps dynamically-generated code within an introspect block as needed, and (ii) returns the input function bound to its arguments. indirect is a general function constructed to handle the interpretive constructs of a language, independent of the specific program being analyzed. The policyTransitionAll function is also shown; it combines the logic from the two functions shown in (a), and serves as the introspector for the instrumentation that is constructed by indirect.

Figure 3: Instrumentation functions that monitor program actions at runtime. See Fig. 2 for their corresponding uses within the analyzed program. We show later in §3 an example of how the references to these function—and the global references used within the functions—can be protected from manipulation by the program being analyzed.
1. \textbf{Introspect}(T, Actions, Reached):
   \textbf{Data}: T: 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
   \textbf{Result}: Terminate execution if a policy violation is detected, or update
   reached and commit each a ∈ Actions otherwise.

\begin{verbatim}
/* Examine each action in the order it was recorded */
2 for each a ∈ Actions do
   3 foreach (φ₁, φ₂) ∈ T do
      4 /* If the policy-transition prestate has been reached and the poststate has not */
      5 if φ₁ ∈ Reached \land φ₂ \notin Reached then
         6 /* If the policy-transition predicate holds */
         7 if a ⊨ φ then
            8 /* If the poststate is final */
            9 if φ₂ is final then
               10 Throw an exception and quit.
               11 else
               12 Reach ← Reached ∪ \{φ₂\}
               13 if a can affect the program state then
               14 Commit a.
\end{verbatim}

\textbf{Algorithm 1:} Introspect specifies the logic that evaluates an action sequence, and either commits or suppresses the effects. The condition \(a \models \varphi\) is satisfied if action \(a\) makes predicate \(\varphi\) true.

1. \textbf{RewriteIntrospect}(P, \Psi_φ): 
   \textbf{Data}: P: source code of the program being analyzed
   \Psi_φ = \{(σ_0, τ_0), \ldots, (σ_n, τ_n)\}: a policy-violating witness
   \textbf{Result}: \(P'\): a rewritten program that prevents \(\Psi_φ\).

2. Let \textit{interp} be a policy-transition symbol indicating an interpretive construct
3. foreach \((s_i, τ_i) \in \Psi_φ\) do
4. if \(τ_i = \textit{interp}\) then
5. Add \(s_i\) to a set of callee-state statements to be processed later by Alg. 3
6 else
7. Let \(T_i\) be the set of policy transitions for which \(s_i\) is already
   instrumented
8. \(T_{prev} ← T_i\)
9. \(T_i ← T_{prev} \cup \{τ_i\}\)
10. Generate or retrieve \(\text{insp}_i\), the introspector function that
   implements Alg. 1 for \(τ_i \in T_i\)
11. if \(T_{prev} = \emptyset\) then
12. Enclose \(s_i\) in a transaction with introspector \(\text{insp}_i\):
13. \(s'_i : \text{introspect}(\text{insp}_i)(s_i)\)
14. else
15. Retrieve \(\text{insp}_{prev}\), the introspector function that implements
   Alg. 1 for all \(τ \in T_{prev}\)
16. \text{Rewrite} \(s_i : \text{introspect}(\text{insp}_{prev})(s_{insp})\):
17. \(s'_i : \text{introspect}(\text{insp}_i)(s_{insp})\)

\textbf{Algorithm 2:} RewriteIntrospect specifies the static-rewriting step of a
policy-weaving algorithm for applying direct transactional introspection.

The observations above allow the rewriting step of Safety-
Weave to be transparently integrated with a transactional enforce-
ment mechanism. However, despite their similar utility, the imple-
mentations 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 strongest-postcondition opera-
tor that we describe in this paper is developed as a built-in feature
of the interpreter, and it produces a single program state in which
\(\varphi_{\text{viol}}\) is evaluated.

