Efficient Runtime-Enforcement Techniques for Policy Weaving

Richard Joiner\(^1\), Thomas Reps\(^1\),\(^5\), Somesh Jha\(^1\), Mohan Dhawan\(^2\), Vinod Ganapathy\(^2\)

\(^1\)University of Wisconsin-Madison, USA \(^2\)GrammaTech, Inc., Ithaca, NY, USA
\(^3\)IBM Research, New Delhi, India \(^4\)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. 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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

\(^*\) 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.

1. INTRODUCTION

Policy weaving for security-policy enforcement is a program-rewriting approach oriented towards striking a balance between static and runtime analysis techniques [1, 19]. 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-rewriting approach is motivated by the acknowledgment that in non-trivial scenarios, static analysis alone cannot prove that a program adheres to a policy [7, 16]. In contrast, runtime analyses such as inline reference monitoring (IRM) [10, 11, 24] 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 [4, 9, 15, 23], and some difficulties have been identified:
• Performance overhead of speculative execution can be prohibitive [6, 17].
• Correct placement of transactional instrumentation is a nontrivial task [15, 23, 25].
• Implementation of introspection logic to recognize and prevent policy violations can be error-prone [23].

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. [28], 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 [1, 5, 12, 14, 19]: the monitoring and enforcement of safety properties during program execution
• Policy/aspect weaving [13, 20, 26, 27]: the use of static analysis and rewriting for fine-grained placement of instrumentation
• Transactional introspection [4, 9, 15, 23]: a powerful monitoring and enforcement tool that allows effects of program execution to be examined before being committed
• Statement indirection [20, 28]: 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.

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 [9, 23]. 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.)

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 2(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 2(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 3.

- Each introspector evaluates a unique combination of policy predicates, and they each can read and write access to the global policyStates array.

This rewriting results in the following runtime behavior:

- 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 §2.3).
RewriteIntrospect \( \mathcal{P}, \Phi, \Psi_\Phi \):

Data: \( \mathcal{P} \): source code of the program being analyzed
\( \Phi \): policy automaton as a set of transitions
\( \Psi_\Phi = \{ (s_0, \tau_0), \ldots, (s_n, \tau_n) \} \): a policy-violating witness

Result: \( \mathcal{P} \) rewritten to prevent \( \Psi_\Phi \)
1. foreach \( (s, \tau) \in \Psi_\Phi \) do
2. Let \( T_\tau \) be the set of policy transitions for which \( s \) is already instrumented in \( \mathcal{P} \)
3. Let \( \sigma_{\text{arg}} \) be the original, uninstrumented statement corresponding to \( s \)
4. if \( \tau \) is null then
5. \( T_\tau \leftarrow T_\tau \cup \{ \tau \} \)
6. if \( T_\tau = \emptyset \) then
7. \( \text{inspect} \leftarrow \text{null} \)
else
8. Generate or retrieve \( \text{inspect} \), the introspector function that implements Algorithm 2 for all \( \tau \in T_\tau \)
9. if \( \sigma_{\text{arg}} \) potentially invokes a higher-order script then
10. \( \mathcal{P} \leftarrow \text{RewriteIndirect}(\mathcal{P}, \Phi, \sigma_{\text{arg}}, \text{inspect} \) \)
else
12. \( \mathcal{P} \leftarrow \text{RewriteIndirect}(\mathcal{P}, \Phi, \Psi_\Phi, \text{inspect} \) \)
13. let \( s' \) be introspect(\( \text{inspect} \)) \( \{ \sigma_{\text{arg}} \} \)
14. Replace \( s \) with \( s' \) in \( \mathcal{P} \)
15. return \( \mathcal{P} \)

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 \( (s, \tau) \) of the witness \( \Psi_\Phi \) consists of a program statement \( s \) and a policy transition \( \tau \) which 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 \( \{ \phi_{\text{pre}}, \phi, \phi_{\text{post}} \} \) consisting of a policy state \( \phi_{\text{pre}} \), a policy state \( \phi_{\text{post}} \), and a predicate \( \phi \) that induces the transition.

2.2 A Formalization of Rewriting

The rewriting that occurs between Figure 2(a) and Figure 2(b) is accomplished via a modified and extended version of the rewriting step of the SafetyWeave algorithm formalized by Fredrikson et al. [13]. 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 \( \text{pre}(s, \varphi_{\text{violation}}) \) of the subsequent state \( s \) and policy predicate \( \varphi_{\text{violation}} \) with respect to the current program state \( \sigma_{\text{pre}} \), and terminates the program if \( \text{pre}(s, \varphi_{\text{violation}})(\sigma_{\text{pre}}) = \text{true} \) (i.e., the execution of the statement could cause a policy violation).

