# Efficient Runtime Policy Enforcement Using Counterexample-Guided Abstraction Refinement

Matthew Fredrikson*, Richard Joiner*, Somesh Jha*, Thomas Reps*$, Phillip Porras†, Hassen Saïdi†, Vinod Yegneswaran†

*University of Wisconsin, $Grammatech, Inc., †SRI International

**Abstract.** Stateful security policies—which specify restrictions on behavior in terms of temporal safety properties—are a powerful tool for administrators to control the behavior of untrusted programs. However, the runtime overhead required to enforce them on real programs can be high. This paper describes a technique for *rewriting* programs to incorporate runtime checks so that all executions of the resulting program either satisfy the policy, or halt before violating it. By introducing a rewriting step before runtime enforcement, we are able to perform static analysis to optimize the code introduced to track the policy state. We developed a novel analysis, which builds on abstraction-refinement techniques, to derive a set of runtime policy checks to enforce a given policy—as well as their placement in the code. Furthermore, the abstraction refinement is tunable by the user, so that additional time spent in analysis results in fewer dynamic checks, and therefore more efficient code. We report experimental results on an implementation of the algorithm that supports policy checking for JavaScript programs.

## 1 Introduction

In this paper, we describe a technique that in-lines enforcement code for a broad class of stateful security policies. Our algorithm takes a program and a policy, represented as an automaton, and re-writes the program by inserting low-level policy checks to ensure that the policy is obeyed. Our key insight is that methods adapted from abstraction-refinement techniques [6] used in software model checking can be applied to optimize the in-lined code. From a security perspective, our approach means that some programs cannot be verified entirely a priori, but it allows us to ensure that any program that is executed will always satisfy the policy. Additionally, by bounding the size of the abstraction used in the optimization phase, the tradeoff between static analysis complexity and optimality of the in-lined code can be fine-tuned. The simplicity of this approach is attractive, and allows our algorithm to benefit from advances in the state-of-the-art in automatic program abstraction and model checking.

We implemented our approach for JavaScript, and applied it to several real-world applications. We found the abstraction-refinement approach to be effective at reducing the amount of instrumentation code necessary to enforce stateful

policies. In many cases, all of the instrumentation code can be proven unnecessary after the analysis learns a handful (one or two) facts about the program through counterexamples. In such cases, the analysis has established that the program is safe to run as is, and thus there is no runtime overhead. In cases where the program definitely has a policy violation that the analysis uncovers, instrumentation is introduced to exclude that behavior (by causing the program to halt before the policy is violated), again using only a few facts established by static analysis.

To summarize, our contributions are:

- A language-independent algorithm for weaving stateful policies into programs, to produce new programs whose behavior is identical to the original on all executions that do not violate the policy.
- A novel application of traditional software model-checking techniques that uses runtime instrumentation to ensure policy conformity whenever static analysis is too imprecise or expensive. The degree to which the analysis relies on static and dynamic information is tuneable, which provides a trade-off in runtime policy-enforcement overhead.
- A prototype implementation of our algorithm for JavaScript, called JAM, and an evaluation of the approach on real JavaScript applications. The evaluation validates our hypothesis that additional time spent in static analysis, utilizing the abstraction-refinement capabilities of our algorithm, results in fewer runtime checks. For five of our twelve benchmark applications, learning just four predicates allows JAM to place an optimal number of runtime checks necessary to enforce the policy.

The rest of the paper is laid out as follows. Section 2 gives an overview of the algorithm. Section 3 presents the technical details of the analysis and discuss JAM. Section 4 evaluates the performance of JAM over a set of real applications. Section 5 discusses related work.

## 2 Overview

We propose a hybrid approach to enforcing policies over code from an untrusted source. Our solution is to perform as much of the enforcement as possible statically, and to use runtime checks whenever static analysis becomes too expensive. This approach allows us to avoid overapproximations on code regions that are difficult to analyze statically. Furthermore, varying the degree to which the analysis relies on runtime information allows us to control the cost of static analysis at the expense of performing additional runtime checks. While this approach means that many programs cannot be verified against a policy a priori before execution, an interpreter provided with the residual information from the static analysis can prevent execution of any code that violates the policy. In fact, as we show in Section 3, the target program can often be rewritten to in-line any residual checks produced by the static analysis, sidestepping the need for explicit support from the interpreter.

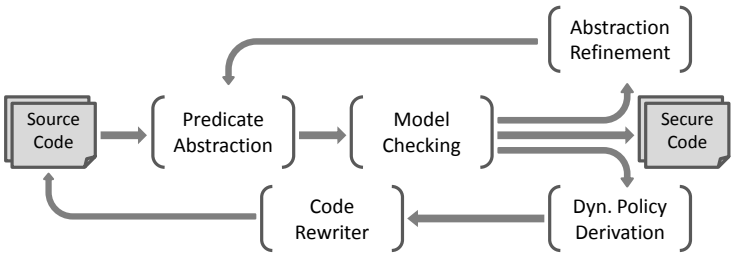



**Fig. 1.** Workflow overview of our approach.

Figure 1 presents an overview of our approach. The security policy is first encoded as a temporal safety property over the states of the target program. The algorithm then begins like other software model checkers by first performing predicate abstraction [12] over the target code, and checking the resulting model for a reachable policy-violating state [20]. Our algorithm differs from previous techniques in how the results of the model checker are used; when the model checker produces an error trace, there are a few possibilities.

1. If the trace is valid, then our algorithm places a dynamic check in the target code to prevent it from being executed on any concrete path.
2. If the trace is not valid, then the algorithm can either:
   (a) Refine the abstraction and continue model checking.
   (b) Construct a dynamic check that blocks execution of the trace *only* when the concrete state indicates that it will violate the policy.

