Policy Weaving with Simple Transactional Enforcement *

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

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

Abstract

The ability to speculatively execute code within a transaction constitutes a powerful tool for ensuring correct program behavior; such a scheme enables examination, and potential suppression, of the concrete effects of a block of code prior to the realization of those effects in the program state. One important form of "correct behavior" is adherence to a stateful security policy. However, given the cross-cutting nature of security policy enforcement, the manual processes of applying transactional instrumention to a program and generating the code needed to introspect on the speculative state of a transaction can be complex and error-prone, and therefore any assurances may be tenuous. In this paper, we examine the benefits of applying policy weaving, a form of tunable static analysis resulting in a rewritten program proven to be safe with respect to a security policy, to the problem of correct and comprehensive transaction placement and introspection. We describe an implementation and report experimental results showing the viability of this approach in the context of real-world JavaScript programs running in a browser.

1. Introduction

In this paper, we explore and demonstrate the utility, in the context of security policy enforcement, of complementing the static analysis of policy-weaving with the dynamic flexibility of transactional memory semantics. We integrate the work of [3] and [4] to generate a joint formalism as well as a concrete instantiation of the methodology.

A policy weaving algorithm takes as input a program to be secured and a security policy in the form of a finite-state automaton that accepts disallowed execution traces. It produces a securely rewritten program in which potentially policy-violating statements are monitored at runtime such that mitigating action can be taken immediately prior to any violation of the policy. The implementation of the monitoring and enforcement mechanism is independent of the algorithm.

We find that the transactional memory model provides a straightforward and effective target 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 calculated and examined prior to the application of those effects to the execution environment.

Transactional memory schemes have been studied in other security literature, and a consistent concerns arise with regard to performance during speculative execution [2] and correctness of transaction placement and introspection [5, 6]. We address both concerns via the formulation described here. The complexity of both manually placing individual transactions and constructing introspection logic has been reduced to formulation of the security automaton, which, in addition to being a much smaller and self-contained artifact, more directly coincides with the intentions and goals of the user. Conversely we find that the use of a transactional paradigm serves as a natural and intuitive runtime enforcement platform for policy weaving, and results in 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 memory instrumentation into a program to enforce security invariants.
- We demonstrate the automatic translation of a security policy specification into transaction introspection code, and show that this substantially reduces the opportunity for implementation error.
- We present JAMScript, a general-purpose but simple extension of the JavaScript language implementing transactional semantics with properties ideally suited to security policy enforcement. This system is an evolution and simplification of the Transcript system described in [4].
- We provide experimental results demonstrating the effectiveness and favorable performance of the system when compared to manual placement of transactions and to other types of enforcement mechanisms that could be targeted by a policy-weaving algorithm.

Organization. §2 gives an overview of language-independent policy weaving techniques and transactional semantics, and how they work together. §3 presents formal specification of the SafetyWeave policy-weaving algorithm, the relevant semantics of transactional

* 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., which has licensed elements of the technology reported in this publication.

memory, and how the two fit together. §4 discusses the unique implementation challenges encountered when applying the composite system to JavaScript, and the solutions. §5 evaluates the performance of JAMScript over a set of real applications. §6 discusses related work. §7 concludes.

2. Overview

This paper will examine the requirements and goals that are, until now, implicit in the task of choosing an instantiation of the enforcement mechanism targeted by a policy-weaving algorithm.

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

- Runtime overhead induced by the mechanism should be minimal and maintain acceptable user experience.
- The mechanism should be powerful and flexible enough to precisely evaluate the policy at runtime, without spuriously blocking traces.
- The mechanism should exhibit minimal semantic complexity, to avoid complicating static analysis and manual understanding.

Since the latter two concerns are directly at odds with each other, and both may indirectly hinder the former (runtime performance), it may be difficult or impossible to arrive at an ideal solution with respect to all of these principles simultaneously. Interestingly, we will show that the methodology of policy-weaving can in effect bring the enforcement mechanism closer to achieving each of these principles than it otherwise would.

We will now introduce an example program and a security policy that we would like to enforce via transactional memory introspection. The code shown in Fig. 1(b) represents a simple program that may violate the policy that is schematically shown in Fig. 1(a), which asserts that network communication should not occur subsequent to certain accesses of local information. A potentially offending statement is simply enclosed in a transaction, with the `doSecure` function, defined to the right in Fig. 1(d) as the transaction "introspector." Here the policy has been translated to to code implementing a runtime monitor on the transacted state; the function containing this logic is invoked after the effects of the speculative execution are logged. As a result of the static analysis, the policy transition labeled `call readHistory` need not be evaluated for this program.