A key insight that led us to integrate SafetyWeave with a 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 \( \text{post} \), which maps the program state \( \sigma_{\text{pre}} \) encountered at transaction entry and the enclosed statement \( s \) to the post-state \( \sigma_{\text{post}} \) that results from the execution of \( \text{post}(\sigma_{\text{pre}}, s) \). A policy predicate \( \varphi_{\text{violation}} \) can then be directly evaluated in this post-state. Formally, the condition for policy violation is \( \text{post}(\sigma_{\text{pre}}, s) \vdash \varphi_{\text{violation}} \).

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 \( \varphi_{\text{violation}} \) 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-
RewriteIndirect($P, \Phi, s, \text{ispect}_e$):

Data: $P$: source code of the program being analyzed
$\Phi$: policy automaton as a set of transitions
$s$: a program statement
\text{ispect}_e$: introspector function that implements Algorithm 2

For all policy transitions that may affect the policy state, directly or by invoking a higher-order script

Result: $P$ with $s$ rewritten by statement indirection

1. Let $e$ be the first subexpression of $s$ (according to execution order) that
   - may affect the policy state, directly or by invoking a higher-order script
   - is not already within an indirection expression

2. If $e$ is not null then
   3. Let $\text{type}_e$ be the expression type of $e$
   4. Let $\text{parts}_e$ be an array of subexpressions comprising $e$
   5. Retrieve $\text{ispect}_e$ of the introspector function that implements
      Algorithm 2 for all transitions $r \in \Phi$
   6. Let $s'$ be the result of replacing $e$ with $s'$ in $s$
   7. Replace $s$ with $s'$ in $P$
   8. Return $\text{RewriteIndirect}(P, \Phi, s', \text{ispect}_e)$

End 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 4.

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

trospector (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 state-
ments 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 [13], which requires an implementation of the pre-image
operator to determine when $s$ will cause a policy transition, the for-
mulation described here relies on the inherent properties of trans-
actional introspection to effectively apply the post-state operator
in the form specified in Algorithm 2 (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 im-
pose this requirement. In fact, we find that maintaining both atomi-
city and security—in the presence of actions that have externally
visible or unpredictable results—is impractical at best. In an im-
perative 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 [9] 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 suc-
cession of partial commits, rather than as an atomic unit.

The introduction of transaction suspension does not require altera-
tion of Algorithm 2. Upon suspension, the action sequence
available through the transaction object will be a consistent prefx
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 pre-
dictable 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 sus-
pension is useful in §3, but we maintain that some form of sus-
pension 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. [28] is ac-
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 Java-
Script 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 estab-
lished by first noting that any higher-order script invoked within
an introspect block will trigger a suspension of the currently ex-
cuting transaction, at which point the generated script is extracted
and wrapped in another transaction block (see lines 10–12 of Al-
gorithm 2). 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 propa-
gated transitively to all scripts that are executed directly or indi-
rectly 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 transac-
tional introspection and statement indirection. Transactional intro-
ception implemented by Algorithm 2 and woven by Algorithm 1 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 4 and woven by Algorithm 3 is a sim-
ple 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 exten-
sion 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 per-
formance 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 re-
ported back to the original process. Despite copy-on-write seman-
tics 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 code application takes about 13ms; with a ver-
son 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 envi-
ronment. We found that due to the underlying mechanism of isola-
tion 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 behav-
ior, 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 consid-
ered, 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 vi-
olation 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 ForkIsol-
ate 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 capabili-
ties to be loaded as necessary for each embedding system, and (ii)
uses transactions to record a sequence of all actions that occur dur-
ing speculative execution.

3.2 Description of JAMScript
With JAMScript, we adopt transactional introspection as the key
enforcement component for static policy weaving. We have imple-
mented 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 (de-
limited 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 imple-
mentation 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 1.
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 embed-
ning 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 §2.3, 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-
tions to source files that implement DOM functionality. As noted by
Yu et al. [28], document.write (or writeln) can be used to out-
put
Object.defineProperty(this, "JAM", { value: (function() {
  var _eval = eval;
  var _call = Function.prototype.call;
  var _document_write = HTMLDocument.prototype.write;
  var _Array_slice = Array.slice;

  return {
    policy: policy,
    commitAction: function(a) {
      if (a.type === "call") {
        var fun = a.target, args = a.args, obj = a.object;
        if (fun === _eval) {
          args[0] = "introspect(JAM.policy.full)" + args[0] + ";"
        } else if (fun === _document_write) {
          JAM.pushDynamicIntrospector(JAM.policy.full);
          var ret = fun.apply(obj, args);
          JAM.popDynamicIntrospector();
          return ret;
        }
      }
      return fun.apply(obj, args);
    }
  }

  var policy0_F = function(tx) {
    var commit = true;
    var ws = tx.getCodeSequence();
    for (var i=0; i<ws.length; i++) {
      var a = ws[i];
      if (a.type === "write" & & a.id === "src" & & a.object instanceof _HTMLImageElement) {
        if (commit) JAM.commit(tx);
        else JAM.prevent(tx);
      }
    }
    return {
      policy0_F: policy0_F,
      full: policy0_F,
    };
  }();