Item (1) has the effect of subtracting a known violating trace from the behaviors of the program, and in general, some number of *similar* behaviors, thereby decreasing the size of the remaining search space. For an individual counterexample, item (2)(a) follows the same approach used in traditional counterexample-guided abstraction refinement-based (CEGAR) software model checking. Item (2)(b) ensures that a potentially-violating trace identified statically is never executed, while avoiding the expense of constructing a proof for the trace. However, the inserted check results in a runtime performance penalty—thus, the choice corresponds to a configurable tradeoff in analysis complexity versus runtime overhead.

To illustrate our approach, consider the program listed in Figure 2(a). This code is a simplified version of the sort of dispatching mechanism that might exist in a command-and-control server [24] or library, and is inspired by common JavaScript coding patterns. The function `execute` takes an instruction code and data argument, and invokes an underlying system API according to a dispatch table created by the program's initialization code.

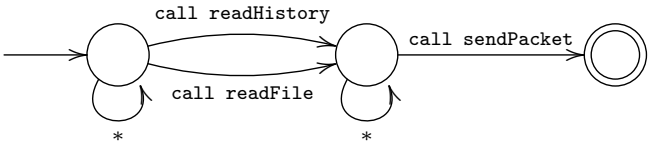
We will demonstrate enforcement of the policy given in Figure 2(c), which is meant to prevent exfiltration of file and browser-history data. Observe that we specify this policy, which corresponds to a temporal safety property, using an automaton that summarizes all of the "bad" paths of a program that might lead to a violation. Thus, policies encode properties on individual paths of a program,

```
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 }
```

**(a)** Original code ($P$).

```
1  policy = 0;
2  api[0] = readFile;
3  api[1] = sendPacket;
4  fun execute(instr, data) {
5      if(api[instr] == readFile
6         && policy == 0) policy++;
7      if(api[instr] == sendPacket
8         && policy == 1) halt();
9      api[instr](data);
10 }
```

**(b)** Safe code (previously safe parts ommitted). Shaded lines contain two checks inserted by our algorithm; our analysis prevented an additional check before line 9.




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

**Fig. 2.** Dangerous code example.

```
api[0] = readFile;
api[1] = sendPacket;
instr, data = read();
execute(0, data);
assume{api[0] == readFile}
api[instr](data);
instr, data = read();
execute(1, data);
assume{api[1] == sendPacket}
api[instr](data);
```

**(1)**

```
api[0] = readFile;
api[1] = sendPacket;
instr, data = read();
execute(0, data);
assume{api[0] == readHistory}
api[instr](data);
instr, data = read();
execute(1, data);
assume{api[1] == sendPacket}
api[instr](data);
```

**(2)**

**Fig. 3.** Counterexamples returned from the example in Figure 2. **(2)** is invalid, and leads to a spurious runtime check.

and we intuitively think of policy violation as occurring when a subsequence of statements in a path corresponds to a word in the language of the policy automaton. The example in Figure 2(c) is representative of the policies used by our analysis, and meant to convey the important high-level concepts needed to understand our approach. For more details about specific policies, see Section 3.

To verify this policy against the program, we proceed initially with software model checking. The first step is to create an abstract model of the program using a finite number of relevant predicates [12]. We begin with three predicates, corresponding to the states relevant to the policy: `@Func = readHistory`, `@Func = readFile`, and `@Func = sendPacket`. We assume the existence of a special state variable `@Func`, which holds the current function being executed; each of these predicates queries which function is currently being executed. With these predicates, the software model checker will return the two counterexamples shown in Figure 3.

**(1)** is valid — it corresponds to a real policy violation. Our analysis updates the program's code by simulating the violating path of the policy automaton over the actual state of the program. This is demonstrated in Figure 2(b); the automaton is initialized on line 1, and makes a transition on lines 6 and 8 after the program's state is checked to match the labels on the corresponding policy transitions. When the transition to the final state is taken, the program halts.

**(2)** is not valid—the assumption that `api[0] == readHistory` never holds. However, for the analysis to prove this, it would need to learn a predicate that encodes this fact, and build a new, refined abstraction on which it can perform a new statespace search. If the user deems this too expensive, then the analysis can simply insert another runtime check before line 7 corresponding to the first transition in the policy automaton, which increments `policy` whenever `api[instr]` holds `readHistory`. This approach will result in a secure program, but will impose an unnecessary runtime cost: every time `execute` is called, this check will be executed but the test will never succeed. Alternatively, the analysis can learn the predicates $\{\texttt{api}[\texttt{instr}] = \texttt{readHistory},\ \texttt{instr} = 0\}$, and proceed with software model checking as described above. This will result in the more efficient code shown in Figure 2(b).

## 3 Technical Description

Our analysis takes a program $\mathcal{P}$ and a policy, and produces a new program $\mathcal{P}'$ by inserting policy checks at certain locations in $\mathcal{P}$ needed to ensure that the policy is not violated. In this section, we describe the policies that we support, as well as the algorithm for inserting policy checks. The algorithm we present has several important properties that make it suitable for practical use:

1. Upon completion, it has inserted a complete set of runtime checks necessary to enforce the policy: any program that would originally violate the policy is guaranteed not to violate the policy after rewriting. Runs that violate the policy must encounter a check, and are halted before the violation occurs.
2. The policy checks inserted will not halt execution unless the current execution of the program will inevitably lead to a policy violation. In other words, *our approach does not produce any false positives.*
3. The running time of the main algorithm is bounded in the size of the program abstraction, which is controllable by the user. This approach yields a trade-off between program running time and static-analysis complexity.
4. JAM always terminates in a finite number of iterations.

We begin with a description of our problem, and proceed to describe our language-independent algorithm for solving it (Section 3.1). The algorithm relies only on standard semantic operators, such as symbolic precondition and abstract statespace search. In Section 3.2, we discuss our implementation of the algorithm for JavaScript.

