Efficient IRM Enforcement of History-Based Access Control Policies

Fei Yan and Philip W. L. Fong
Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada
{ feiya200, pwlfong }@cs.uregina.ca

ABSTRACT

Inlined Reference Monitor (IRM) is an established enforcement mechanism for history-based access control policies. IRM enforcement injects monitoring code into the binary of an untrusted program in order to track its execution history. The injected code denies access when execution deviates from the policy. The viability of IRM enforcement is predicated on the ability of the binary rewriting element to optimize away redundant monitoring code without compromising security.

This work proposes a novel optimization framework for IRM enforcement. The scheme is based on a constrained representation of history-based access control policies, which, despite its constrained expressiveness, can express such policies as separation of duty, generalized Chinese Wall policies, and hierarchical one-out-of-k authorization. An IRM optimization procedure has been designed to exploit the structure of this policy representation. The optimization scheme is then extended into a distributed optimization protocol, in which an untrusted code producer attempts to help boost the optimization effectiveness of an IRM enforcement mechanism administered by a distrusting code consumer. It is shown that the optimization procedure provably preserves security even in the midst of distributed optimization. A prototype of the optimization procedure has been implemented for Java bytecode, and its effectiveness has been empirically profiled.

Categories and Subject Descriptors

D.2.0 [**Software Engineering**]: General—*protection mechanisms*; D.3.4 [**Programming Languages**]: Processors—*code generation, optimization*; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*

General Terms

Security, Languages, Verification, Performance

ASIACCS '09, March 10–12, 2009, Sydney, NSW, Australia

Keywords

Language-based security, history-based access control policies, inlined reference monitors, security automata, distributed optimization protocol.

1. INTRODUCTION

This paper presents novel implementation techniques for the protection mechanism of extensible systems, that is, software systems composed of a trusted application core collaborating with a number of untrusted software components, all running within the same address space. To support the late binding of features to an application, the latter could be made extensible by adopting a plug-in architecture or offering scripting support. This paper focuses on language-based extensible systems [30] such as those developed on the safe language environments Java and .Net. In these systems, untrusted components collaborate with the application core through a well-defined Application Programming Interface (API). To protect the integrity of the resources encapsulated by the API, it is in the interest of the application core to ensure that access requests made by the untrusted components through the API honor certain security policies. A notable such family of security policies are history-based access control policies, also known as safety properties in the literature [29]. To enforce these policies, authorization decisions are made solely on the basis of the execution history of the target program as observed by the enforcement mechanism at run time. Examples of such policies include the Chinese Wall policy [10], Biba's low water mark policy [8], one-out-of-k authorization [14], assured pipelines [9], as well as Stack Inspection [38] and its variants [1].

Execution monitoring [14, 15, 40] is the standard enforcement mechanism for history-based access control policies. The classical implementation strategy is to interpose a reference monitor at the entry points of the API, so that the monitor may track the API calls previously made, arguments passed, or even the run-time state of the untrusted component to ensure policy compliance. This is the implementation strategy adopted by the Java platform in its Stack Inspection mechanism [18]. A modern implementation strategy for execution monitoring is Inlined Reference Monitor (IRM) [36], in which monitoring code is injected into an untrusted component through binary rewriting. The advantage of IRM over interpositioning is that IRM fully decouples the enforcement mechanism from the application core, thereby allowing the security model to evolve separately from the application code base. An important challenge faced by IRM enforcement mechanisms is the run-time

overhead induced by the injected code [38, 29]. Viability of the IRM approach is predicated on the ability of the binary rewriting element to optimize away unnecessary monitoring code [12].

In this work, we explore the interplay between security concerns and optimization procedures for IRM enforcement of history-based access control policies. Our contribution is twofold:

1. **Optimization-friendly policy representation**: Since [29, 35, 36], the Security Automaton has become the standard representation for security policies to be enforced by execution monitoring. A research concern [5, 17, 22, 33, 34] of the language-based security community has been the following: Can we trade off the expressiveness of policy representation (i.e., by adopting a less powerful version of the Security Automaton) for improved resource consumption (e.g., time, space, information) of the execution monitor? In this work, we address a related but novel research question: *Can we trade off the expressiveness of policy representation for improved effectiveness of the optimization element in the IRM binary rewriter?* By adopting a declarative state representation and imposing structures on state transitions, we have shown that one can employ standard compiler optimization techniques to optimize away state transition code that would otherwise be injected into the target program, and do so without compromising security. We also demonstrate that the resulting policy representation is still expressive enough to encode a wide range of history-based access control policies.

2. **Distributed optimization protocol**: To further enhance the effectiveness of IRM optimization, we propose a distributed optimization protocol that has been inspired by Proof-Carrying Code [24]. Specifically, an untrusted code producer sends a software component to a distrusting code consumer for execution. To promote usage of the component, the code producer ships a version of the component that has been annotated with optimization directives, which are hints on how the code consumer can aggressively optimize the monitor code to be injected into the component for IRM enforcement. As the code producer could very well be malicious, blindly following the optimization directives could lead to the omission of key monitoring logic, thus compromising security. To counter this, the code consumer injects into the component specially designed run-time checks that will be completely optimized away if the code producer is honest about the optimization directives, but will detect the dishonesty at run time if the code producer attempts to mislead the code consumer.

The rest of this paper outlines the proposed policy representation (Sect. 3), the optimization procedure that takes advantage of this policy representation (Sect. 4), a corresponding distributed optimization protocol (Sect. 5), as well as an implementation (Sect. 6) and its empirical evaluation (Sect. 7).

2. RELATED WORK

What we call history-based access control policies are also known in the literature as safety properties[1]. Schneider characterized the security policies enforceable by execution monitoring to be safety properties [29], and proposed Security Automata (SA) as the standard representation of execution monitors. (A recent sharpening of this result can be found in [19].) Inlined Reference Monitoring was first proposed in [36] as a framework to unify previous work [14, 15] that employs binary rewriting to enforce history-based access control policies. Fong proposed an information-based characterization of security policies enforceable by execution monitors consuming only a limited portion of history information [17]. The goal was to understand the trade-off between the differentiating power of an execution monitor and the resource to which it is made available, a goal first articulated by Ligatti *et al* [5, 22]. The work has been refined by Talhi *et al* to obtain a characterization of execution monitors operating under memory constraints [33, 34]. Our work poses a related but novel question: can the expressiveness of policy representation be restricted to facilitate IRM optimization? Our policy representation is formally akin to STRIPS planning operators [16].

A first principled design of optimization procedures for IRM enforcement mechanisms is [12], which assumes each transition has a constant cost. Our optimization procedure is designed for unbounded state space, and thus we adopted a different performance metric (see Sect. 4). As IRM enforcement could be seen as a special-case of Aspect-Oriented Programming (AOP) [21], previous work on optimization techniques for AOP languages (e.g., [4]) is also relevant. Our work is unique in that we facilitate optimization by trading off policy expressiveness and by adopting a distributed optimization protocol.

Proof-Carrying Code (PCC) [24] pioneered the idea of self-certifying code. Specifically, a proof of safety is shipped along with an untrusted program, allowing the code consumer to verify safety in a tractable manner. Rose and Rose proposed a lightweight Java bytecode verification framework [27], in which type states are shipped along with Java class-files, so that bytecode verification can be performed more efficiently. In model-carrying code [31], the code producer ships an untrusted program together with its behavior model. The model is checked by the code consumer against a preset policy for compliance. The verified model is then employed to monitor the execution of the untrusted program. In [2], a PCC-style safety proof is attached to an untrusted program to certify that an execution monitor has been properly inlined. Compared to the work above, ours is unique in that it is the first to propose annotations for facilitating IRM optimization rather than verification.