  Object.freeze(policy);

})(this, "JAM");

var _eval = eval;
var _call = Function.prototype.call;
var _document_write = HTMLDocument.prototype.write;
var _Array_slice = Array.slice;

return {
  policy: policy,
  commitAction: function(a) {
    if (a.type === "call") {
      var fun = a.target, args = a.args, obj = a.object;
      if (fun === _eval) {
        args[0] = "introspect(JAM.policy.full)"
      } else if (fun === _document_write) {
        JAM.pushDynamicIntrospector(JAM.policy.full);
        var ret = fun.apply(obj, args);
        JAM.popDynamicIntrospector();
        return ret;
      }
    }
    return fun.apply(obj, args);
  }
}

var policy = (function() {
  var _HTMLImageElement = HTMLImageElement;

  function policy0_F(tx) {
    var commit = true;
    var ws = tx.getCodeSequence();
    for (var i=0; i<ws.length; i++) {
      var a = ws[i];
      if (a.type === "write" & & a.id === "src" & & a.object instanceof _HTMLImageElement) {
        if (commit) JAM.commit(tx);
        else JAM.prevent(tx);
      }
    }
    return {
      policy0_F: policy0_F,
      full: policy0_F,
    };
  }();

  Object.freeze(policy);

})(this, "JAM");

isect = JAM.policy.full;
else 
  return f.apply(rec, args);
},
  };
  */ end JAM object literal */
}(function() /* end anonymous function call */{
  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 li-
brary methods are elided to conserve space.) Lines 2–5 define private
references to native objects. The policy object (defined in Figure 5) is
incorporated (line 8), and the JAM library object is frozen (line 44) so
that its properties cannot be overwritten.

put 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 imple-
mentation associates these functions with new scripts as they are
created. When such a script begins execution, the JavaScript inter-
preter runs it as if it were enclosed in (nested) transaction blocks
with the associated functions as the introspectors. Lines 16–21 in
Figure 4 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 pro-
tect 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 mon-
itoring are statically known, so the instrumentation is packaged as
objects with methods that close over these references when created.
Both the JAM library (Figure 4) and the policy object (Figure 5)
are initialized by an anonymous function with local variables and
nested functions that reference these variables. The nested func-
tions become methods of the returned enforcement object, and
JavaScript’s lexical scoping ensures that the private state is acces-
sible only from within these trusted methods.

We must also ensure that the methods of the JAM library can-
not 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 4 to set
the writeable and configurable attributes to false, and use
Object.freeze to render all properties immutable (line 44). Java-
Script 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 “shad-
ows,” 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 execu-
tion 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 con-
structor has led many security researchers to limit their tools to lan-
guage subsets [3]. However, surveys have shown that higher-order
scripts are commonly used in existing JavaScript programs [22].
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 sec-
tion, 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.

Consider the callsite in Figure 6(a). If static analysis cannot ex-
clude that obj.meth references the eval function, we must con-
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 7. 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, anote, and mwwidgets) were developed and contributed by the team at MIT Lincoln Laboratory.

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 [9, 23] 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 [13], modified to use the rewriting technique presented in Algorithm 1 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 8 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 opposite ends of a performance trade-off between transaction start-up time and in-transaction processing time. A weav-
ing 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 9 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.

### 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 3. 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 3 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 refer 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 3) 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 3), and is generally larger because multiple introspector functions are needed to evaluate different combinations of policy predicates.
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 10 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 allowed 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 [1, 5, 19] 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 [2, 12, 14]. In contrast, our work purses 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. [13] describe the JAM policy weaver for Java-Script 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. [28] 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 [21], ADsafe [8], ConScript [20], and Maffeis et al. [18], 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 inline reference monitoring, which was developed around 2000 by Schneider and Erlingsson [10, 11, 24]. 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.

Script [9] is an extension to the JavaScript language that implements speculative execution with introspection for security-policy enforcement. As presented in [9], 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 Script 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. [23] 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 [23] 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) [4] 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. TBox [15] 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.
6. REFERENCES