### 3.1 Runtime Checks for Safety Properties

**Preliminaries.** A run of a program $\mathcal{P}$ executes a sequence of statements, where each statement transforms an element $\sigma \in \Sigma_{\mathcal{P}}$ (or *state*) of $\mathcal{P}$'s state space to a new, not necessarily distinct, state $\sigma'$. For the purposes of this section, we do not make any assumptions about the form of $\mathcal{P}$'s statements or states. We use a labeling function $\iota$ for each statement $s$ in $\mathcal{P}$, so that $\iota(s)$ denotes a unique integer. Let a *state trace* be a sequence of states allowed by $\mathcal{P}$, and let $\mathcal{T}_{\mathcal{P}} \subseteq \Sigma^*$ be the complete set of possible state traces of $\mathcal{P}$.

The policies used by our analysis are based on *temporal safety properties*. A temporal safety property encodes a finite set of sequences of events that are not allowed in any execution of the program. We represent these properties using automata.[1] The events that appear in our policies correspond to concrete program states, and we do not allow non-trivial cycles among the transitions in the automaton.

**Definition 1** *(Temporal safety automaton).* A *temporal safety automaton* $\Phi$ is an automaton $\Phi = (Q, Q_s, \delta, Q_f, \mathcal{L})$ where

- $Q$ is a set of states (with $Q_s \subseteq Q$ and $Q_f \subseteq Q$ the initial and final states, respectively). Intuitively, each $q \in Q$ represents sets of events that have occurred up to a certain point in the execution.
- $\delta \subseteq Q \times \mathcal{L} \times Q$ is a deterministic transition relation that does not contain any cycles except for self-loops.
- $\mathcal{L}$ is a logic whose sentences represent sets of program states, i.e., $\phi \in \mathcal{L}$ denotes a set $[\![\phi]\!] \subseteq \Sigma$.

For a given $(q, \phi, q') \in \delta$, the interpretation is that execution of a statement from a program state $\sigma$ where $\phi$ holds (i.e., $\sigma \in [\![\phi]\!]$) causes the temporal safety automaton to transition from $q$ to $q'$. Self-loops are necessary to cover statements that do not affect the policy state, but other types of cycles can prevent our algorithm from terminating in a finite number of iterations. This leads to the definition of property matching: a program $\mathcal{P}$ **matches** a temporal safety automaton $\Phi$, written $\mathcal{P} \models \phi$, if it can generate a state trace that matches a word in the language of the automaton.

Our problem is based on the notion of property matching. Given a program $\mathcal{P}$ and temporal safety automaton $\Phi$, we want to derive a new program $\mathcal{P}'$ that: (i) does not match $\Phi$, (ii) does not contain any *new* state traces, and (iii) preserves all of the state traces from $\mathcal{P}$ that do not match $\Phi$.

**Policy Checks from Proofs.** Our algorithm is shown in Algorithm 1, and corresponds to the workflow in Figure 1. SafetyWeave takes a program $\mathcal{P}$, a finite set of predicates $E$ from $\mathcal{L}$, a bound on the total number of predicates $k$, and a temporal safety automaton policy $\Phi$. We begin by using predicate set $E$ to build a sound abstraction of $P$ [12] (this functionality is encapsulated by Abs in Algorithm 1). Note that $E$ must contain a certain set of predicates, namely

[1] This formalism is equivalent to past-time LTL.

**Algorithm 1** SafetyWeave($\mathcal{P}, E, k, \Phi$)

**Require:** $k \geq 0$
**Require:** $\phi \in E$ for all $(q, \phi, q') \in \delta$
  **repeat**
    $\mathcal{P}_E \leftarrow$ Abs($\mathcal{P}, E$) {Build abstraction}
    $\pi \leftarrow$ IsReachable($\mathcal{P}_E, \Phi$)
    **if** $\pi =$ NoPath **then**
      **return** $\mathcal{P}$
    **else if** isValid($\pi$) **then**
      {Build runtime policy check; rewrite P to enforce it}
      $\Psi_D^\pi \leftarrow$ BuildPolicy($\mathcal{P}, \pi, \Phi$)
      $\mathcal{P} \leftarrow$ Enforce($\mathcal{P}, \Psi_D^\pi$)
    **else**
      {Refine the abstraction}
      **if** $|E| < k$ **then**
        $E \leftarrow E \cup$ NewPreds($\pi$)
      **else**
        {We have reached saturation of the abstraction set $E$}
        {Build runtime policy check; rewrite P to enforce it}
        $\Psi_D^\pi \leftarrow$ BuildPolicy($\mathcal{P}, \pi, \Phi$)
        $\mathcal{P} \leftarrow$ Enforce($\mathcal{P}, \Psi_D^\pi$)
      **end if**
    **end if**
  **until forever**

$\phi_i$ for each $(q_i, \phi_i, q_i') \in \delta^\Phi$. The abstraction is then searched for traces from initial states to bad states (encapsulated by IsReachable), which correspond to final states in $\Phi$. If such a trace $\pi$ is found, it is first checked to see whether it corresponds to an actual path through $\mathcal{P}$ (performed by IsValid). If it does, or if we cannot build an abstraction that does not contain $\pi$, then a runtime policy check $\Psi_D^\pi$ is derived (encapsulated by BuildPolicy) and added to $\mathcal{P}$ (performed by Enforce). $\Psi_D^\pi$ identifies a concrete instance of $\pi$.

If $\pi$ does not correspond to an actual policy-violating path of $\mathcal{P}$, and we have fewer than $k$ predicates, then the abstraction is refined by learning new predicates (encapsulated by NewPreds). Otherwise, we add a runtime check to prevent the concrete execution of $\pi$. This process continues until we have either proved the absence of violating paths (via abstraction refinement), or added a sufficient set of runtime checks to prevent the execution of possible violating paths.