CMV [32] is a model checker for verifying complete mediation [28] in the Stack Inspection enforcement mechanism of a Java Virtual Machine (JVM) implementation. Our work could be seen as a generalization of the static analysis performed by CMV to (1) support a more general class of safety properties and (2) inject dynamic checks when a target property cannot be statically verified. Both systems employ a notion of method interfaces (called method summaries in

[1]We adopt the nomenclature of [14], and use the term "history-based access control" to refer to execution monitoring in general. Recently, some authors (e.g., [39]) use the term to refer to a variant of Stack Inspection [38] proposed by Abadi and Fournet [1]. We deviate from the latter usage.

```
manager();
if (...) {
  accountant();
}
if (...)  {
  critical();
  manager();
}
accountant();
critical();
```

Figure 1: Program before monitor inlining.

Program Point	Event
after `manager()`	m
after `accountant()`	a
before `critical()`	c

Figure 2: Mapping program points to access events.

[32]) to modularize analysis. While method summaries are computed by a special-purpose algorithm, method interfaces are generated by a work-list-based whole-program analysis [41, Appendix A].

3. AN OPTIMIZATION-FRIENDLY POLICY REPRESENTATION

3.1 Inlined Reference Monitor

Consider the enforcement of Separation of Duty [11] in an example program shown in Fig. 1 (adapted from [12], in turn inspired by [20, 6]). Our goal is to ensure that the `critical()` operation is performed only under the endorsement of both the `manager()` and `accountant()` operations. To precisely articulate this policy, we interpret the run-time traversal of certain program points to be security-relevant events (Fig. 2): events m, a and c correspond respectively to the three operations. Program execution therefore generates an event sequence. For example, if both of the "then" branches are executed, then the event sequence *macmac* will be generated. Our policy can then be phrased as a safety property regarding the generated event sequences [29]. One way to enforce such a policy is through Inlined Reference Monitors (IRMs) [36]. Specifically, monitoring code is injected into the program points of interest, tracking the history of execution, and aborting execution whenever a policy violation is detected. In Fig. 3, monitoring code has been injected into the original programs identified in Fig. 1, tracking the occurrences of events m and a, and ensuring that every occurrence of event c is properly guarded by both m and a.

Since [29, 35, 36], history-based access control policies are represented by Security Automata. A ***Security Automaton (SA)*** is a quadruple $M = \langle \Sigma, Q, q_0, \{\delta_a\}_{a\in\Sigma}\rangle$, where (i) Σ is a countable set of ***access events***, (ii) Q is a countable set of ***monitor states***, (iii) $q_0 \in Q$ is a distinguished ***start state***, and (iv) $\{\delta_a\}_{a\in\Sigma}$ is a family of ***transition functions***, indexed by access events, such that each transition function $\delta_a : Q \rightharpoonup Q$ is a partial function mapping the current monitor state to an optional next state. Given an event sequence $w \in \Sigma^*$, we write δ_w for the partial function defined inductively as follows: $\delta(\epsilon) = \iota_Q$, the total identity function for Q, and $\delta_{a\cdot w} = \delta_w \circ \delta_a$ (i.e., function composition). Note

```
bool p_m = false;
bool p_a = false;
manager();
p_m = true;
if (...) {
  accountant();
  p_a = true;
}
if (...) {
  if (p_m ∧ p_a) { p_m = false; p_a = false; }
  else throw new IRMException();
  critical();
  manager();
  p_m = true;
}
accountant();
p_a = true;
if (p_m ∧ p_a) { p_m = false; p_a = false; }
else throw new IRMException();
critical();
```

Figure 3: Program after monitor inlining.

that, since δ_w is partial, $\delta_w(q)$ may not be defined for every state q. An event sequence $w \in \Sigma^*$ is considered policy compliant iff $\delta_w(q_0)$ is defined.

At the program points corresponding to event a, IRM injects a code fragment that simulates δ_a. A competitive IRM implementation will subject this code fragment to aggressive optimization.

3.2 A Constrained Policy Representation

Any practical policy representation must place constraints on the Q and δ components [35, 36, 3]. We consider representation constraints that facilitate IRM optimization. Our proposed policy representation is based on two design choices that balance efficiency considerations against policy expressiveness.

Design choice 1: Unbounded state space, finitary transitions.

Unlike [12], which assumes Q to be finite, we anticipate the state space to be unbounded for practical IRM. Specifically, we envision the employment of IRM rewriting at load time, such that the state space may have to be expanded when new code units are dynamically loaded. It is therefore assumed that each application domain is associated with a *countable* set Π of propositional variables[2], called ***state variables***. A Π-state, or simply a state, is an assignment of truth values to propositional variables from Π, such that the assignment differs from one of the following three truth assignments for only finitely many propositional variables: (i) all propositions are undefined, (ii) all propositions are false, and (iii) all propositions are true. Such a truth assignment can be represented using only a finite amount of memory. Henceforth, we identify a state by the set of literals that are satisfied by the state. If neither of the literals for a proposition appears in the set, then the proposition is undefined in the state. Thus the empty set denotes the state in which all

[2]Although we focus on boolean state variables here, our scheme can be readily generalized to handle variables of finite domains.

propositions are undefined. Unless specified otherwise, it is assumed[3] that $q_0 = \emptyset$.

To render execution monitoring tractable, every transition function must be ***finitary***, meaning that (1) only a finite number of state variables determine if the transition is defined at a given state, and (2) the resulting state can be obtained by altering only a finite number of state variables, so that the new value of each variable is a function of only a finite number of state variables in the original state. A finitary transition function is called an ***operator***.

Design choice 2: Conjunctive preconditions, constant effects (CPCE).

An operator can be represented by two elements: (1) a ***precondition expression*** (a boolean expression in terms of a finite number of state variables) indicating if the transition is defined at a given state, and (2) for each state variable that can potentially be altered by the transition function, an ***effect expression*** (a boolean expression in terms of a finite number of state variables) that computes the new value for the variable. While this arrangement is fully general, we impose further syntactic restrictions to arrive at a representation that is optimization-friendly: (1) the precondition expression must be a *conjunction of literals*; (2) every effect expression must be a *constant truth value*. Operators satisfying these restrictions are called ***CPCE operators***. Formally, we represent a CPCE operator by a pair $\langle pre, \mathit{eff}\rangle$, where:

pre: a finite set of ***preconditions***, each of which is a ***literal*** (i.e., p or $\neg p$), such that, for each proposition p, at most one of p or $\neg p$ belongs to the set

eff: a finite set of ***effects***, each of which is a ***generalized literal*** (i.e., p, $\neg p$, or $?p$), such that, for each proposition p, at most one of p, $\neg p$ or $?p$ appears in the set

The state obtained by applying the CPCE operator $\langle pre, \mathit{eff}\rangle$ to a state S (i.e., a set of literals) is:

$$\langle pre, \mathit{eff}\rangle(S) \stackrel{\text{def}}{=} \begin{cases} S \oplus \mathit{eff} & \text{if } pre \subseteq S \\ \text{undefined} & \text{otherwise} \end{cases}$$

where, given a set P of propositions, a set S of literals and a set L of generalized literals,

$$S \oplus L \stackrel{\text{def}}{=} (S \setminus \mathit{lits}(\mathit{vars}(L))) \cup (L \cap \mathit{lits}(\mathit{vars}(L)))$$

$$\mathit{vars}(L) \stackrel{\text{def}}{=} \{p \in \Pi \mid p \in L \ \vee \ \neg p \in L \ \vee \ ?p \in L\}$$

$$\mathit{lits}(P) \stackrel{\text{def}}{=} \{p, \neg p \mid p \in P\}$$

Intuitively, the operator is defined at state S if the conjunction *pre* is satisfied by the truth assignment S. In the resulting state, a propositional variable p is set to true if $p \in \mathit{eff}$, false if $\neg p \in \mathit{eff}$, undefined if $?p \in \mathit{eff}$, or otherwise its original value. As a special case, the ***empty operator*** $\langle \emptyset, \emptyset\rangle$ represents the total identity function ι_Q for monitor states. Also notice that the preconditions of an operator cannot be used for detecting if a proposition is undefined in a given state, but effects could set propositions to undefined. This intentional asymmetry serves an important function to be discussed in the sequel (in the proof of Thm. 4).

[3]The proposed optimization scheme can be easily adopted to the case when this assumption does not hold.