A policy-weaving algorithm may utilize arbitrary (conservative)
static-analysis techniques to investigate a program’s behavior with
respect to a security policy \(\Phi\). However, in a departure from tradi-
tional static verification methodology, policy weaving uses a rewriting
step, which serves as the interface between the static and run-
time 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 re-
  source 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 con-
crete program, as determined via symbolic execution, and an “in-
valid” trace is one that cannot occur in the concrete program, but
is extracted from the abstract model of the program due to impre-
cision. 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
allowing predicates constituting the policy to be directly eval-
uated in the context of the (speculative) program state. Viewed an-
other way, a code fragment that consists of a transaction block con-
taining a statement \(s\) implements the strongest-postcondition op-
erator \(post(\sigma_{\text{pre}}, s) = \sigma_{\text{post}}\) that maps the program state \(\sigma_{\text{pre}}\)
evacuated at transaction entry and the enclosed statement \(s\) to the
poststate \(\sigma_{\text{post}}\) that results from the execution of \(s\). A policy
predicate \(\varphi_{\text{viol}}\) can then be directly evaluated in this poststate.
Formally, the condition for policy violation is \((post(\sigma_{\text{pre}}, s)) \cap
[\varphi_{\text{viol}}]) \neq \emptyset\).

In the direct rewriting of transaction blocks described above, we define a second type of rewriting, called callsite indi-
rection, by which any callsites that potentially invoke \textit{interpretive}
constructs are transformed in a way that allows transactional intro-
spection to be performed selectively at runtime. Interpretive con-
structs are those language facilities that convert a string value into
dynamically generated code fragment, which is either immedi-
ately executed or can be executed at a later time. Examples of inter-
pretive constructs abound in modern scripting languages, such as
JavaScript (\textit{eval}, \textit{Function}), Python (\textit{eval}, \textit{exec}, \textit{compile}), and Perl (\textit{eval}).

Delaying the decision to perform full introspection on these
callsites serves as an important optimization in languages (includ-
ing each of those previously mentioned) in which functions can be
invoked by indirect reference. In moderately large systems written
in these languages, callgraph construction becomes quite imprecise
when performed conservatively. I.e., the targeted function may be
statically unconstrained for many callsites. The strategy of callsite
indirection serves to avoid enclosing the preponderance of all call-
sites in transaction blocks in favor of a runtime check to determine
whether a given call needs to be evaluated with transactional intro-
spection.

2.2 A Formalization of Rewriting

The rewriting that occurs between Fig. 2(a) and Fig. 2(b) is
accomplished via a modified and extended version of the rewriting
step of the SafetyWeave algorithm formalized by Fredriksen et al.
[7]. 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 \(\text{pre}(s, \varphi_{\text{viol}})\)
of the subsequent statement \(s\) and policy predicate \(\varphi_{\text{viol}}\) with
respect to the current program state \(\sigma_{\pre}\), and terminates the pro-
gram if \(\text{pre}(s, \varphi_{\text{viol}})(\sigma_{\pre}) = \text{true}\) (i.e., the execution of
the statement could cause a policy violation).

A key insight that led us to integrate SafetyWeave with a trans-
actional enforcement mechanism is that the semantics of introspec-

RewriteIndirect($P$, $s$):
Data: $P$: source code of the program being analyzed
$s$: program statement containing a callsite
Result: $P'$: rewritten program that contains code that is dynamically generated by $s$ over all policy transitions
Let $\mathcal{T}$ be the set of policy transitions for which $s$ is instrumented with an introspect block
1. if $\mathcal{T} = \emptyset$ then
2. $\text{Rewrite } s: \text{ret} = c($args$...)$;
3. $s'_1: f = \text{indirect}(c, [\text{args}...])$;
4. $s'_2: \text{ret} = f()$;
5. else
6. $\text{Retrieve } \text{insp}, \text{the introspector function that implements Alg. 1 for all } \tau \in \mathcal{T}$
7. $\text{Rewrite } s: \text{introspect}(\text{insp})[\text{ret} = c(\text{args}...)]$;
8. $s'_1: \text{introspect}(\text{insp})[v0 = c; v1 = [\text{args}...]]$;
9. $s'_2: f = \text{indirect}(v0, v1)$;
10. $s'_3: \text{introspect}(\text{insp})[\text{ret} = f()]$

Algorithm 3: RewriteIndirect specifies the rewriting step done as a postprocessing step of a policy-weaving algorithm. It applies the callsite-indirection transformation to a statement that may target interpretive constructs.

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 [7]. We now develop a precise specification of the rewriting step that facilitates policy enforcement via transactional introspection (see Alg. 2). We also specify a final rewriting step, which occurs after all invocations of Alg. 2) and applies callsite indirection to all statements that can target interpretive constructs (see Alg. 3).