**Termination.** SafetyWeave is guaranteed to terminate in a finite number of iterations, due to the following properties: *(i)* the algorithm will stop trying to prove or disprove the validity of a single trace after a finite number of iterations, due to the bounded abstraction size ($|E|$ is limited by $k$). *(ii)* In the worst case, it must insert a policy check for each transition in $\Phi$ before every statement in $\mathcal{P}$. Once $\mathcal{P}$ is thus modified, IsReachable will not be able to find a violating trace $\pi$, and will terminate.

**Abstracting $\mathcal{P}$ (Abs).** On each iteration, an abstraction $\mathcal{P}_E$ is built from the predicate set $E$ and the structure of $\mathcal{P}$. $\mathcal{P}_E$ has two components: a control automaton $G_C$ and a data automaton $G_D$. Each automaton is a nested word automaton (NWA) [2] whose alphabet corresponds to the set of statements used in $\mathcal{P}$. $G_C$ overapproximates the set of all paths through $\mathcal{P}$ that are valid with respect to $\mathcal{P}$'s control structure (i.e., call/return nesting, looping, etc.), whereas $G_D$ overapproximates the paths that are valid with respect to the data semantics of $\mathcal{P}$. In $G_C$, states correspond to program locations, and each program location corresponds to the site of a potential policy violation, so each state is accepting. In the data automaton, states correspond to sets of program states, and transitions are added according to the following rule: given two data-automaton states $q$ and $q'$ representing $\phi$ and $\phi'$, respectively, the $G_D$ contains a transition from $q$ to $q'$ on statement $s$ whenever $\phi \wedge \mathsf{Pre}(s, \phi')$ is satisfiable, where $\mathsf{Pre}$ is the symbolic precondition operator. *We then have that $L(G_C) \cap L(G_D)$ represents an overapproximation of the set of valid paths through $\mathcal{P}$; this is returned by* Abs.

Separating the abstraction into $G_C$ and $G_D$ allows us to provide a straightforward, well-defined interface for extending the algorithm to new languages. To be able to instantiate Algorithm 1 to work on programs written in a different language, a tool designer need only provide (i) a symbolic pre-image operator for that language to build $G_D$, and (ii) a generator of interprocedural control-flow graphs (ICFGs) to build $G_C$.

**Proposition 1** *$L(G_D)$ corresponds to a superset of the traces of $\mathcal{P}$ that might match $\Phi$.*

**Checking the Abstraction (IsReachable).** Given an automaton-based abstraction $\mathcal{P}_E = G_C \cap G_D$, IsReachable finds a path in $\mathcal{P}_E$ that matches $\Phi$. This operation is encapsulated in the operator $\cap_{\text{pol}}$ specified in Definition 2. In essence, Definition 2 creates the product of two automata—$\mathcal{P}_E$ and $\Phi$. However, the product is slightly non-standard because $\mathcal{P}_E$ has an alphabet of program statements, whereas $\Phi$ has an alphabet of state predicates. Note that when we refer to the states of $\mathcal{P}_E$ in Definition 2, we abuse notation slightly by only using the program state component from $G_D$, and dropping the program location component from $G_C$.

**Definition 2** *(Policy Automaton Intersection $\cap_{\text{pol}}$).* Given a temporal safety automaton $\Phi = (Q^\Phi, Q^\Phi_s, \delta^\Phi, Q^\Phi_f)$ and an NWA $G = (Q^G, Q^G_s, \delta^G, Q^G_f)$ whose states correspond to sets of program states, $G \cap_{\text{pol}} \Phi$ is the nested word automaton $(Q, Q_s, \delta, Q_f)$, where

- $Q$ has one element for each element of $Q^G \times Q^\Phi$.
- $Q_s = \{(\phi, q^\Phi) \mid q^\Phi \in Q^\Phi_s, \phi \in Q^G_s\}$, i.e., an initial state is initial in both $G_D$ and $\Phi$.
- $\delta = \langle \delta_{\mathsf{in}}, \delta_{\mathsf{ca}}, \delta_{\mathsf{re}} \rangle$ are the transition relations with alphabet $\mathcal{S}$. For all $(q^\Phi, \phi'', q'^\Phi) \in \delta^\Phi$, and $\phi, \phi' \in Q^G$ such that $\phi \wedge \mathsf{Pre}(s, \phi' \wedge \phi'')$ is satisfiable, we define each transition relation using the transitions in $\delta^G = (\delta^G_{\text{in}}, \delta^G_{\text{ca}}, \delta^G_{\text{re}})$:
  - $\delta_{\text{in}}$: when $(\phi, s, \phi') \in \delta^G_{\text{in}}$, we update $\delta_{\mathsf{in}}$ with: $((\phi, q^\Phi), s, (\phi', q'^\Phi))$.

- $\delta_{\text{ca}}$: when $(\phi, s, \phi') \in \delta_{\text{ca}}^G$, we update $\delta_{\mathsf{ca}}$ with: $((\phi, q^\Phi), s, (\phi', q'^\Phi))$.
- $\delta_{\text{re}}$: when $(\phi, \phi''', s, \phi') \in \delta_{\text{re}}^G$, we update $\delta_{\mathsf{re}}$ with: $((\phi, q^\Phi), (q'''^\Phi, \phi'''), s, (\phi', q'^\Phi))$ for all $q'''^\Phi \in Q^\Phi$.

– $Q_f = \{(\phi, q^\Phi) \mid q^\Phi \in Q_f^\Phi, \phi \in Q_f^G\}$, i.e., a final state is final in $\Phi$ and $G$.