The rewriting that occurs between Fig. 1(b) and Fig. 1(c) is accomplished via a simplified version of the SafetyWeave algorithm laid out in [3]. While that paper focuses primarily on the static policy-weaving algorithm, it represents the targeted enforcement mechanism as an inline reference monitor that evaluates a predicate representing the weakest precondition for a policy transition with respect to the subsequent statement and terminates the program prior to any policy violation.

A primary insight leading to the integration of SafetyWeave with a transactional enforcement mechanism as proposed in this paper is that the semantics of speculative execution allows predicates constituting the policy to be directly evaluated in the context of the (speculative) program state. Viewed another way, transactional semantics are a runtime implementation of the strongest postcondition operator, which is the dual construct of weakest liberal precondition. This fact allows the rewriting step of SafetyWeave to be transparently integrated with a transactional memory enforcement mechanism. Therefore the transactional memory model for security policy enforcement studied in recent work [1, 4, 5] presented as an excellent candidate to obviate the fraught task of implementing of the symbolic precondition operator.

In §3, we will specify the alterations made to SafetyWeave to accomplish this novel integration, and the required characteristics of a transactional memory implementation that are needed or helpful towards this integration. In §4, we describe an extension to the JavaScript language that implements transactional semantics and show how it has evolved to better support the principles listed above.

3. Technical Description

We will now present a language-independent description of the incorporation of dynamic speculative execution via transactions as the targeted enforcement primitive for the SafetyWeave policy-weaving algorithm.

3.1 Modifications to SafetyWeave

[3] describes the language-independent SafetyWeave algorithm as an instantiation of policy-weaving that rewrites a program $\mathcal{P}$ to adhere to a security policy Φ by conservatively inserting dynamic policy checks at statically determined locations. This methodology constitutes a hybrid static-dynamic approach that departs from a traditional static verification algorithm through the introduction of a program rewriting step that is invoked in two situations.

1. A valid execution trace is identified
2. An invalid execution trace is identified, and a configurable resource bound has been met

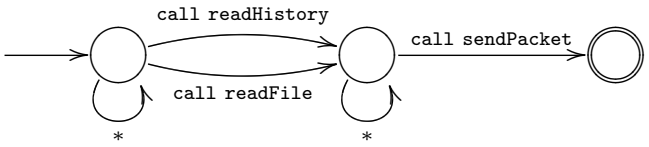
Here, a "valid" execution trace is one that is realizable in the concrete program, as determined via symbolic execution, and an "invalid" trace is one that is cannot occur in the concrete program, but is extracted from the abstract model of the program due to imprecision. Rewriting the program in the first case allows the policy-weaving algorithm to produce a safely instrumented program from an original program that can generate policy violations, while the rewriting in the second case allows the algorithm to terminate in finite and bounded time despite complexity that would normally preclude static analysis.

The nature of the dynamic enforcement mechanism (the primitive inserted into the program by the SafetyWeave rewriting step) is discussed in formal but generic terms in [3]; the methodology of policy-weaving is formulated at a high level such that implementation details of both the static analysis and the runtime enforcement can be specific to the domain of application. In the definition below, the "dynamic check" Ψ^{π}_{Φ} is a sequence representing a potentially policy-violating trace consisting of statements associated with a policy transition predicate. Enforce, the rewriting step in the SafetyWeave algorithm converts Ψ^{π}_{Φ}—the output of a single round of static analysis—into a new program $\mathcal{P}'$, and is defined here in close paraphrase of [3].

DEFINITION 3.1. *(Functionality of* Enforce*).* Given a program $\mathcal{P}$ and a dynamic check $\Psi^{\pi}_{\Phi} = \{(s_0, \phi_0), \ldots, (s_n, \phi_n)\}$, Enforce produces a new program $\mathcal{P}'$. $\mathcal{P}'$ uses a numeric variable, `policy`, which is initialized to zero. Enforce performs the following steps for each element $(s_i, \phi_i) \in \Psi^{\pi}_{\Phi}$:

1. Let $\phi_{\text{pre}} \equiv \mathsf{wlp}(s_i, \phi_i) \wedge \texttt{policy} = i$
2. Insert a new statement before s_i that either:
   - Increments `policy` whenever ϕ_{pre} is true and `policy` $< |\Psi^{\pi}_{\Phi}|$.
   - Halts the execution of $\mathcal{P}'$ whenever ϕ_{pre} is true and `policy` $= |\Psi^{\pi}_{\Phi}|$.