3.3 Evaluation of Expressiveness

We evaluate the expressiveness of the proposed policy representation by a number of case studies.

Simple Integrity Policies.

Complete Mediation [32, 28] requires every sensitive operation to be performed only after a monitoring operation has been invoked. The policy prescribes an event set $\Sigma = \{sen, mon\}$. To enforce the policy, a monitor is constructed with state variable set $\Pi = \{p_m\}$, and transition functions $\delta_{sen} = \langle\{p_m\}, \{\neg p_m\}\rangle$ and $\delta_{mon} = \langle\emptyset, \{p_m\}\rangle$. The transition function δ_{mon} asserts p_m, thus enabling *sen*, which in turn negates p_m.

Separation of Duty (Sect. 3.1) prescribes an access event set $\Sigma = \{a, m, c\}$. The policy is enforced by a monitor for which $\Pi = \{p_a, p_m\}$, where p_a and p_m indicate, respectively, that events a and m have occurred. The transition functions are defined as follows: $\delta_a = \langle\emptyset, \{p_a\}\rangle$, $\delta_m = \langle\emptyset, \{p_m\}\rangle$, $\delta_c = \langle\{p_a, p_m\}, \{\neg p_a, \neg p_m\}\rangle$. The monitor ensures that c only occurs after both a and m have occurred, without imposing an ordering of a and m.

Generalized Chinese Wall Policy.

The Chinese Wall Policy [10] prevents conflicts of interest that may arise from allowing access to data sets that belong to competing parties. Lin proposed a generalization, in which conflict relationships need not form an equivalence relation [23]. In extensible systems, Lin's Generalized Chinese Wall Policy can be employed to ensure that conflicting operations are not executed by an untrusted component, thereby protecting the integrity of the core. Formally, a Generalized Chinese Wall Policy is characterized by a conflict graph $\langle\Sigma, E\rangle$, where Σ is a countable set of operations, and each undirected edge in E connects a pair of operations in conflict with one another. Execution of an operation $a \in \Sigma$ renders all neighbors of a forbidden in the future. Under the mild assumption that vertices of the conflict graph has bounded degrees, the Generalized Chinese Wall Policy can be expressed as CPCE operators as follows. Define $\Pi = \{p_a \mid a \in \Sigma\}$, $q_0 = \{\neg p_a \mid a \in \Sigma\}$, and $\delta_a = \langle\{\neg p_b \mid ab \in E\}, \{p_a\}\rangle$. The construction ensures that the set of executed operations is always an independent set in the conflict graph.

Hierarchical One-Out-Of-k Authorization.

One-out-of-k authorization [14] classifies applications into equivalence classes based on the access rights required for successful execution. For example, a *browser* needs the right to open network connections but never accesses user files, and an *editor* needs the right to access user files but never connects to the network. The protection goal is to ensure that untrusted code only exercises the access rights of a known application class: e.g., an application that both reads a user file and connects to the network is neither a browser nor editor, and thus must be rejected. Formally, an One-Out-Of-k Policy is characterized by a family $\{\mathcal{C}_i\}_{1\le i\le k}$ of application classes such that $\mathcal{C}_i \subseteq \Sigma$. The policy requires that, every time a program is executed, there is a $\mathcal{C}_i$ such that every access right exercised during that execution belongs to $\mathcal{C}_i$. One-out-of-k authorization, in its full generality, is not necessarily expressible as CPCE operators [41, Thm. 1]. Fortunately, there is an important special case that the CPCE representation can capture.

DEFINITION 1. *An One-Out-Of-k Policy* $\{\mathcal{C}_i\}_{1\leq i\leq k}$ *is said to be* ***hierarchical*** *iff both of the following hold:*

$$\forall i,j\,.\,\mathcal{C}_i\cap\mathcal{C}_j\neq\emptyset\Rightarrow\exists m\,.\,\mathcal{C}_m=\mathcal{C}_i\cap\mathcal{C}_j \quad (1)$$

$$\forall i,j,m\,.\,(\mathcal{C}_i\subseteq\mathcal{C}_m\wedge\mathcal{C}_j\subseteq\mathcal{C}_m)\Rightarrow(\mathcal{C}_i\subseteq\mathcal{C}_j\vee\mathcal{C}_j\subseteq\mathcal{C}_i) \quad (2)$$

Condition (1) asserts that the family of application classes is closed under non-empty intersection. Condition (2) asserts that the subclasses of any given class are totally ordered. The Hasse diagram [13] of classes satisfying conditions (1) and (2) is a forest (hence "hierarchical").

THEOREM 2. *Every hierarchical One-Out-Of-k Policy is enforceable by CPCE operators.*

See Appendix A for a proof. For balanced hierarchies, there is a policy encoding in which the size of each precondition and effect set is $\log k$ [41, Thm. 4]. Most naturally-occurring One-Out-Of-k Policies are either hierarchical, or can be made hierarchical without affecting safety [41, Thm. 6] (e.g., the policy in [17]).

4. THE BASIC OPTIMIZATION PROCEDURE

Given a program represented as control flow graphs (CFGs) [20, 6], an IRM enforcement mechanism proceeds in three phases:

Phase 1: By consulting a security policy, construct an associative array $op[\cdot]$, assigning to every program point n some (possibly empty) operator $op[n]$.

Phase 2: Optimize the operator assignment by updating the entries in $op[\cdot]$, in some semantic-preserving manner, with the objective that the resulting execution time is improved.

Phase 3: Instrument the target program by injecting, (a) at the program entry point, a code fragment that initializes a globally accessible monitor state, and, (b) at each program point n, a code fragment simulating $op[n]$. The latter code fragment will behave as follows at run time:

- The preconditions in $op[n].pre$ are checked against the current monitor state. If any of the preconditions is not satisfied, the a security exception is raised[4].
- The effects are asserted into the monitor state.

The focus of this work is **Phase 2** — the design of optimization procedures.

Given $op[\cdot]$, a control flow path is ***feasible*** iff all operator preconditions are satisfied along the path. Unlike [12], which assumes all transitions to have the same cost, we adopt the following performance metric: the ***overhead*** of a feasible path is the total number of preconditions checked and effects asserted along the path. More precisely, an operator $\langle pre, \mathit{eff}\rangle$ incurs an overhead of $|pre|+|\mathit{eff}|$ every time it is executed. The fewer preconditions and effects are involved in an operator, the less overhead it incurs on the target program. For example, the empty operator does not impose an overhead of zero. This performance metric is adopted because the number of propositions appearing in a Π-state can in principle be unbounded, and thus no constant-time implementation of transitions is available.

An ***execution trace*** is a control flow path that starts at the entry point of the program. An optimization procedure is ***safe*** iff infeasible execution traces remain infeasible, and ***unobtrusive*** iff feasible execution traces remain feasible[5]. Given a history-based access control policy, an unsafe optimization procedure may cause an execution trace rejected by the policy to materialize at run time, thereby failing to enforce the policy. Ensuring safety is thus central to the security enterprise. A safe optimization procedure is ***effective*** iff, (a) the overhead of a feasible execution trace is never increased by the procedure, and (b) there is at least one program and a feasible execution trace for that program such that the overhead is *strictly* reduced by the procedure.

We focus on two kinds of optimization: precondition and effect elimination. That is, the optimization procedure eliminates redundant members of $op[n].pre$[6] and $op[n].\mathit{eff}$. As the overhead of a feasible path is never increased by an optimization procedure that is based on precondition and effect elimination, such a procedure is always effective so long as it is safe. The remaining challenge is to conduct precondition and effect elimination without sacrificing safety or incurring obtrusiveness.

4.1 Simple Programs

We describe how precondition and effect elimination can be performed for a single CFG. Let n_{entry}, n_{exit} and N_{instr} be, respectively, the entry node, the exit node and the set of the rest of the nodes in the CFG. Henceforth, we assume that $op[n]=\langle\emptyset,\emptyset\rangle$ initially for $n\notin N_{instr}$. Optimization proceeds in four steps.

Step 1 - Compute a conservative approximation of the guaranteed set for each program point.