In Alg. 2, the “policy-violating witness” $\Psi_\phi$ represents an execution trace along which the policy is violated. Each element $(s_i, \tau_j)$ of $\Psi_\phi$ consists of a program statement $s_i$ and a policy transition $\tau_j$. A policy transition may be a special symbol $\text{interp}$, which indicates that $s_i$ contains a callsite that potentially targets an interpretive construct; these statements are merely collected to be processed later by Alg. 3. Otherwise, $\tau_j$ is a triple $(\varphi_{\text{pre}}, \varphi, \varphi_{\text{post}})$ of a policy state $\varphi_{\text{pre}}$, a policy state $\varphi_{\text{post}}$, and a predicate $\varphi$ that induces the transition. Together, Alg. 2 and Alg. 3 rewrite program $P$ so that every witness $\Psi_\phi$ is prevented in the resulting program $P'$.

Informally, given a policy-violating execution trace that is realizable in the conservative, abstract model of the program, the program is rewritten to include transactional introspection of each statement $s$ in the trace that can induce a policy transition $\tau$. As an alternative to hard-coding introspect blocks to monitor callsites that target interpretive constructs, callsite indirection is used to apply transactional introspection only when needed.

In contrast to the rewriting step specified in past work on policy weaving [7], which requires an implementation of the pre-image operator to determine when $s$ will cause a policy transition, Alg. 2 and Alg. 3 rely on the inherent properties of transactional introspection to effectively apply the dynamic implementation of the strongest postcondition operator as specified in Alg. 1.

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 introduce the concept of 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 partially rather than as an atomic unit.

The introduction of transaction suspension does not require alteration of Alg. 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 do not need to trigger a suspension—can be maintained to reduce overhead. We discuss specific scenarios in which transaction suspension has utility 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 Semantic Guarantees

The precise specification of the semantics of an introspector function given above is important for maintaining predictable semantics of the rewritten program, and therefore not invalidating the results previously given by the static analysis. The transactional introspection implemented by Alg. 1 and woven by Alg. 2 is guaranteed, up to policy violation by the enclosed code, to (i) have no visible side-effects, and (ii) maintain sequential execution. “Sequential execution” means that the execution of a transaction block will always produce the same results as a prefix of the original unprotected code.

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.

3.1 A Strawman Approach

An initial attempt at implementing 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 POSIX 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 3–4 orders of magnitude slower than standard (unprotected) execution.

Our experience with ForkIsolate motivated the requirement that an enforcement framework be general with regards to the code that it can evaluate. The core JavaScript language provides methods for executing dynamically-generated code, most notoriously through the use of the $\text{eval}$ function. This fact necessitates that the speculative-execution framework be able to maintain faithfully the semantics of all possible executions (in the absence of policy violations). 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, resulted in unexpected behavior, including freezes and
crashes. Because of the architecture of ForkIsolate, these dangerous cases had to be recognized and handled within the introspection mechanism itself. Consequently, converting ForkIsolate from a proof-of-concept experiment into a general-purpose tool would have required special logic for a practically unbounded number of scenarios induced by (i) JavaScript’s tendency to be modified or bypassed. In JAMScript, we leverage ForkIsolate from being modified or bypassed. In JAMScript, we leverage Java-

Moreover, the embedded character of a typical JavaScript environment motivated the requirement that introspection capabilities be extensible. Even if the ForkIsolate mechanism were able to maintain correct semantics of arbitrary code, additional logic would still be needed to achieve sound introspection for all executions. For example, if the speculative execution of an expression results in a call to the document.write DOM method, the goal of comprehensive policy enforcement dictates that the contents of any script elements within the generated HTML should be speculatively executed as well; to perform these speculative executions, it is necessary to have a way of identifying and extracting such code during introspection.

In contrast, a framework like JAMScript, which allows specialized enforcement capabilities to be loaded as necessary for each embedding system (such as the DOM), is more general and maintainable than a system with built-in domain-specific logic.

3.2 Description of JAMScript

With JAMScript, we adopt the paradigm of 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 introspection logic.