Informally, given a policy-violating execution trace that is realizable in the conservative, abstract model of the program, the program is rewritten to include a guard for predicate ϕ immediately prior to each statement s that corresponds to a policy state transition in the trace. These checks compare the state of the program to the precondition for policy predicate ϕ relative to s. This entails an



(a) Security policy that says "do not read from a file or the history and subsequently write to the network."

```
1 api[0] = readFile;
2 api[1] = sendPacket;
3 fun execute(instr, data) {
4   api[instr](data);
5 }
6 while(*) {
7   instr, data = read();
8   execute(instr, data);
9 }
```

(b) Original code.

```
1 api[0] = readFile;
2 api[1] = sendPacket;
3 fun execute(instr, data) {
4   transaction(doSecure) {
5     api[instr](data);
6   }
7 }
8 while(*) {
9   instr, data = read();
10  execute(instr, data);
11 }
```

(c) Secured code. On line 4 is the opening of a transaction that guards against a potential policy violation at line 5.

```
1 policyState = [0];
2 fun doSecure(tx) {
3   cs = tx.getCallSequence();
4   for (c in cs) {
5     if (policyState.contains(0)
6       && c.target == readFile) {
7         policyState.append(1);
8     }
9     if (policyState.contains(1)
10      && c.target == sendPacket) {
11        return;
12    }
13  }
14  tx.commit();
15 }
```

(d) Introspection code that examines actions recorded during the transaction. Static analysis determines `call readHistory` need not be checked for this program.

Figure 1. An example security policy, a program that may violate it, and a secured version.

implementation the weakest liberal precondition operator to determine the condition on which the subsequent program statement s will generate a policy-transitioning state (i.e. $wlp(s, \phi)$).

We can now specify an alternative but essentially equivalent definition of Enforce that incorporates an enforcement mechanism with transactional semantics. For fun and distinction we'll call it Transforce. The main thing to notice is that the step involving the computation of $\mathsf{wlp}(s_i, \phi_i)$ drops out of the algorithm.

DEFINITION 3.2. *(Functionality of* Transforce*).* Given a program $\mathcal{P}$ and a dynamic check $\Psi^{\pi}_{\Phi} = \{(s_0, \phi_0), \ldots, (s_n, \phi_n)\}$, Transforce produces a new program $\mathcal{P}'$. $\mathcal{P}'$ uses a numeric variable, `policy`, which is initialized to zero. Transforce performs the following steps for each element $(s_i, \phi_i) \in \Psi^{\pi}_{\Phi}$:

1. Retrieve or generate a handler function $h_{\phi i}$ implementing the following.
   - If `policy` $\neq i$, commit the transaction and return.
   - Else evaluate ϕ_i in the speculative state
   - Increment `policy` whenever ϕ_i is true and `policy` $< |\Psi^{\pi}_{\Phi}|$.
   - Halt the execution of $\mathcal{P}'$ whenever ϕ_i is true and `policy` $= |\Psi^{\pi}_{\Phi}|$.
2. Enclose s_i in a transaction with handler argument $h_{\phi i}$.

4. Implementation

This section discusses the implementation of JAMScript, which is the evolution and simplification of the Transcript extension to the JavaScript language, and its integration with the JAM policy-weaver.

4.1 Origins of the Implementation

The implementation of SafetyWeave developed as a prototype in [3] is a JavaScript analysis dubbed JAM. While examples of the dynamic checks as described above in relation to the general algorithm are provided in C-like pseudocode, a general and automated implementation of the symbolic precondition operator for an arbitrary policy predicate turned out to be prohibitively complex in practice when applied to a language with such complex and unconstrained semantics as JavaScript. As an extreme example, consider the JavaScript statement s : `eval(v)`; and the policy predicate `set(document.cookie)` (which is true in the poststate of s when the current webpage's cookie is written within s). What is the precondition for this poststate relative to s? Given that `v` may be an arbitrary string representing an arbitrary JavaScript statement (potentially invoking a function containing arbitrary statements), it should be clear that a succinct formulation of such a precondition is not possible.