A literal l belongs to the ***guaranteed set*** of a program point n iff l is established by every feasible path from n_{entry} to n. This forward analysis is a form of constant propagation [25]:

$$\mathsf{GUA}_{out}[n]=(\mathsf{GUA}_{in}[n]\oplus op[n].pre)\oplus op[n].\mathit{eff} \quad \text{for } n\in N_{instr} \quad (3)$$

$$\mathsf{GUA}_{out}[n]=\emptyset \quad \text{for } n\in\{n_{entry}\} \quad (4)$$

$$\mathsf{GUA}_{in}[n]=\cap_{m\in pred[n]}\mathsf{GUA}_{out}[m] \quad \text{for } n\in N_{instr}\cup\{n_{exit}\} \quad (5)$$

Note the form of (3). By checking the preconditions, an operator has essentially ruled out the infeasible paths. Those that remain must have the preconditions established as a result. Consequently, preconditions could be seen as ***implicit assertions***, while effects are ***explicit assertions***. Notice also that the order of assertion is significant: explicit assertions override implicit assertions.

[4]It is assumed that the target program cannot catch such an exception.

[5]In other words, an execution monitor that is produced by a safe optimization procedure will never generate a false negative, and an unobtrusive optimization procedure produces execution monitors that never generate a false positive.

[6]Given a record r with schema $\langle f_1,\ldots,f_k\rangle$, we refer to the f_i component of r by the notation $r.f_i$. Thus, if $op[n]=\langle pre,\mathit{eff}\rangle$, then $op[n].pre$ refers to pre.

Step 2 - Eliminate redundant preconditions.

A precondition l is considered redundant at program point n if l is guaranteed to be established at n. Precondition elimination is achieved by the following update:

$$op[n].pre := op[n].pre \setminus \mathsf{GUA}_{in}[n] \quad \text{for } n \in N_{instr} \tag{6}$$

Step 3 - Compute a conservative approximation of the live set at each program point.

A proposition p is ***live*** at program point n iff there is a path from n to another program point n' such that (1) p is checked at n', and (2) there is no (implicit or explicit) effect assertion involving p along any path from n to n' [25]. This backward analysis is defined as follows:

$$\mathsf{LIV}_{in}[n] = (\mathsf{LIV}_{out}[n] \setminus kill_{\mathsf{LIV}}[n]) \cup gen_{\mathsf{LIV}}[n] \quad \text{for } n \in N_{instr} \tag{7}$$

$$\mathsf{LIV}_{in}[n] = \emptyset \quad \text{for } n \in \{n_{exit}\} \tag{8}$$

$$\mathsf{LIV}_{out}[n] = \cup_{m \in succ[n]} \mathsf{LIV}_{in}[m] \quad \text{for } n \in N_{instr} \cup \{n_{entry}\} \tag{9}$$

where, for $n \in N_{instr}$,

$$kill_{\mathsf{LIV}}[n] \stackrel{\text{def}}{=} vars(op[n].\mathit{eff})$$

$$gen_{\mathsf{LIV}}[n] \stackrel{\text{def}}{=} vars(op[n].pre)$$

Step 4 - Eliminate redundant effects.

A proposition is ***dead*** at program point n iff it is not live at n. An effect is considered redundant if the effect proposition is dead at the program point where the effect is asserted. Effect elimination is achieved by the following update:

$$op[n].\mathit{eff} := op[n].\mathit{eff} \cap \mathit{gen\text{-}lits}(\mathsf{LIV}_{out}[n]) \quad \text{for } n \in N_{instr} \tag{10}$$

where, given a set P of propositions,

$$\mathit{gen\text{-}lits}(P) \stackrel{\text{def}}{=} \{p, \neg p, ?p \mid p \in P\}$$

THEOREM 3. *The four-step optimization procedure is safe, unobtrusive and effective.*

PROOF. Since only guaranteed preconditions and dead effects are eliminated, the feasibility of a path is not altered by the optimization procedure. Safety and unobtrusiveness thus follow. Effectiveness follows from the fact that the procedure performs only precondition and effect elimination. □

Discussion.

By adopting conjunctive preconditions and constant effects, rather than unconstrained precondition and effect expressions, we have obtained an elegant and informed optimization procedure. First, a function of the form $f_L(S) = S \oplus L$ for a fixed set L of generalized literals is a monotone function [25]. Our representation is thus readily amenable to guaranteed set analysis. Second, the syntactic restriction allows the analyses to deduce more information about guaranteed sets (see (3)) and live sets (see (7)) than an unconstrained representation.

4.2 Procedure Calls

To accommodate programs made up of multiple procedures, we extend our program representation, so that a program is a collection of CFGs. Besides the usual node types n_{entry}, n_{exit} and N_{instr}, every CFG also comes with (a) a set N_{call} of call nodes, (b) a set N_{ret} of return nodes, (c) a bijection $E_{inv} : N_{call} \to N_{ret}$, and (d) a function *callee* mapping call nodes to CFGs. We envision a modular optimization scheme, in which the four-step optimization procedure is applied to CFGs one at a time, and the order in which CFGs are processed is not material. To this end, we adjust data flow equations (4), (5), (8) and (9) as follows:

$$\mathsf{GUA}_{out}[n] = \emptyset \quad \text{for } n \in N_{ret} \cup \{n_{entry}\} \tag{11}$$

$$\mathsf{GUA}_{in}[n] = \cap_{m \in pred[n]} \mathsf{GUA}_{out}[m] \quad \text{for } n \in N_{instr} \cup N_{call} \cup \{n_{exit}\} \tag{12}$$

$$\mathsf{LIV}_{in}[n] = \Pi \quad \text{for } n \in N_{call} \cup \{n_{exit}\} \tag{13}$$

$$\mathsf{LIV}_{out}[n] = \cup_{m \in succ[n]} \mathsf{LIV}_{in}[m] \quad \text{for } n \in N_{instr} \cup N_{ret} \cup \{n_{entry}\} \tag{14}$$

While (12) and (14) are cosmetic changes, (11) and (13) pose significant challenges:

Challenge #1 On entry to a procedure, no knowledge about the caller's state at the call node is available. We are forced to assume the guaranteed set at the procedure entry node is empty (i.e., (11)), thereby reducing the opportunities for precondition elimination within the procedure body.

Challenge #2 On exit from a procedure, no knowledge about the caller's live set at the return node is available. We are forced to assume that all propositions are live (i.e., (13)), thereby reducing the opportunities for effect elimination within the procedure body.

Challenge #3 By (13), effects asserted prior to a call node cannot be readily eliminated.

Challenge #4 By (11), precondition checks following a return node cannot be readily eliminated.

In the next section, we discuss a distributed optimization protocol that would allow an untrusted code producer to assist a distrusting code consumer in addressing the above challenges.

5. A DISTRIBUTED OPTIMIZATION PROTOCOL

5.1 Cooperative Optimization without Assuming Trust

Consider a program distribution scenario inspired by [24], in which an untrusted code producer $\mathcal{P}$ distributes a program $\mathbb{P}$ to a code consumer $\mathcal{C}$ for execution. Suppose $\mathcal{C}$ employs IRM to enforce a history-based access control policy, while $\mathcal{P}$, eager to promote the usage of $\mathbb{P}$, offers to help boost the optimization effectiveness of $\mathcal{C}$. How can $\mathcal{C}$ securely accept the contribution of $\mathcal{P}$? We propose the following ***distributed optimization protocol***.

Stage 1: $\mathcal{C}$ publishes, over an *untrusted* media, a security policy $\pi = \langle \Pi, \{\delta_a\}, \alpha \rangle$, where Π is a set of state variables, $\{\delta_a\}$ a family of operators for Π-states, and α a procedure that computes, for a program $\mathbb{P}$, an associative array $op[\cdot]$ mapping every program point in $\mathbb{P}$ to an operator from $\{\delta_a\}$.

Stage 2: $\mathcal{P}$ submits π and an *untrusted* program $\mathbb{P}$ to an *untrusted* ***oracle***, which generates a set D of ***optimization directives***. D contains annotations designed to inform $\mathcal{C}$ of how aggressive optimization can be achieved.