The words in $L(\mathcal{P}_E \cap_{\text{pol}} \Phi)$ are the sequences of statements (traces) in $\mathcal{P}$ that respect the sequencing and nesting specified in the program, and may lead to an error state specified by $\Phi$. As long as $G_C$ and $G_D$ overapproximate the valid traces in $\mathcal{P}$, we are assured that if an erroneous trace exists, then it will be in $L(\mathcal{P}_E \cap_{\text{pol}} \Phi)$. Additionally, if $L(\mathcal{P}_E \cap_{\text{pol}} \Phi) = \varnothing$, then we can conclude that $\mathcal{P}$ cannot reach an error state.

**Checking Path Validity (IsValid); Refining the Abstraction (NewPreds).** Given a trace $\pi \in L(\mathcal{P}_E)$, we wish to determine whether $\pi$ corresponds to a possible path through $\mathcal{P}$ (i.e., whether it is *valid*). This problem is common to all CEGAR-based software model checkers [3, 16], and typically involves producing a formula that is valid iff $\pi$ corresponds to a real path. We discuss an implementation of IsValid for JavaScript in Section 3.2.

Because $\mathcal{P}_E$ overapproximates the error traces in $\mathcal{P}$, two conditions can hold for a trace $\pi$. *(i)* The sequence of statements in $\pi$ corresponds to a valid path through $\mathcal{P}$ that leads to a violation according to $\Phi$, or it cannot be determined whether $\pi$ is a valid trace or not. *(ii)* The sequence of statements in $\pi$ can be proven to be invalid. In the case *(i)*, a runtime check is added to $\mathcal{P}$ to ensure that the error state is not entered at runtime (see the following section for a discussion of this scenario). In the case of *(ii)*, $\mathcal{P}_E$ is refined by adding predicates to $G_D$ (encapsulated in the call to NewPreds). Standard techniques from software model checking may be applied to implement NewPreds, such as interpolation [25] and unsatisfiable-core computation [16]; we discuss our JavaScript-specific implementation in Section 3.2.

**Deriving and Enforcing Dynamic Checks.** The mechanism for deriving dynamic checks that remove policy-violating behavior is based on the notion of a *policy-violating witness*. A policy-violating witness is computed for each counterexample trace produced by the model checker that is either known to be valid, or cannot be validated using at most $k$ predicates. A policy-violating witness must identify at runtime the concrete instance of the trace $\pi$ produced by the model checker before it violates the policy $\Phi$. To accomplish this, we define a policy-violating witness as a sequence containing elements that relate statements to assertions from $\Phi$. The fact that a check is a sequence, as opposed to a set, is used in the definition of Enforce.

**Definition 3** *(Policy-violating witness).* A *policy-violating witness* $\Psi_\Phi^\pi \in (\mathbb{N} \times \mathcal{L})^*$ for a trace $\pi$ and policy $\Phi$ is a sequence of pairs relating statement elements in $\pi$ to formulas in $\mathcal{L}$. We say that $\pi' \models \Psi_\Phi^\pi$ (or $\pi'$ *matches* $\Psi_\Phi^\pi$) if there exists a subsequence $\pi''$ of $\pi'$ that meets the following conditions:

1. The statements in $\pi''$ match those in $\Psi_\Phi^\pi$: $|\pi''| = |\Psi_\Phi^\pi|$, and for all $(i, \phi_i) \in \Psi_\Phi^\pi$, $\iota^{-1}(i)$ is in $\pi''$.

2. Immediately befre $\mathcal{P}$ executes a statement $s$ corresponding to the $i^{th}$ entry of $\Psi^{\pi}_{\Phi}$ (i.e. $(\iota(s), \phi_i)$), the program state satisfies $\phi_i$.

Suppose that $\Phi = (Q^{\Phi}, Q^{\Phi}_i, \delta^{\Phi}, Q^{\Phi}_f)$ is a temporal safety automaton, and $\pi$ is a path that causes $\mathcal{P}$ to match $\Phi$. Deriving $\Psi^{\pi}_{\Phi}$ proceeds as follows: because $\pi$ is a word in $L(\mathcal{P}_E = G_D \cap G_C)$, there must exist some subsequence $s_{i_1} s_{i_2} \ldots s_{i_m}$ of $\pi$ that caused transitions between states in $\Phi$ that represent distinct states in $\Phi$. We use those statements, as well as the transition symbols $[\phi_i]_{i \in \{i_1, i_2, \ldots, i_m\}}$ from $\Phi$ on the path induced by those statements, to build the $j^{th}$ element of $\Psi^{\pi}_{\Phi}$ by forming pairs $(i_j, \phi_j)$, where the first component ranges over the indices of $s_{i_1} s_{i_2} \ldots s_{i_m}$.

More precisely, for all $i \in i_1, i_2, \ldots, i_m$, there must exist $(q_i, \phi_i, q'_i) \in \delta^{\Phi}$ and $((\phi, q_i), s, (\phi', q'_i)) \in \delta^{\mathcal{P}_E \cap_{\text{pol}} \Phi}$ such that $\phi' \wedge \phi_i$ is satisfiable (recall the $\cap_{\text{pol}}$ from Definition 2). Then:

$$\Psi^{\pi}_{\Phi} = [(i_{i_1}, \phi_1), (i_{i_2}, \phi_2), \ldots, (i_{i_m}, \phi_m)]$$

Intuitively, $\Psi^{\pi}_{\Phi}$ captures the statements in $\pi$ responsible for causing $\Phi$ to take transitions to its accepting state, and collects the associated state assertions to form the policy-violating witness.