As previously stated, the integration of JAM with a speculative execution framework allows predicates constituting the policy being enforced to be directly evaluated in the context of the (speculative) program state by emulating the strongest postcondition operator instead of calculating the weakest liberal precondition. Given the implementation status and promising performance results reported in [4] in an evaluation of the Transcript system, we adopted this work as a starting point for the implementation of our enforcement mechanism.

The Transcript system, an extension of JavaScript for imposing transactional semantics on untrusted scripts [4] exhibits characteristics that are desirable in a runtime enforcement mechanism as required by JAM. While the use cases for Transcript as anticipated in [4] and required by JAM differ somewhat, we find that the capabilities offered by Transcript come close to satifying the goal of introspecting on the effects of statements prior to executing them.

In fact, the very flexible semantics of Transcript transactions and all the features that the system provides proved to be more than was needed for the purposes of JAM, and therefore we sought to restrict the behavior of transactions to the minimum needed to enforce security policies. Simple, well-determined behavior is important in the context of a static program analysis; we did not want to introduce features that might unnecessarily make the JavaScript language even more complex to soundly analyze. Furthermore, simple, well-determined behavior is very important to programmers work-

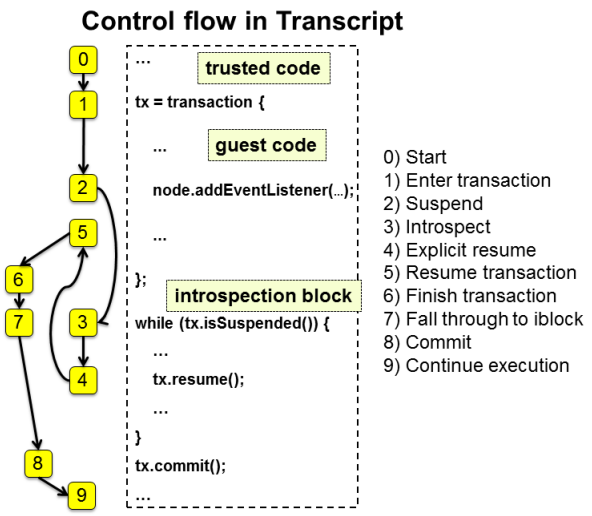


Figure 2. Control flow in a Transcript-enhanced interpreter.

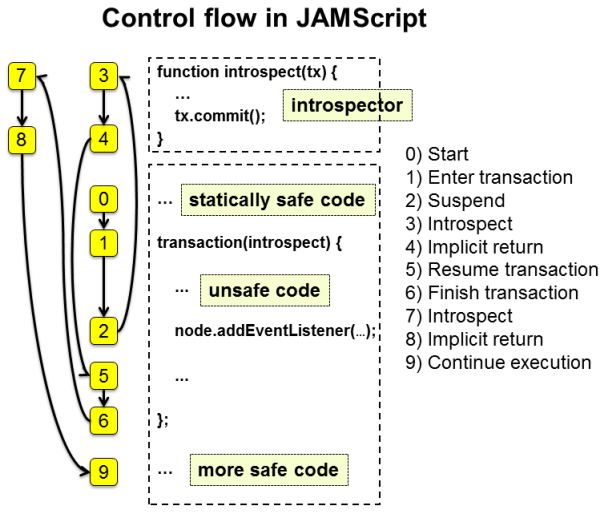


Figure 3. Control flow in a JAMScript-enhanced interpreter.

ing in a language. By restricting the semantics of transactions, we address criticism [5, 6] of the Transcript approach that it can be difficult to reason about and provides too many opportunities for user error.

4.2 Modular versus Granular Transactions

In contrast to the modular transaction paradigm envisioned by the designers of Transcript, in which few transactions containing large blocks of code would be executed in the course of loading a page, the JAM case requires that many transactions, each containing a single statement, ~~would be~~ invoked.

4.3 Simplified Semantics

The structure of a transaction block, its accompanying introspection block, the control flow between the two, ~~and the corresponding actions that occur in the Transcript runtime are reproduced from [4]~~ in Fig. 2. This diagram stands in contrast to the simplified model of JAMScript shown in Fig. 3.

As the Transcript implementation was adopted as the targeted enforcement mechanism for the JAM static analysis, it became clear that modifications to the system would be needed to facilitate the altered use case. However we also sought to maintain the general-purpose utility of transactions as a programming tool. Motivation for altering the architecture and semantics of transactions in JavaScript, a description of the resulting solutions, and performance measurements indicating the viability of the new approach with respect to the both the original and modified use cases will be presented in the following sections.