Stage 3: $\mathcal{P}$ ships the package $\langle \mathbb{P}, D \rangle$ to $\mathcal{C}$ via an *untrusted* channel.

Stage 4: $\mathcal{C}$ performs the steps below before executing $\mathbb{P}$:

Phase 1: Use procedure α to construct operator assignment $op[\cdot]$ for $\mathbb{P}$.

Phase 2: Update $op[\cdot]$ as follows: **(a)** D is exploited to optimize $op[\cdot]$ aggressively. **(b)** As D cannot be fully trusted, blindly following the optimization directives may destroy the safety of the optimization procedure. Additional "guards" are injected into $op[\cdot]$, so that fraudulent annotations are detected when $\mathbb{P}$ is executed.

Phase 3: Inject $op[\cdot]$ into $\mathbb{P}$.

The protocol is particularly appropriate for a $\mathcal{C}$ that is computationally constrained (e.g., IRM via load-time binary rewriting), and a $\mathcal{P}$ having access to a computationally powerful oracle (e.g., offline certification service). In the sequel, we specialize the protocol for addressing the four optimization challenges outlined in Sect. 4.2.

5.2 Procedure Interfaces

We postulate that the code producer attaches a ***procedure interface*** to every procedure it ships. Each procedure interface is a quadruple $\langle pre, post, dead_{in}, dead_{out} \rangle$, where:

pre: a set of literals guaranteed by the caller to be established at the call node.

post: a set of literals guaranteed by the procedure to be established at the exit node.

$dead_{in}$: a set of propositions guaranteed by the procedure to be dead at the entry node.

$dead_{out}$: a set of propositions guaranteed by the caller to be dead at the return node.

The main procedure must have an interface of $\langle \emptyset, \emptyset, \Pi, \Pi \rangle$. Interfaces of other procedures can be generated by the code producer using an appropriate whole-program analysis (see [41, Appendix A] for a complete algorithm).

5.3 Using Procedure Interfaces as Optimization Directives

The code consumer treats the procedure interfaces as optimization directives. Specifically, $\mathcal{C}$ uses the interfaces to perform more accurate analyses in **Step 1** and **Step 3** of the four-step optimization procedure. To see this, assume there is a symbol table function *symtbl* that maps every call node to the procedure interface of the callee.

Step 1 - Guaranteed set analysis.

We replace data flow equation (11) by the following:

$$\mathsf{GUA}_{out}[n] = pre \qquad \text{for } n \in \{n_{entry}\} \quad (15)$$

$$\mathsf{GUA}_{out}[n] = symtbl(E_{inv}^{-1}(n)).post \qquad \text{for } n \in N_{ret} \quad (16)$$

(The expression $symtbl(E_{inv}^{-1}(n))$ refers to the callee's procedure interface for $n \in N_{ret}$.) Rather than indiscriminately taking guaranteed sets to be $\emptyset$ at the entry node and the return nodes, the interface components *pre* and *post* now inform guaranteed set analysis, thereby creating more opportunities for precondition elimination, and thus addressing **Challenges 1 & 4**. This works so long as *pre* and *post* are trustworthy annotations.

Step 3 - Liveness analysis.

We replace data flow equation (13) by the following:

$$\mathsf{LIV}_{in}[n] = (\Pi \setminus dead_{out}) \cup vars(op[n].pre) \quad \text{for } n \in \{n_{exit}\} \quad (17)$$

$$\mathsf{LIV}_{in}[n] = (\Pi \setminus symtbl(n).dead_{in}) \cup vars(op[n].pre) \quad \text{for } n \in N_{call} \quad (18)$$

(The subexpression $vars(op[n].pre)$ does not concern us for now, because, by setting $op[n]$ initially to $\langle \emptyset, \emptyset \rangle$ for $n \notin N_{instr}$, the subexpression is essentially $\emptyset$. It becomes indispensable when $op[n]$ is not empty, as is the case once (19), (20) and (21) have been introduced.) If the interface components $dead_{in}$ and $dead_{out}$ are trustworthy, then they inform liveness analysis at the exit node and the call nodes, thereby addressing **Challenges 2 & 3**.

5.4 Guarding Against Fraudulent Procedure Interfaces

But the procedure interfaces are not to be trusted! They could cause essential monitoring logic to be optimized away. To prevent this, **Steps 2** and **4** of the four-step optimization procedure are adapted as follows.

Step 2 - Precondition elimination.

This step now involves two subtasks. First, associate an ***auxiliary operator*** to the exit node and each call node:

$$op[n] := op_{guard}(symtbl(n).pre) \qquad \text{for } n \in N_{call} \quad (19)$$

$$op[n] := op_{guard}(post) \qquad \text{for } n \in \{n_{exit}\} \quad (20)$$

where, given a set S of literals, $op_{guard}(S)$ is the effect-less operator $\langle S, \emptyset \rangle$. The injected operators guarantee that the assumptions made in data flow equations (15) and (16) are verified at run time.

The second subtask is precondition elimination, which is performed also on the newly introduced operators:

$$op[n].pre := op[n].pre \setminus \mathsf{GUA}_{in}[n] \quad \text{for } n \in N_{instr} \cup N_{call} \cup \{n_{exit}\} \quad (21)$$

Step 4 - Effect elimination.

Again, this step is now divided into two subtasks. First, an auxiliary operator is assigned to every entry and return

node.

$$op[n] := op_{assert}(\mathsf{GUA}_{out}[n], dead_{in}) \quad \text{for } n \in \{n_{entry}\} \tag{22}$$

$$op[n] := op_{assert}(\mathsf{GUA}_{out}[n], symtbl(E_{inv}^{-1}(n)).dead_{out}) \quad \text{for } n \in N_{ret} \tag{23}$$

where, given a set S of literals and a set P of propositions, $op_{assert}(S, P)$ is the precondition-less operator $\langle \emptyset, (S \cap lits(P)) \cup \{?p \mid p \in P \backslash vars(S)\}\rangle$. The operator assigns a value to each of the propositions in P. For each proposition in P that also appear in a literal in S, the assigned value is specified by the literal. For each proposition in P that does not appear in a literal in S, the assigned value is undefined. Essentially, the operator forces all propositions in P to become dead at run time, and serves as a "guard" for the assumptions made in (17) and (18).

The second subtask is effect elimination, which is also performed on the newly introduced auxiliary operators.

$$op[n].\mathit{eff} := op[n].\mathit{eff} \cap \mathit{gen\text{-}lits}(\mathsf{LIV}_{out}[n]) \quad \text{for } n \in N_{instr} \cup N_{ret} \cup \{n_{entry}\} \tag{24}$$

THEOREM 4. *The revised optimization procedure is safe.*

PROOF. Suppose the value of $op[n]$ has been updated from $\langle \emptyset, \emptyset \rangle$ to $op_{guard}(S)$ for some set S of literals. As the operator is effect-less, every infeasible path containing n remains infeasible. The introduction of $op_{guard}(S)$ in updates (19) and (20) thus preserves safety.

Now, suppose the value of $op[n]$ has been updated from $\langle \emptyset, \emptyset \rangle$ to $op_{assert}(\mathsf{GUA}_{out}[n], P)$, for some set P of propositions. Consider an effect asserted by the auxiliary operator. If the effect is of the form $?p$, then it only causes future precondition checks to fail, but never establishes any precondition (recall that a precondition cannot be used to check if a proposition is undefined). If the effect is a literal, and it establishes a precondition, then the precondition is already guaranteed prior to the assertion of the literal. In either case, infeasible paths remain infeasible. Updates (22) and (23) thus preserve safety. □

In other words, if the code producer attempts to mislead the code consumer by sending fraudulent procedure interfaces, the fraud will be detected by the IRM at run time. Thm. 4 therefore guarantees the security of the distributed optimization protocol.