We now turn to Enforce, which takes a policy-violating witness $\Psi^{\pi}_{\Phi}$, and a program $\mathcal{P}$, and returns a new program $\mathcal{P}'$ such that $\mathcal{P}'$ does not contain a path $\pi$ such that $\pi \models \Psi^{\pi}_{\Phi}$. The functionality of Enforce is straightforward: for each element $(i, \phi)$ in $\Psi^{\pi}_{\Phi}$, insert a guarded transition immediately before $\iota^{-1}(i)$ to ensure that $\phi$ is never true after executing $\iota^{-1}(i)$. The predicate on the guarded transition is just the negation of the precondition of $\phi$ with respect to the statement $\iota^{-1}(i)$, and a check that the *policy variable* (inserted by Enforce) matches the index of $(i, \phi)$ in $\Phi$. When the guards are true, the statement either increments the policy variable, or halts if the policy variable indicates that all conditions in $\Psi^{\pi}_{\Phi}$ have passed. A concrete example of this instrumentation in Figure 2.

Note that a given occurrence of statement $s$ in $\mathcal{P}$ may be visited multiple times during a run of $\mathcal{P}$. Some subset of those visits may cause $\Phi$ to transition to a new state. In this scenario, notice that our definition of Enforce will insert multiple guarded transitions before $s$, each differing on the condition that they check—namely, each transition $(q, \phi, q')$ of $\Phi$ that was activated by $s$ in the policy-violating witness will have a distinct condition for $\mathsf{Pre}(s, \phi)$ that either increments the policy variable or halts the program. Additionally, the check on the policy variable in each guard prevents the policy variable from being updated more than once by a single check.

**Definition 4** *(Functionality of* Enforce*).* Given a program $\mathcal{P}$ and a dynamic check $\Psi^{\pi}_{\Phi} = \{(i_1, \phi_1), \ldots, (i_n, \phi_n)\}$, 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 $(i, \phi) \in \Psi^{\pi}_{\Phi}$:

1. Let $\phi_{\text{pre}} \equiv \mathsf{Pre}(\iota^{-1}(i), \phi) \wedge \texttt{policy} = j$, where $j$ is the index of $(i, \phi)$ in $\Phi^{\pi}_{\Phi}$.
2. Insert a new statement before $\iota^{-1}(i)$ that either:

– Increments `policy` whenever $\phi_{\text{pre}}$ is true and `policy` $< |\Psi^{\pi}_{\Phi}|$.
– Halts the execution of $\mathcal{P}'$ whenever $\phi_{\text{pre}}$ is true and `policy` $= |\Psi^{\pi}_{\Phi}|$.

For Enforce to operate correctly, $\mathcal{L}$ must be closed under the computation of pre-images, and pre-images of formulas in $\mathcal{L}$ must be convertible to code in the target language. When Enforce is called on all counterexample paths returned by Algorithm 1, the resulting program will not match $\Phi$.

### 3.2 JavaScript Prototype

We implemented our algorithm for JavaScript, in a tool called JAM. There are four components to Algorithm 1 that must be made specific to JavaScript: the control ($G_C$) and data ($G_D$) automaton generators (Abs), the path validity checker (IsValid), and the predicate learner (NewPreds). To build the control automaton, we used Google's Closure Compiler [17], which contains methods for constructing an intraprocedural control flow graph (CFG) for each function in a program, as well as dataflow analyses for determining some of the targets of indirect calls. The only language-specific aspect of the data-automaton generator is the computation of symbolic pre-state for a given statement in $\mathcal{P}$. We use Maffeis et al.'s JavaScript operational semantics [21], lifted to handle symbolic term values, and implemented as a set of Prolog rules. Computing a satisfiability check to build $G_D$ in this setting amounts to performing a query over this Prolog program, with ground state initialized to reflect the initial state of the program. To learn new predicates (i.e., to compute NewPreds), we apply a set of heuristics to the failed counterexample trace that we have developed from our experience of model checking real JavaScript programs. Our heuristics are based on the statement that made the trace invalid; the predicate they build depends on the type of that statement (e.g., if the statement is an `if` statement, the new predicate will be equivalent to the statement's guard expression).

Currently, the JAM implementation does not handle programs that contain dynamically generated code—e.g., generated via language constructs, such as eval() or Function(), or via DOM interfaces, such as document.write(). JAM currently only handles a subset of the DOM API that most browsers support. None of these are fundamental limitations, although supporting dynamically generated code soundly could cause a large number of runtime checks to be introduced. Dynamically generated code can be supported by inserting code that updates the state of the `policy` variable (Definition 4) by simulating the policy automaton before each dynamically generated statement, in the manner of Erlingsson *et al.* [8]. Additional DOM API functions can be supported by adding reduction rules to our semantics that capture the behavior of the needed DOM API.

## 4 Experimental Evaluation

In this section, we summarize the performance and effectiveness of JAM in applying realistic security policies to ten JavaScript applications (plus alternative

versions of two of them that we seeded with policy-violating code). The results, summarized in Table 1, demonstrate that the time devoted to static analysis during the abstraction-refinement stage often leads to fewer runtime checks inserted into the subject program. Additionally, because the CEGAR process evaluates the validity of the potentially-violating execution traces found in the abstract model, time spent during this stage also yields greater confidence that the inserted checks are legitimately needed to prevent policy violations during runtime.

The benchmark applications used to measure JAM's performance are real programs obtained from the World Wide Web. Consequently, the policies we developed typically address cross-domain information-leakage issues and data-privacy issues that are of concern in that domain. Our measurements indicate that under such realistic circumstances, (i) JAM is able to identify true vulnerabilities while (ii) reducing spurious dynamic checks, and (iii) is able to do so with analysis times that are not prohibitive.

### 4.1 Results

Because the goal of the system is to derive a minimal set of runtime checks needed to ensure adherence to a policy, we sought to measure the benefits of refining the program model against the cost of performing such analysis. This information was gathered by comparing the running time and result of JAM's analysis under varying levels of abstraction refinement, achieved by placing a limit on the number of predicates learned during the CEGAR analysis before proceeding to the saturation phase. The validation of counterexamples and learning of new predicates can be disabled altogether, which establishes the baseline effectiveness of static analysis without abstraction refinement. Measurements of JAM's effectiveness and efficiency with different levels of abstraction refinement are presented in Table 1.