4.4 Transaction Suspension and DOM Consistency

The traditional definition of a transaction (originating in database literature) is based on the so-called ACID properties (atomicity, consistency, isolation and durability). The implementation of Transcript satisfies each of these except for atomicity; given the irrevocability of some actions (such as sending an `XMLHTTPRequest`) inherent to JavaScript running in a browser, Transcript implements the concept of transaction suspension prior to such actions. The suspend operation (which in Transcript occurs either automatically prior to irrevocable actions or manually via the `suspend` keyword) escapes execution of the transaction block into the introspection code where the speculative state up can be evaluated and committed prior to resumption. Due to this process, a transaction may end up being partially committed rather than as an atomic unit. Nevertheless, we maintain use of the term "transaction" throughout the paper when referring to the functionality of Transcript and of the updated system the we have integrated with the SafetyWeave analysis.

In JAMScript, we maintain the concept of transaction suspension, but restrict its application to predictable circumstances; in particular, suspension occurs at all invocations of native functions (including those triggered by accesses of properties with getter and setter methods). While we do maintain a small whitelist of some native JavaScript native functions, such as some pure functions of the `Math` object, all interfaces provided by systems outside of the core JavaScript language, for example the DOM, cause suspension of a running transaction to allow the ambient memory state to "catch up" to the speculative state prior to executing such calls, which may read or write to the memory. The upshot of this approach is that external systems that wish to integrate with a JavaScript interpreter do not need to consider the potentially transactional nature of what's occuring in the JavaScript heap.

As a corollary to this self-contained approach to the implementation of transactions, the JavaScript interpreter does not need to be aware of the systems into which it is integrated. The Transcript system maintains a cloned DOM structure to which all DOM-dependent actions are redirected during a transaction. This approach necessitates rather complex consistency checking and conflict resolution at transaction commit time, and still requires suspension of transactions for particular actions that may generate irrevocable side-effects (such as assignment to the `src` property of an `HTMLImageElement` object, which may trigger network communication to retrieve an image). The simple model of suspending a transaction on all natively implemented calls allows for transactional semantics of the JavaScript language that are not entangled with the semantics of ancillary systems.

Additionally, the situation described above is the *only* time that a transaction will suspend. This implies that there is no `suspend` keyword in JAMScript, as was implemented in Transcript. This decision is justified in two ways. First, a `suspend` keyword that can be invoked within the code being monitored grants that code some

```
var policy = (function() {

  // Policy states previously visited
  var states = [0];

  // Close over native objects for immutable references.
  var _HTMLElement = HTMLElement;

  // An evaluator checks a single policy predicate.
  function evaluateReads(tx) {
    var rs = tx.getReadSequence();
    for (var i=0; i<rs.length; i++) {
      var node = rs[i];
      if (node.id === "appendChild"
          && node.obj instanceof _HTMLElement) {
        return true;
      }
    }
    return false;
  }

  // Return the policy object itself.
  return {
    // Object contains methods that dispatch the
    // evaluator(s) and maintain the policy state.
    // See Appendix A for a full representation.
    // ...
  };
}());
```

Figure 4. Full representation of the method generating the `policy` object as a self-enclosed entity.

potential to muddle the control flow with results that are difficult to reason about statically, both by a programmer and a machine. Second, adding `suspend` to the language breaks backwards compatibility of the language specification in a way that is more subtle (and therefore harder to detect at parse time) than with the syntactically isolated `transaction` keyword.

4.5 Protecting the Instrumentation

One concern that prevails in the development of new security techniques ~~that use the target language to implement the technique~~ is that of protecting the instrumentation itself from modification and other forms of subversion. In JAMScript, we are able to leverage JavaScript's implementation of closures to create an immutable and self-enclosed introspectors. Because we are able to determine statically all the native objects that will need to be referenced by the enforcement code, the system automatically generates an introspector package that closes over these references when created. See Fig. 4 for an example of how this is implemented. This also indicates that the `policy` object maintains its state information privately. See §A for a full representation of the code that generates the policy object and the JAMScript library.

One assumption that is needed for full assurance is that the code that generates these closures is run prior to any modifications of the references at hand. Again, we rely on the static analysis to generate a final application that ensures this property; this is very simply accomplished by including the policy and JAMScript library code prior to any other JavaScript on the page.