The interface $\langle pre, post, dead_{in}, dead_{out}\rangle$ of a procedure *proc* is said to be ***conservative*** iff all the following hold: (a) $pre \subseteq \mathsf{GUA}_{in}[n]$ for every call node n for which *proc* is the callee, (b) $post \subseteq \mathsf{GUA}_{in}[n]$ for the exit node n of *proc*, (c) $dead_{in} \subseteq \Pi \setminus \mathsf{LIV}_{out}[n]$ for the entry node n of *proc*, and (d) $dead_{out} \subseteq \Pi \setminus \mathsf{LIV}_{out}[n]$ for every return node n for which *proc* is the callee.

THEOREM 5. *With conservative interfaces, the revised optimization procedure is unobtrusive and effective.*

PROOF. If all procedure interfaces are conservative, then the updates (21) and (24) will completely remove the preconditions and effects of the auxiliary operators introduced in (19), (20), (22) and (23). □

In other words, if the code producer is honest about the optimization directives, then the run-time checks for fraud detection will be optimized away. Conservative procedure interfaces can be generated by the interface generation algorithm described in [41, Appendix A].

5.5 Accommodating Java-Style Language Constructs

Exception handling.

In [41, Sect. 6.1] we provide details on how our optimization procedure can be extended to accommodate Java-style exception handling constructs. As a highlight, procedure interfaces must now assume the form $\langle pre, post, esc, dead_{in}, dead_{out}, dead_{fail}\rangle$, where the new components have the following roles:

esc: a set of literals guaranteed by the procedure to be established when an exception escapes the procedure

$dead_{fail}$: a set of propositions guaranteed to be dead by handlers of exceptions escaping from the procedure

Method overriding.

In the presence of dynamic method dispatching, the code consumer must verify that method overriding honors certain constraints among method interfaces. Given method interfaces $\mathcal{I} = \langle pre, post, esc, dead_{in}, dead_{out}, dead_{fail}\rangle$ and $\mathcal{I}' = \langle pre', post', esc', dead'_{in}, dead'_{out}, dead'_{fail}\rangle$, we write $\mathcal{I}' \sqsubseteq \mathcal{I}$ iff all of the following hold:

$$pre \supseteq pre' \qquad post \subseteq post' \qquad esc \subseteq esc'$$
$$dead_{in} \subseteq dead'_{in} \qquad dead_{out} \supseteq dead'_{out} \qquad dead_{fail} \supseteq dead'_{fail}$$

The constraints follow the usual contravariant pattern of function subtyping [26]. To preserve safety, the code consumer must verify that $\mathcal{I}' \sqsubseteq \mathcal{I}$ whenever a method with interface $\mathcal{I}'$ overrides a method with interface $\mathcal{I}$. Since $\sqsubseteq$ is transitive, only direct method overrides need to be verified. The interface generation algorithm in [41, Appendix A] can be used by the code producer to generate method interfaces guaranteed to satisfy the above. Details on the treatment of method overriding can be found in [41, Sect. 6.2].

6. IMPLEMENTATION EXPERIENCE

We developed a Java prototype for the revised optimization procedure (Sect. 5), with Java bytecode as the target language. Our prototype handles the entire Java bytecode language. The prototype was developed in Soot [37], a framework for Java bytecode manipulation and optimization. Soot provides facilities for converting Java bytecode into more manageable internal representations, performing control flow analysis to construct control flow graphs, as well as providing infrastructure code for iterative, intraprocedural data flow analyses. Specifically, our prototype consists of three components: (1) a modular optimization procedure, which applies the revised four-step optimization procedure to a CFG and an operator assignment, (2) an instrumentation module that converts a CFG and an operator assignment to Java bytecode, and (3) a method interface generator, which is a whole-program analysis built on top of the modular optimization procedure [41, Appendix A].

Soot's built-in control flow analyzer has been adopted to construct control flow graphs in the presence of exceptions.

Name/Version	Description	#classes	#methods
BCEL/5.2	framework for manipulating Java bytecode	384	3184
BcVer/1.0	prints classfile version	11	120
JavaCC/4.0	parser generator	137	2091
JavaTar/2.5	tar-style archiving tool	15	176
ProGuard/4.2	classfile shrinker, optimizer, obfuscater & pre-verifier	447	4211
SableCC/3.2	parser generator	285	2366

Figure 4: Benchmarking suite.

Although Soot provides "hooks" for programmers to customize the control flow analyzer so that more accurate exception flows can be obtained, we refrain from following that trail, as precise exception escape analysis is outside of the scope of this work. We however modified the code base of the Soot data flow analysis framework to accommodate the complex data flow equations caused by exception handling (see [41, Sect. 6.1] for details).

7. EMPIRICAL EVALUATION

We employed our prototype to empirically assess the degree to which an IRM enforcement mechanism can benefit from the four-step optimization procedure (Sect. 4), as well as the further improvements brought about by adopting method interfaces as optimization directives in a distributed optimization protocol (Sect. 5). To benchmark our optimization schemes against production-quality control flow graphs, we selected a suite of open source Java applications for our experiments (see Fig. 4). We intentionally consider only batch-processing applications, so that we can fully automate the benchmarking process. For each program, we also select a naturally-occurring input to accompany the program.

To profile the performance of our optimization procedure against history-based access control policies of various structural characteristics, we designed a stochastic procedure for generating benchmarking policies. Given a program $\mathbb{P}$ and an input $\mathbb{I}$, an instance of the ***experimental configuration*** $\mathbf{EC}[p_{node}, p_{\mathit{eff}}, p_{pre}]$ (where p_{node}, p_{eff} and p_{pre} are probabilities) is an operator assignment $op[\cdot]$ stochastically constructed as follows:

1. Select a set N of program points from $\mathbb{P}$ as targets of operator injection. Each program point is selected with probability p_{node}. Operator assignment $op[n]$ will remain $\langle \emptyset, \emptyset \rangle$ for $n \notin N$.

2. Fix a set Π of ten propositions (i.e., boolean variables). For each $n \in N$, set $op[n]$ to $\langle \emptyset, \mathit{eff}_n \rangle$, where each eff_n is constructed independently as follows: Select a subset P of Π, such that each $p \in \Pi$ is selected independently with probability p_{eff}. Then, construct eff_n such that, for each $p \in P$, with equal probability either p or $\neg p$ appears in eff_n.

3. Instrument $\mathbb{P}$ with $op[\cdot]$ and then execute $\mathbb{P}$ on input $\mathbb{I}$. Record the traversed control flow path.

4. For each program point $n \in N$ that appears on the recorded path, compute the set $\mathsf{GUA}_{in}[n]$ of literals guaranteed to be satisfied at n whenever n is visited during the above execution.

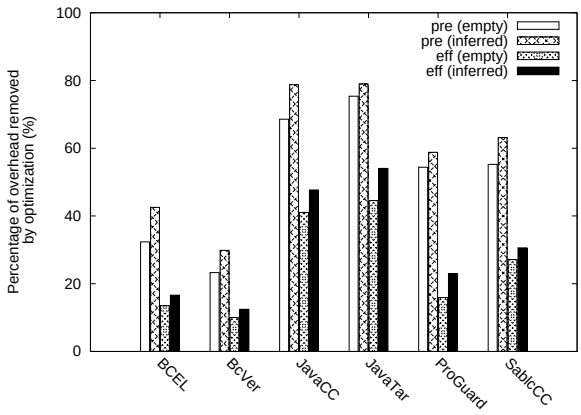

Figure 7: Optimization effectiveness with and without method interfaces.

5. For each $n \in N$, select a subset pre_n of literals from $\mathsf{GUA}_{in}[n]$, such that each member of $\mathsf{GUA}_{in}[n]$ is selected with probability p_{pre}.

6. Set $op[n]$ to $\langle \mathit{pre}_n, \mathit{eff}_n \rangle$ for each $n \in N$. This is the operator assignment we seek to construct.

The construction procedure guarantees that, on input $\mathbb{I}$, program $\mathbb{P}$ honors the policy represented by $op[\cdot]$, and thus benchmarking will not be interrupted by security exceptions. The probability p_{node} measures ***operator density***, while the probabilities p_{eff} and p_{pre} measure ***effect density*** and ***precondition density*** respectively.