Additional utilities for runtime enforcement, such as the implementation of the indirect function described in §2, are provided as part of the JAMScript library. This code is written in JavaScript and is loaded prior to the rewritten program, and after the policy logic that is generated at rewriting time by Alg. 2. Because the JAMScript 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. 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.

As previously mentioned, interaction with all interfaces 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. The upshot of this approach is that external systems that wish to interact with 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 enables transactional semantics of the JavaScript interpreter that are not entangled with the semantics of ancillary systems.

3.3 Protecting the Instrumentation

An overarching concern when a new runtime security technique is developed is that the instrumentation itself must be protected from being modified or bypassed. In JAMScript, we leverage JavaScript’s implementation of lexical scoping and closures to create

```
var policy = (function () {
    // A record of reached policy states
    var policyStates = [0];

    // Close over native objects for reliable references
    var HTMLElement = HTMLElement;

    // Introspector preventing access to the "appendChild" method
    // of objects with a prototype chain that includes HTMLElement
    function introspectReads(tx) {
        var commit = true;
        var rs = tx.getReadSequence();
        for (var i=0; i<rs.length; i++) {
            var node = rs[i];
            if (node.id === "appendChild" && node.obj instanceof HTMLElement) {
                commit = false;
                break;
            }
        }
        if (commit) {
            JAMScript.commit();
        } else {
            JAMScript.prevent();
        }
    }

    // Return the policy object itself
    return {
        introspectors: {
            introspectReads: introspectReads
        }
    };

    Object.freeze(policy);
```

Figure 4: Generating the policy object as a self-enclosed entity. The local HTMLElement variable is used to save a private reference to an object that is needed for policy evaluation. Function introspectReads recognizes when the appendChild property of an HTMLElement object is read, and prevents the read from occurring.
```
1 var ret = obj.meth(arg);

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

2 var bound = JAMScript.bind(obj, [obj.meth, arg]);
3 var ret = bound();

(b) Transformed callsite that enables the call to be examined at runtime, at which point the policy can be applied.

Figure 5: Example of callsite indirection to extend the policy to dynamically-generated code. If obj.meth references a function that can generate code dynamically, such as eval or Function, the use of the JAMScript.bind method allows the security policy to be applied dynamically (see Fig. 6).

names declared variables that would shadow the JAMScript instrumentation. Because the global JAMScript object is the entry point for all enforcement code, this identifier is the only one of interest for this step, although the technique could easily be applied to multiple global-variable names.

3.4 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 [1]. However, surveys have shown that interpretive constructs are commonly used in existing JavaScript programs found on the Web [13]. In this section, we explain a program transformation that enables the safe, sound, and efficient analysis of dynamically-generated code at runtime.

The core JavaScript language provides two ways to generate code dynamically. The eval function executes a string passed as its first argument, and the Function constructor coerces a string argument to code that is used as the body of a function that can be executed later. External systems such as the DOM expose other routes to inject code into the execution stream. The method that we describe below applies to many of these cases as well, but we will focus on the application to eval and Function.

The key observation in developing the statement transformation described below is that any code that is to be generated dynamically will be passed as a string-valued argument to some function call. In both cases examined here, the string to be executed is the first argument passed to either eval or Function. We therefore implement a transformation on all callsites for which the target of invocation cannot be determined statically (or at least for which eval and Function cannot be ruled out as targets). Consider the callsite in Fig. 5(a). If it cannot be excluded via static analysis that the reference obj.meth points to an interpretive construct, we must consider the possibility that the value of arg may be executed as code.

The transformation passes the function, receiver, and arguments for each affected callsite to the JAMScript.bind method, which (i) examines the function reference to determine if it is an interpretive construct, (ii) wraps any string argument that will be interpreted as code in a transaction block, and (iii) returns a new function that is bound (using JavaScript’s bind method) to its receiver and new arguments. This function is then invoked to produce the same effect as the original method-invocation expression, except that it is instrumented to monitor the policy state and prevent violations.

The core JavaScript language defines a few other constructs that must be detected and handled within the body of JAMScript.bind to interpose on all cases of dynamically generated code; these cases involve the ways in which interpretive constructs may be invoked indirectly. One such avenue for invoking a function indirectly in JavaScript is to use the call, apply, or bind methods of function objects. When an invocation of any of these methods is detected at runtime within JAMScript.bind, we make use of the
```