JavaScript also presents the phenomenon of variable shadowing as a potential method of policy subversion. Since our immutable enforcement library is known to be referred to by the name `JAMScript`, it seems that the malicious guest code could also declare its own `JAMScript` variable whose value overrides, or "shadows," the instrumentation itself. This is addressed simply by a preprocessing step in the JAM static analysis which recognizes this situation and renames declared variables that would subvert the JAMScript instrumentation. Since the global `JAMScript` object is the entry point for all enforcement code, this is the only name of interest for this step, though the technique could easily be applied to multiple global variable names.

Application	AST Nodes	Transactions Inserted by JAM Analysis
SecureNote		
QRCode		
JavaScipt Menu		
Picture Puzzle		
GoogieSpell		
GreyBox		
Color Picker		
Analytics		
Midori		
PlusOne		
Hulu Respawn		
JSBeautifier		
Beacon		
DoubleClick		
Flickr		
SquirrelMail		
JSSec		

Figure 5. Statistics for applications analyzed.

Application	Action	Unprotected Runtime (s)	JAMScript Runtime (s)	Modular Runtime (s)
SecureNote	Load			
SecureNote	Edit			
SecureNote	Unlock			
QRCode	Load			
QRCode	Decode			
JavaScript Menu	Load			
Picture Puzzle	Load			
GoogieSpell	Load			
GreyBox	Load			
Color Picker	Load			
Analytics	Load			
Midori	Load			
PlusOne	Load			
Hulu Respawn	Load			
JSBeautifier	Load			
Beacon	Load			
DoubleClick	Load			
Flickr	Load			
SquirrelMail	Load			
JSSec	Load			

Figure 6. Performance of unprotected execution, JAMScript-protected execution, and manually placed modular transactions on various user actions in benchmark applications.

5. Experimental Results

In this experimental evaluation, we aim to answer the following questions:

- How do the performance characteristics of transactional memory executions manifest in real-world applications? Is the overhead acceptable?
- What are the characteristics of programs that benefit the most from a policy-weaving approach to transactional introspection?
- As a measure of user-exposed complexity, how does the size of a security-policy automaton compare to the generated introspection code?

Application	JAM Policy AST Nodes	Generated Policy AST Nodes
SecureNote		
QRCode		
JavaScipt Menu		
Picture Puzzle		
GoogieSpell		
GreyBox		
Color Picker		
Analytics		
Midori		
PlusOne		
Hulu Respawn		
JSBeautifier		
Beacon		
DoubleClick		
Flickr		
SquirrelMail		
JSSec		

Figure 7. Code locations in a JAM policy versus the automatically generated enforcement code.

6. Related Work

6.1 Transaction-based Policy Enforcement

[4] describes an implementation of speculative execution with introspection as an extension to the JavaScript language called Transcript. The system adds the `transaction` keyword and accompanying syntax that gives the programmer the ability to enclose sections of code in transaction blocks, and to provide introspection, or "iblock," code that can evaluate the actions that were recorded during the speculative execution and determine whether the speculative state should be committed or suppressed.

The goal for the Transcript system as presented by Dhawan, et. al. in [4] is that of security policy enforcement applied to untrusted code. They targeted the use case in which a web application is composed of several modules, delineated as different source code files included in a web page via HTML `script` tags. This is indeed the canonical formulation of the execution environment for JavaScript applications (TODO: cite other papers using this setup). To invoke the utility of Transcript, any modules that are deemed to be untrusted (usually defined as being provided by some untrusted third-party developer), are included via a `script` tag augmented with a new `func` attribute that references a JavaScript function; the body of the untrusted script is effectively wrapped in this function, which the host can then invoke in the context of a transaction.

[1] applies the principles of transactional instrumentation to multithreaded server software and demonstrates additional benefits of this 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, with communication via exchange of strings [8]), and thus parallelism does not factor in to our implementation, but this work demonstrates that transaction-based enforcement is viable and even preferable in a multithreaded context.

[5] applies speculative execution semantics (refered to as delimited histories) to untrusted code in an automatic but coarse-grained fashion, where untrustedness is based on the browser's same origin policy. Unlike in the JAM system, the entirety of the third-party code is speculatively executed, which, as we have shown, affects the performance profile of the execution. The assumption that the host code can be trusted may be reasonable in many contexts, though it always leaves open the possibility of indirect subversion of the policy by clever attackers that can manipulate the environment in such a way that coerces the host code into violating its own policy. It also precludes the practice of hosting copies of untrusted third-party code or integrating untrusted code snippets into the host program. Another differentiator is that the introspection code in [5] is written manually on a case-by-case (albeit reusable) basis in C++ modules, with the intent of shielding the instrumentation from manipulation. Contrastingly, JAM automatically produces introspection code as a translation of the policy automaton, and we provide a relatively simple scheme for protecting the integrity of the instrumentation.