One dimension on which to evaluate the behavior of JAM is the number of necessary versus spurious checks that it inserts. All checks that are inserted during the CEGAR phase are known to be necessary, because the abstract counterexample that gave rise to each such check has been proven valid. In contrast, spurious checks may be inserted in the saturation phase. We inspected the applications manually to determine the number of necessary checks Columns 5 and 6 of Table 2 classify the checks identified during saturation as valid or spurious according to our manual classification. A lower number of spurious checks inserted under a particular configuration represents a more desirable outcome vis a vis minimizing runtime overhead.

Reported performance statistics are the averages of multiple runs on a VirtualBox VM running Ubuntu 10.04 with a single 32-bit virtual processor and 4GB memory. The host system is an 8-core HP Z600 workstation with 6GB memory running Red Hat Enterprise Linux Server release 5.7. Execution time and memory usage refer to the total CPU time and maximum resident set size as reported by the GNU time utility version 1.7.

The results for `flickr` demonstrate the benefit of additional effort spent on abstraction refinement. Analysis of the unrefined model identifies two potential

| Benchmark application | Predicates | | Checks | | | | Execution time (s) | Memory (KB) |
|---|---|---|---|---|---|---|---|---|
| | Learned | Total | CEGAR | Saturation | | Total | | |
| | | | | Valid | Spurious | | | |
| flickr | 2 | 3 | 1 | 0 | 0 | 1 | 138.67 | 60737 |
| flickr | 1 | 2 | 1 | 0 | 1 | 2 | 74.49 | 61472 |
| flickr | 0 | 1 | 0 | 1 | 1 | 2 | 24.23 | 63520 |
| beacon | 0 | 3 | 2 | 0 | 0 | 2 | 74.50 | 62215 |
| jssec | 1 | 2 | 0 | 0 | 0 | 0 | 14.04 | 46591 |
| jssec | 0 | 1 | 0 | 0 | 1 | 1 | 7.59 | 56023 |

**Table 1.** Performance of JAM on selected benchmarks. *Learned* denotes the number of predicates learned through abstraction refinement, *Total* to the number of learned predicates plus those in the initial state from the policy. *CEGAR* denotes the number of checks placed before the abstraction size limit is reached, *Saturation* to those placed afterwards.

violations of the policy, one of which is spurious and the other valid (according to our manual classification of checks). When allowed to learn a single predicate, JAM is able to avoid a spurious trace, and identify the valid counterexample. Allowing JAM to learn two predicates causes it to prove the spurious counterexame invalid, and rule out the un-needed runtime check.

The policy for the `beacon` benchmark is more involved—using multiple transition sequences to characterize the policy violation; it states "a cookie should never be written after the DOM is inspected using `document.getElementById` or `document.getElementsByTagName`." This policy represents a cross-domain information-leakage concern that JAM is able to identify and validate in the first iteration of the analysis. The `jssec` application is intended to allow a website user to open and close a section of the page being viewed. The policy for `jssec` states that the only allowable change to a DOM element's style properties is to the `display` attribute; otherwise, the code could change the `backgroundImage` attribute, thereby initiating an HTTP call to a remote server. JAM successfully proves that the program is free of violations by learning the prototype of an object whose member is the target of an assignment.

## 5 Related Work

*In-Lined Reference Monitors.* In-lined reference monitors were first discussed by Erlingsson and Schneider [8, 28] who applied the idea to both Java and x86 bytecode. Their prototype, SASI, supports security policies as finite-state machines with transitions denoting sets of instructions (i.e., predicates over instructions) that may be executed by the untrusted program. Note the distinction from the policy automata used in our work, where transitions have predicates that refer to the program state, *not* just restrictions on the next instruction to execute. SASI works by inserting policy-automaton-simulation code before every instruction in the program, and then uses local simplification to remove as much of the added code as possible. This amounts to applying the available local static information at each location to evaluate the instruction predicate to the greatest

degree possible; the authors opted not to use global static analysis in the interest of maintaining a small TCB. In this respect, the primary focus of our work is quite different from Erlingsson and Schneider's foundational work.

Since Erlingsson and Schneider's work, this has been an active area of research. Nachio [9] is an in-lined-monitor compiler for C, where policies are given as state machines with fragments of imperative code that execute at each state. The Java-MOP (Monitor-Oriented Programming) system [5] allows users to choose from a set of temporal logics, domain-specific logics, and languages in which to express policies. ConSpec [1] performs in-lined reference monitoring based on policies similar to those used by Erlingsson and Schneider, and takes the additional step of formally verifying the in-lined monitor. SPoX [14] built on aspect-oriented programming to implement in-lined reference monitoring for Java, using as policies automata whose edges are labeled with pointcut expressions. They define a formal semantics for their policies, laying the groundwork for future work on verified implementations of in-lined reference monitors; this feature can also aid in developing analyses for optimizing the in-lined monitor code, although the authors do not pursue this idea. Sridhar and Hamlen [29] designed an IRM-compiler for JavaScript bytecodes, and showed how software model checking can be applied to verify the compiled in-lined monitor code. Hamlen et al. [15] designed MOBILE, an extension to the .NET runtime that supports IRMs with the advantage that well-typed MOBILE code is guaranteed to satisfy the policy it purports to enforce. The primary difference between these previous efforts and our own is our focus on optimizing in-lined monitor code, and our use of abstraction-refinement techniques to do this in a tuneable manner.