same logic described above for sanitizing interpretive constructs that may be the receiver of the original invocation—i.e., by rearranging the arguments and recursively calling JAMScript.bind (see Fig. 6). This recursive approach also protects against (presumably malicious) attempts to obfuscate a call to an interpretive construct within multiple nested invocations of call and apply. Additionally, the defineProperty, _defineGetter_, and _defineSetter_ methods of the Object class are of interest if the getter/setter being assigned is an interpretive construct. In a typical, well-intentioned JavaScript program, this is likely a very rare practice, but such a technique could be used by malicious code to attempt to subvert a policy.

3.5 Extending the Callsite Indirection Transformation

It is useful for the JAMScript enforcement mechanism to support a policy-specification language that includes “call” predicates, which induce a policy transition when particular native functions are invoked. Due to the inherent imprecision when static analysis is applied to a language with first-class functions, it is common for several callsites that target user-defined functions at runtime to be instrumented with transaction blocks. During testing, we found that this situation can cause substantial performance degradation due to an oftentimes deep nesting of transactions. Concretely, if a call to a user-defined function is enclosed in a transaction block,
Figure 7: Example of the callsite transformation that serves to avoid transaction nesting at runtime and to extend the policy to dynamically generated code.

```javascript
1 function introspectCalls(tx) {
2   var commit = true;
3   var cs = tx.getCallSequence();
4   for (var i=0; i<cs.length; i++) {
5       var node = cs[i];
6       if (node.value === _createElement && node.args[0] === "script") {
7           commit = false;
8           break;
9       }
10   }
11   // ... commit or prevent.
12 }
```

(b) Introspector code to evaluate the original call predicate.

and that function contains other callsites enclosed in transaction blocks, arbitrary nesting of transactions can occur. To address this problem, we rely on an extension of the callsite transformation described above, along with an associated automated refactoring of the policy-introspection code (see Fig. 7 for an example). The version is structured so that the assignments to the identifiers v1 and v2 correspond to (i) the invoked function value, and (ii) an array containing the receiver and arguments, and the introspector is restructured accordingly. In effect, this converts a “call” policy predicate into an equivalent “write” predicate. Consequently, no callsites are directly enclosed within introspect blocks and nesting is prevented, while, simultaneously, the policy can be applied to dynamically generated code.

4. Experimental Results

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

- How well does transaction-based introspection perform when applied to real-world applications? Is the overhead acceptable?
- Does the policy-weaving approach to transaction placement provide performance benefits over a naive use 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?

For this paper, we have not addressed experimentally the question of the soundness of our implementation. Rather, we are working with an independent “red team,” which has probed the system from an attacker’s point of view and has provided invaluable feedback. We also intend to release JAMScript as an open-source project to be subjected to scrutiny by the community.

4.1 Setup and Methodology

To answer the questions posed above, we collected a set of 20 distinct JavaScript applications and an additional 51 sub-applications of the SMS2 DNA analysis suite from the World Wide Web. 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.

We do not present data related to the strawman implementation described in §3.1, because it performed much 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 at a time. The latter approach has been used in prior work [5, 14] in which scripts are delimited by individual script tags or by the browser’s same-origin policy. Additionally, to test the absolute performance of transactions as an enforcement techniques.