6.2 Combining Static and Dynamic Analysis

[7] describes a technique for enforcing synchronization safety in multithreaded applications through a combination of abstraction refinement and program rewriting (in the form of lock insertion). This is similar to a policy-weaving algorithm in which the property being enforced is an absence of deadlocks and race conditions, and the enforcement mechanism consists of thread synchronization primitives. The choice of refining the abstraction or rewriting the program with synchronization primitives in [7] corresponds to the tunable balance in policy-weaving between continuing refinement in the CEGAR loop and weaving policy-monitoring instrumentation into the program. However the formulations diverge in that [7] may fail to find both a refinement strategy and a safe rewriting, in which case the algorithm must terminate without a program proven to be safe w.r.t. the synchronization property; the SafetyWeave algorithm used in JAM always terminates with a conservative instrumentation of the code. The application of policy-weaving to the enforcement of safe thread synchronization is an interesting topic for future research.

7. Conclusions

References

[1] A. Birgnisson, M. Dhawan, U. Erlingsson, V. Ganapathy, and L. Iftode. Enforcing authorization policies using transactional memory introspection. In *CCS*, 2008.

[2] C. Caşcaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, 6(5):46–58, September 2008.

[3] M. Fredrikson, R. Joiner, S. Jha, T. Reps, P. Porras, H. Saïdi, and V. Yegneswaran. Efficient runtime policy enforcement using counterexample-guided abstraction refinement. In *CAV*, 2012.

[4] V. G. Mohan Dhawan, Chung-chieh Shan. Enhancing javascript with transactions. In *ECOOP*, 2012.

[5] G. Richards, C. Hammer, F. Z. Nardellia, S. Jagannathan, and J. Vitek. Flexible access control for javascript. In *OOPSLA*, 2013.

[6] M. Song and E. Tilevich. Tae-js: Automated enhancement of javascript programs by leveraging the java annotations framework. In *PPPJ*, 2013.

[7] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *STTT*, 2013.

[8] WHATWG. Web workers - html standard. `http://www.whatwg.org/specs/web-apps/current-work/multipage/workers` 2013.

A. Instrumentation

This appendix provides a full representation of the policy object that is generated by the JAM analysis, as well as relevant portions of the JAMScript library that interact with the policy.

```
var policy = (function() {

  // Policy states previously visited
  var states = [0];

  // Close over native objects for immutable references.
  var _HTMLElement = HTMLElement;

  // An evaluator function checks a single policy predicate.
  function evaluateReads(tx) {
    var rs = tx.getReadSequence();
    for (var i=0; i<rs.length; i++) {
      var node = rs[i];
      if (node.id === "appendChild" && node.obj instanceof _HTMLElement) {
        return true;
      }
    }
    return false;
  }

  // A list of objects representing transitions of the policy
  var transitions = [
    {pre: 0, post: -1, evaluator: evaluateReads},
  ];

  // Return the policy object itself.
  return {
    // This method is invoked to check one policy transition.
    checkState: function(tx, trans) {

      // If prestate has not been reached, continue.
      if (states.indexOf(trans.pre) === -1)
        return true;

      // If the (non-final) poststate has been reached, continue.
      if (trans.post !== -1 && states.indexOf(trans.post) !== -1)
        return true;

      var evaluator = trans.evaluator;

      var bad = evaluator(tx);
      if (bad) {
        if (trans.post === -1) {
          // -1 indicates a final state; suppress the transaction.
          return false;
        } else if (trans.post > 0) {
          // Otherwise, log that we've reached this state.
          states.push(trans.post);
        }
      }
      return true;
    },

    // Map introspectors to sets of policy transitions.
    introspectors: {
      introspectReads: function(tx) { JAMScript.process(tx, [transitions[0]]); },
    },
  };
}());
```

Figure 8. Generating the `policy` object as a self-enclosed entity.

```
// TODO
```

Figure 9. This code generates the global `JAMScript` object, which encapsulates and interacts with the policy object shown in Fig. 8.