Given a program $\mathbb{P}$, an input $\mathbb{I}$, and an operator assignment $op[\cdot]$, the effectiveness of an optimization procedure is measured as follows. First, $\mathbb{P}$ is instrumented with $op[\cdot]$, and the instrumented program is executed with input $\mathbb{I}$. The overhead of execution (as defined in Sect. 4) is recorded. To better assess the relative effectiveness of precondition and effect elimination, we record the number of preconditions checked as O^{org}_{pre}, and the number of effects checked as O^{org}_{eff}. Second, the process is repeated with an optimized operator assignment obtained by applying Ω to $op[\cdot]$. The overhead of execution as incurred by precondition checks and effect assertions are recorded as O^{opt}_{pre} and O^{opt}_{eff}. Optimization effectiveness is then expressed as the ratios $R_{pre} = 1 - O^{opt}_{pre}/O^{org}_{pre}$ and $R_{\mathit{eff}} = 1 - O^{opt}_{\mathit{eff}}/O^{org}_{\mathit{eff}}$. More effective optimization procedures have larger R_{pre} and R_{eff}.

Our experiments were conducted on an IntelCore 2 Duo 2.33GHz iMac with 2GB of RAM, running Mac OS X 10.4.9, JDK 1.6.0 Update 3, Soot 2.2.5 and Jasmin 2.2.5.

7.1 Experiment 1: Optimization With and Without Optimization Directives

In a first experiment, two instantiations of the revised optimization procedure (Sect. 5) were considered. In the first instantiation, all method interfaces are set to $\langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \Pi \rangle$. Adopting an (almost) empty method interface reduces the revised optimization procedure to the basic version reported in Sect. 4, except that by setting $\mathit{dead}_{\mathit{fail}}$ to Π we avoid confusing the optimization algorithm with the overly conservative control flow analysis built into Soot for analyzing exception flow. In the second instantiation, we employed the method interface generation algorithm [41, Appendix A] to generate conservative method interfaces for all methods, and then set $\mathit{dead}_{\mathit{fail}}$ uniformly to Π for the same reason. This

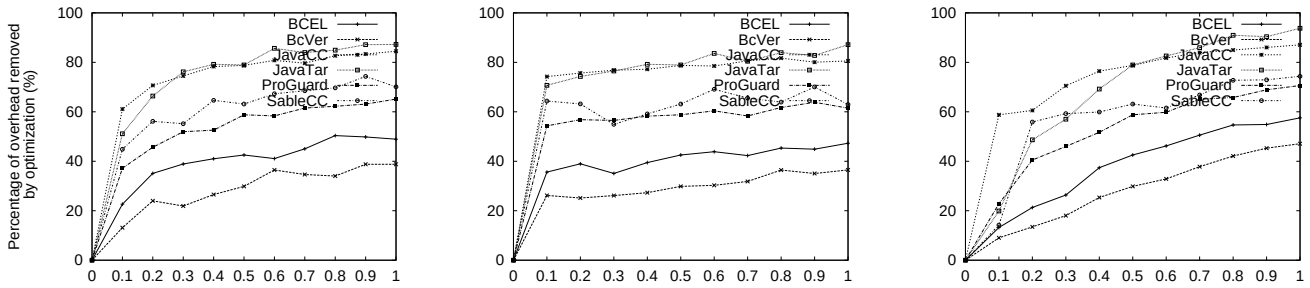

Figure 5: R_{pre} **with different (a)** p_{eff} **(b)** p_{pre} **(c)** p_{node}**.**

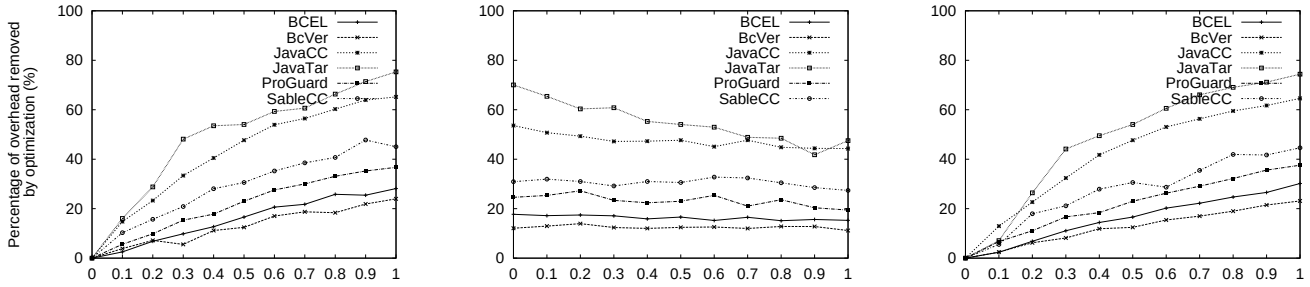

Figure 6: R_{eff} **with different (a)** p_{eff} **(b)** p_{pre} **(c)** p_{node}**.**

second instantiation allows us to measure the effectiveness of the revised optimization procedure (Sect. 5) when the distributed optimization protocol is employed. Note that the second instantiation never underperforms the first because the method interfaces used were conservative.

We generated ten instances of $\mathbf{EC}[0.5, 0.5, 0.5]$ for each program in Fig. 4, and then measured the optimization effectiveness ratios R_{pre} and R_{eff} for each instantiation of the optimization procedure. The measurements for the ten instances were averaged and shown in Fig. 7. The bars labeled **pre (empty)** and **eff (empty)** show the average R_{pre} and R_{eff} for the optimization procedure with empty method interfaces, while **pre (inferred)** and **eff (inferred)** correspond to average R_{pre} and R_{eff} for the optimization procedure with inferred method interfaces.

Three observations can be made from Fig. 7. (1) Both precondition and effect elimination deliver significant reduction in performance overhead, even when method interfaces are not present. (2) Precondition elimination has a much higher effectiveness than effect elimination. (3) The added effectiveness of method interfaces is noticeable but not dramatic.

7.2 Experiment 2: Varying Policy Characteristics

To characterize optimization effectiveness under various policy structures, we subject the revised optimization procedure (with inferred method interfaces) to different experimental configurations. Specifically, we varied each of p_{node}, p_{eff} and p_{pre} from 0 to 1, by increments of 0.1, while keeping the other two parameters fixed at 0.5. Again, ten instances of each experimental configuration were generated, and the average effectiveness ratios R_{pre} and R_{eff} for each configuration are depicted respectively in Fig. 5 and 6.

From Fig. 6 (a) and (b), we notice that R_{eff} increases with an increasing effect density (p_{eff}), but decreases with an increasing precondition density (p_{pre}). We argue that this can be readily explained by data flow equation (7). A higher p_{eff} increases the size of $\mathit{kill}_{\mathsf{LIV}}[\cdot]$, creating larger dead sets, and thus promotes effect elimination. A higher p_{pre}, however, increases the size of $\mathit{gen}_{\mathsf{LIV}}[\cdot]$, creating smaller dead sets, and thus discourages effect elimination. Similarly, from Fig. 5 (a) and (b), we notice that R_{pre} increases with either an increasing effect density (p_{eff}) or an increasing precondition density (p_{pre}). This can be explained readily by data flow equation (3), in which larger effect and precondition sets produce larger guaranteed sets, which in turn promote precondition elimination. Notice also that implicit assertion is overridden by explicit assertion, thus explaining why Fig. 5 (b) shows a less dramatic increase than Fig. 5 (a). The above observations imply that:

> *If two different encodings of the same security policy incur similar overhead, then we should prefer the encoding with more effects and less preconditions, for such a policy is more amenable to optimization.*

Fig. 5 (c) and 6 (c) show that higher operator density (p_{node}) produces higher optimization effectiveness.