Clara [4] is a framework for incorporating static analysis into the reference-monitor in-lining process. The setting in which Clara operates is similar to ours: an untrusted program and a security policy, represented by a finite-state machine, are provided, and the goal is to produce a rewritten program that always obeys the policy. It works on top of an aspect-weaving framework for Java [18] by first weaving the policy (represented as an aspect) into the program, and subsequently applying a modular set of static analyses to remove as many join points as possible. In this regard, Clara is conceptually similar to our work; it is conceivable that parts of our work could be combined as a path-sensitive, semantics-driven static-analysis component inside of Clara's modular framework. Otherwise, our work differs from Clara in one important respect: the policies we use provide direct means to refer to the dynamic state of the program, allowing richer and more concise policies. Clara's dependence on AspectJ limits the building blocks of expressible policies to a pre-defined set of pointcuts.

*JavaScript Policy Enforcement.* Several recent projects attempt to identify subsets of JavaScript that are amenable to static analysis. Two early examples are ADSafe [7] and FBJS [10], which facilitate "mashups" by removing language elements that make it difficult to isolate the effects of distinct JavaScript programs executing from the same domain. Maffeis *et al.* explored a similar approach [22, 23], but took the additional step of formally verifying their subsets against small-step operational semantics of the ECMAScript specification. More

recently, Google has released Caja [11], uses the object-capability model to provide isolation. Our work differs from efforts to identify secure JavaScript subsets for isolation primarily in the class of policies we are able to support. Rather than sandbox-based object-capability policies, JAM can verify arbitrary safety properties, including flow-sensitive temporal-safety properties.

Guarnieri and Livshits presented GATEKEEPER, a "mostly static" JavaScript analysis based on points-to information that is calculated using Datalog inference rules [13]. Unlike JAM, Gatekeeper is not capable of checking flow-insensitive policies, and it is not clear how it can be made flow-sensitive without greatly increasing cost. Kudzu [27] is a JavaScript bug-finding system that uses forward-symbolic execution. This functionality stands in contrast to JAM, as dangerous program paths are reported to the user at analysis time, whereas in JAM they are rewritten to halt at runtime before the dangerous (policy-violating) payload is executed: JAM always inserts sufficient instrumentation to soundly and completely enforce a given policy.

Yu *et al.* proposed a safe browsing framework based on syntax-directed rewriting of the JavaScript source according to an edit automaton [30]. Their work is formalized in terms of a JavaScript subset they call *CoreScript*, which excludes the same difficult language elements as most other static JavaScript analyses. While our current implementation does not support the full language either, this is not a limitation of our approach. The dynamic compoment of our policy-enforcement method is capable of monitoring the execution of these language elements. The syntax-directed nature of their rewriting framework effectively restricts the class of policies it can enforce. More recently, Meyerovich and Livshits presented ConScript [26], which is an in-browser mechanism for enforcing fine-grained security policies for JavaScript applications. One of the primary contributions of ConScript is a type system for checking policy-instrumentation code against several different types of attack *on the integrity of the policy*. Essentially, ConScript is a system for specifying and implementing advice [19] on JavaScript method invocations. Thus, ConScript is complementary in function to JAM: while JAM takes a high-level logical formula that represents a security policy, and finds a set of program locations to place policy instrumentation, ConScript is capable of soundly and efficiently enforcing that instrumentation on the client side, during execution.

## References


1. I. Aktug and K. Naliuka. Conspec – a formal language for policy specification. *ENTCS*, 197, February 2008.
2. R. Alur and P. Madhusudan. Adding nesting structure to words. *JACM*, 56(3), 2009.
3. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.
4. E. Bodden, P. Lam, and L. Hendren. Clara: a framework for statically evaluating finite-state runtime monitors. In *RV*, 2010.

5. F. Chen and G. Roşu. Java-MOP: A monitoring oriented programming environment for java. In *TACAS*, 2005.
6. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *JACM*, 50(5), 2003.
7. D. Crockford. Adsafe: Making JavaScript safe for advertising. http://www.adsafe.org.
8. U. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *NSPW*, 2000.
9. D. Evans and A. Twyman. Flexible policy-directed code safety. *SP*, 1999.
10. Facebook, Inc. FBJS. http://wiki.developers.facebook.com/index.php/FBJS.
11. Google inc. The Caja project. http://code.google.com/p/google-caja/.
12. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
13. S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Security*, Aug. 2009.
14. K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *PLAS*, 2008.
15. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *PLAS*, 2006.
16. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
17. G. Inc. Closure Compiler. http://code.google.com/closure/compiler/.
18. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, 2001.
19. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.
20. S. Kiefer, S. Schwoon, and D. Suwimonteerabuth. Moped: A model checker for pushdown systems. http://www.fmi.uni-stuttgart.de/szs/tools/moped/.
21. S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, 2008.
22. S. Maffeis and A. Taly. Language-based isolation of untrusted Javascript. In *CSF*, 2009.
23. S. Maffeis and J. M. A. Taly. Language-based isolation of untrusted JavaScript. In *SP*, 2010.
24. L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. In *RAID*, 2008.
25. K. L. McMillan. Applications of Craig interpolants in model checking. In *TACAS*, 2005.
26. L. Meyerovich and B. Livshits. Conscript: Specifying and enforcing fine-grained security policies for javascript in the browser. In *SP*, 2010.
27. P. Saxena, D. Akhawe, S. Hanna, S. McCamant, F. Mao, and D. Song. A symbolic execution framework for JavaScript. In *SP*, 2010.
28. F. B. Schneider. Enforceable security policies. *TISSEC*, 3, February 2000.
29. M. Sridhar and K. W. Hamlen. Model-checking in-lined reference monitors. In *VMCAI*, 2010.
30. D. Yu, A. Chander, N. Islam, and I. Serikov. JavaScript instrumentation for browser security. In *POPL*, 2007.