<table>
<thead>
<tr>
<th>Application</th>
<th>AST Nodes</th>
<th>Description of Applied Policy</th>
<th>Transactions Inserted by JAM Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>squirrelmail</td>
<td>110</td>
<td>disallow access to src property</td>
<td>0</td>
</tr>
<tr>
<td>doubleclick-loader</td>
<td>271</td>
<td>prevent navigation</td>
<td>0</td>
</tr>
<tr>
<td>userprefs</td>
<td>375</td>
<td>disallow appendChild and eval</td>
<td>12</td>
</tr>
<tr>
<td>sunspider</td>
<td>407</td>
<td>disallow XMLHttpRequest.open</td>
<td>3</td>
</tr>
<tr>
<td>kraken</td>
<td>414</td>
<td>disallow XMLHttpRequest.open</td>
<td>3</td>
</tr>
<tr>
<td>beacon</td>
<td>787</td>
<td>isolate document from cookie</td>
<td>36</td>
</tr>
<tr>
<td>pluseone</td>
<td>1199</td>
<td>prevent script creation, document edits</td>
<td>34</td>
</tr>
<tr>
<td>imageloder</td>
<td>3957</td>
<td>disallow document.write</td>
<td>40</td>
</tr>
<tr>
<td>sms2-*</td>
<td>6656.7</td>
<td>prevent all network communication</td>
<td>162.3</td>
</tr>
<tr>
<td>snote</td>
<td>6852</td>
<td>certain elements write-once/readonly</td>
<td>59</td>
</tr>
<tr>
<td>prwik</td>
<td>7132</td>
<td>isolate document from cookie</td>
<td>230</td>
</tr>
<tr>
<td>mwwidgets</td>
<td>7504</td>
<td>certain elements write-once/readonly</td>
<td>59</td>
</tr>
<tr>
<td>mudon</td>
<td>9018</td>
<td>prevent modification of cookie</td>
<td>17</td>
</tr>
<tr>
<td>greybox</td>
<td>9914</td>
<td>prevent creation of script elements</td>
<td>343</td>
</tr>
<tr>
<td>googleplay</td>
<td>11803</td>
<td>disallow document write</td>
<td>40</td>
</tr>
<tr>
<td>ga</td>
<td>13236</td>
<td>prevent script creation, document edits</td>
<td>388</td>
</tr>
<tr>
<td>halurespawn</td>
<td>30269</td>
<td>disallow local storage access</td>
<td>1496</td>
</tr>
<tr>
<td>colorpicker</td>
<td>32653</td>
<td>prevent navigation, src/cookie access</td>
<td>309</td>
</tr>
<tr>
<td>afshlsi</td>
<td>37709</td>
<td>isolate document from cookie</td>
<td>227</td>
</tr>
<tr>
<td>puzzle</td>
<td>104486</td>
<td>prevent creation of script elements</td>
<td>319</td>
</tr>
</tbody>
</table>

Figure 8: Program size and number of woven transactions for analyzed applications. The “sms2-*” item represents the average over a set of 51 applications taken from the Sequence Manipulation Suite for DNA analysis.
A substantial benefit of the automated policy-weaving approach to program security is that it permits policies to be specified declaratively for the secured applications to that of the original program.

The static analysis was performed by the JAM policy weaver [7], modified to use the rewriting technique described in Alg. 2. 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 Fig. 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 represent opposing sides 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, as we measured an overall 25% speed-up for program actions instrumented with fine-grained transactions versus coarse-grained transactions when summarized by the geometric mean. This breaks into a 12% speed-up for page-load actions versus coarse-grained transactions when summarized by the geometric mean. This breaks into a 12% speed-up for page-load actions versus coarse-grained transactions when summarized by the geometric mean. This breaks into a 12% speed-up for page-load actions versus coarse-grained transactions when summarized by the geometric mean.

The static analysis was performed by the JAM policy weaver [7], modified to use the rewriting technique described in Alg. 2. 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.

### 4.3 Policy Complexity

A substantial benefit of the automated policy-weaving approach to program security is that it permits policies to be specified declarative for the secured applications to that of the original program.

The static analysis was performed by the JAM policy weaver [7], modified to use the rewriting technique described in Alg. 2. 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.

### 4.3 Policy Complexity

A substantial benefit of the automated policy-weaving approach to program security is that it permits policies to be specified declarative for the secured applications to that of the original program.

The static analysis was performed by the JAM policy weaver [7], modified to use the rewriting technique described in Alg. 2. 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.

### 4.3 Policy Complexity

A substantial benefit of the automated policy-weaving approach to program security is that it permits policies to be specified declarative for the secured applications to that of the original program.

The static analysis was performed by the JAM policy weaver [7], modified to use the rewriting technique described in Alg. 2. 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.

### 4.3 Policy Complexity

A substantial benefit of the automated policy-weaving approach to program security is that it permits policies to be specified declarative for the secured applications to that of the original program.

The static analysis was performed by the JAM policy weaver [7], modified to use the rewriting technique described in Alg. 2. 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.

### 4.3 Policy Complexity

A substantial benefit of the automated policy-weaving approach to program security is that it permits policies to be specified declarative for the secured applications to that of the original program.
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 Fig. 1 (the input policy) and Fig. 3(a) (the specialized policy). The baseline policy, which is a single introspector function that monitors all transitions of the input policy for a transaction object, is not shown. The policy weaver that we have integrated with JAMScript provides a policy-specification language, which is a dialect of JavaScript that supports predicates over calls (and corresponding arguments) to native functions, reads and writes to properties of native objects, reference (in)equality, and various string relations. To quantify the benefits provided by the automated production of introspection code, we compare the size of the input policy to the size of the generated code in Fig. 11. Summarized by the geometric mean, the implementation of the baseline policy is 6.3x larger than the input policy for our benchmarks, and the specialized policy is 14.5x larger. The size of the representation is admittedly an indirect measure of implementation complexity, but we also propose 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

Transaction-Based Policy Enforcement. The use of transactional introspection for security-policy enforcement can be viewed as an evolution of inline reference monitoring, which was developed around 2000 by Schneider and Erlingsson [6, 15, 17]. 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.

Transcript [5] is an extension to the JavaScript language that implements speculative execution with introspection for security-policy enforcement. As presented, transactions are applied in a modular fashion to untrusted “guest” code, delineated as different source-code files included in a web page via HTML script tags. 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.

The Transactional Memory Introspection (TMI) system [2] applied transactional instrumentation to multithreaded server software and demonstrated the benefits of the approach in that context. The JavaScript language does not exhibit true multithreaded behavior (the proposed standardization of the Worker API allows for parallel execution, but with each thread in a natively isolated environment, and communication limited to the exchange of strings [18]). Thus, TMI addresses issues that are orthogonal to the ones addressed in this paper.

Richards et al. [14] 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 speculatively executed 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 reduced by using the results of a conservative static analysis. The assumption that the host code can be trusted may be reasonable in some contexts, but leaves open the possibility of indirect subversion of the policy by clever attackers that can manipulate the environment to coerce the host code into violating the intended policy.

It also precludes the practice of hosting copies of untrusted third-party code or integrating untrusted code snippets into the host program. Another difference between their work and ours is that the introspection code in [14] 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.

The TxBox system [9] provides transactional introspection as an operating-system security feature. Their approach uses system-level policies to limit the access of applications to system resources, in contrast to JAMScript’s application-specific policies that can restrict internal program behavior. Our approach is also distinguished by the use of static analysis for runtime performance benefits, as well as the techniques for securing dynamically generated code.

Aspect Weaving for JavaScript. A number of prior works investigate an aspect-oriented approach to enforcing 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 mechanism, and the resulting ability to safely handle the full JavaScript language, rather than a restricted subset.

Fredriksson et al. [7] describe the JAM policy weaver for JavaScript as an implementation of their general technique. Policy weaving is differentiated from aspect weaving by an emphasis on semantics-based—as opposed to syntax-based—rewriting.) 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 with a wide range of static-analysis techniques.

Yu et al. [19] present a rewriting scheme that introduces the concept of callsite indirectness for applying security policies to dynamically generated code. That work emphasizes a comprehensive and sound placement of instrumentation and the formal specification of policies. In contrast, we discuss the concrete requirements of an effective enforcement mechanism and the translation of policies into operational code.

ConScript is a browser-based aspect system for security [12]. The primary security mechanism utilized by ConScript is a built-in form of indirectness (implemented as a modification to the JavaScript interpreter in Internet Explorer 8). This mechanism allows a function object to be wrapped within a user-defined function that vets arguments and potentially suppress calls at runtime. ConScript policies are constructed either manually or by an automatic dynamic-observation process, and a static analysis is used to gain confidence (but not prove) that the implementations are correct. In contrast, in our system the policy implementation is generated automatically from a declarative specification.

6. Conclusions

We have developed and evaluated an approach for using transactional introspection and callsite indirectness as integrated runtime enforcement mechanisms for policy weaving. Introspection enables the examination and potential suppression of the effects of program statements identified by a static analysis, and indirectness serves as a lightweight runtime analysis that can apply introspection to dynamically generated code that is not available to the static analysis. We have shown how programs can be rewritten to apply these techniques; the resulting programs are protected against policy violations for all possible program executions.
References