> *IRM benefits more from precondition and effect elimination when more program points are interpreted as access events.*

8. CONCLUDING REMARKS

We proposed a constrained policy representation for facilitating IRM optimization. Our policy representation is expressive enough to represent simple integrity policies, Generalized Chinese Wall Policies, and Hierarchical One-Out-Of-k Policies. Our core optimization procedure is safe, unobtrusive and effective. The optimization procedure has been extended to accommodate a distributed optimization protocol, in which an untrusted code producer may formulate method interfaces to boost the optimization effectiveness of a distrusting code consumer. A prototype of the procedure has been implemented, and demonstrated to exhibit positive performance characteristics.

We are exploring alternative optimization directives that could lead to more effective optimization than our current design of method interfaces. While our current policy representation and distributed optimization protocol are designed for supporting control flow-based policies, we are also exploring how they can be extended to enforce data flow constraints [7].

9. ACKNOWLEDGMENTS

This work is supported in part by a NSERC Discovery Grant and a NSERC Strategic Network Grant.

10. REFERENCES

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS'03)*, San Diego, California, USA, Feb. 2003.

[2] I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. In *Proceedings of the 15th International Symposium on Formal Methods (FM'08)*, Turku, Finland, May 2008.

[3] I. Aktug and K. Naliuka. ConSpec – a formal language for policy specification. In *Proceedings of the First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07)*, volume 197 of *Electronic Notes in Theoretical Computer Science*, 2007.

[4] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Proceedings of the 22nd ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'07)*, Montréal, Québec, Canada, Oct. 2007.

[5] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Proceedings of the Workshop on Foundations of Computer Security (FCS'02)*, Copenhagen, Denmark, July 2002.

[6] F. Besson, T. Jensen, D. L. Métayer, and T. Thorn. Model checking security properties of control flow graphs. *Journal of Computer Security*, 9(3):217–250, 2001.

[7] S. Bhatkar, A. Chaturvedi, and R. Sekar. Dataflow anomaly detection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, Berkeley, CA, USA, May 2006.

[8] K. Biba. Integrity considerations for secure computer systems. Technical Report 76–372, U. S. Air Force Electronic Systems Division, 1977.

[9] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, pages 18–27, Oct. 1985.

[10] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium on Research in Security and Privacy (S&P'89)*, pages 206–214, Oakland, California, USA, May 1989.

[11] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy (S&P'87)*, pages 184–194, May 1987.

[12] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 54–66, Boston, MA, USA, Jan. 2000.

[13] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2nd edition, 2002.

[14] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS'98)*, San Francisco, California, USA, 1998.

[15] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (S&P'99)*, pages 32–45, Oakland, California, USA, May 1999.

[16] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[17] P. W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy (S&P'04)*, pages 43–55, Berkeley, California, USA, May 2004.

[18] L. Gong and R. Schemers. Implementing protection domains in the Java Development Kit 1.2. In *Proceedings of the 1998 ISOC Symposium on Network and Distributed System Security (NDSS'98)*, San Diego, Carlifornia, USA, Mar. 1998.

[19] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, Jan. 2006.

[20] T. Jensen, D. L. Métayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (S&P'99)*, pages 89–103, Oakland, California, USA, May 1999.

[21] C. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *LNCS*, Finland, June 1997.

[22] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005.

[23] T. Y. Lin. Chinese Wall security policy: An aggressive model. In *Proceedings of the Fifth Annual Computer Security Applications Conference (ACSAC'89)*, pages 282–289, Tucson, Arizona, USA, Dec. 1989.

[24] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 106–119, Paris, France, Jan. 1997.

[25] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2004.

[26] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[27] E. Rose and K. H. Rose. Lightweight bytecode verification. In *The OOPSLA'98 Workshop on Formal Underpinnings of Java*, Vancouver, BC, Canada, Nov. 1998.

[28] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, 1975.

[29] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[30] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 86–101, 2000.

[31] R. Sekar, V. N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, Oct. 2003.

[32] A. P. Sistla, V. N. Venkatakrishnan, M. Zhou, and H. Branske. CMV: Automatic verification of complete mediation for Java Virtual Machine. In *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security (ASIACCS'08)*, pages 100–111, Tokyo, Japan, Mar. 2008.

[33] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement for limited-memory systems. In *Proceedings of the 2006 Conference on Privacy, Security and Trust (PST'06)*, Markham, Ontario, Canada, Oct. 2006.

[34] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2–4):158–184, Feb. 2008.

[35] Úlfar Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 New Security Paradigm Workshop (NSPW'99)*, pages 87–95, Caledon Hills, Ontario, Canada, Sept. 1999.

[36] Úlfar Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy (S&P'00)*, pages 246–255, Berkeley, California, USA, May 2000.

[37] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the 9th International Conference on Compiler Construction (CC'00)*, pages 18–34, 2000.

[38] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, Oct. 2000.

[39] J. Wang, Y. Takata, and H. Seki. HBAC: A model for history-based access control and its model checking. In *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS'06)*, volume 4189 of *LNCS*, pages 263–278, Hamburg, Germany, Sept. 2006. Springer.

[40] I. Welch and R. J. Stroud. Using reflection as a mechanism for enforcing security policies on compiled code. *Journal of Computer Security*, 10(4):399–432, 2002.

[41] F. Yan and P. W. L. Fong. Efficient IRM enforcement of history-based access control policy. Technical Report CS-2008-03, Department of Computer Science, University of Regina, Regina, Saskatchewan, Canada, Nov. 2008.

APPENDIX

A. PROOF OF THEOREM 2

PROOF. Consider a Hierarchical One-Out-Of-k Policy $\{\mathcal{C}_i\}_{1\leq i\leq k}$. Without loss of generality, assume that every $a \in \Sigma$ belongs to at least one $\mathcal{C}_i$. Define the ***home class*** $\mathcal{H}(a)$ of access $a \in \Sigma$ to be $\bigcap\{\, \mathcal{C} \in \{\mathcal{C}_i\}_{1\leq i\leq k} \mid a \in \mathcal{C} \,\}$, that is, the smallest class containing a. (The existence of such a class is guaranteed by condition (1).) A pair of accesses, say a and b, is said to be ***consistent*** iff they belong to the same application class: i.e., $\exists i \,.\, \{a, b\} \subseteq \mathcal{C}_i$. Otherwise, they are ***in conflict***. Notice that a and b are consistent iff $\mathcal{H}(a) \subseteq \mathcal{H}(b) \vee \mathcal{H}(b) \subseteq \mathcal{H}(a)$. (The "if" direction is immediate. The "only if" direction follows from $\{a, b\} \subseteq \mathcal{C}_i$ by an application of condition (2).)

To obtain the required CPCE representation of $\{\mathcal{C}_i\}_{1\leq i\leq k}$, construct $\Pi = \{p_{\mathcal{C}} \mid \mathcal{C} \in \{\mathcal{C}_i\}_{1\leq i\leq k}\}$, $q_0 = \{\neg p_{\mathcal{C}} \mid \mathcal{C} \in \{\mathcal{C}_i\}_{1\leq i\leq k}\}$, and $\delta_a = \langle pre_a, eff_a\rangle$, where:

$$pre_a = \{\neg p_{\mathcal{C}} \mid \mathcal{H}(a) \not\subseteq \mathcal{C} \wedge \mathcal{C} \not\subseteq \mathcal{H}(a)\}$$
$$eff_a = \{p_{\mathcal{H}(a)}\}$$

It is easy to see that, with the CPCE operators above, at run time, the set H of accesses that have occurred so far are pair-wise consistent. What we want is that there is a $\mathcal{C}_i$ such that $H \subseteq \mathcal{C}_i$. We prove this by induction.

The base cases for $|H| \leq 2$ can be handled trivially. Suppose, for some $k > 2$, all event set H with $|H| = k$ is such that $H \subseteq \mathcal{C}_i$ for some i whenever H contains pairwise-consistent events. Consider a set $H' = H \cup \{a\}$ where $|H| = k$, $a \notin H$, and events in H' are pairwise consistent. By way of contradiction, assume the following holds:

$$\textit{There is no } \mathcal{C}_i \textit{ such that } H' \subseteq \mathcal{C}_i. \tag{25}$$

Because H contains pairwise-consistent events, the induction hypothesis implies that there is a class $\mathcal{C}^\star$ such that $H \subseteq \mathcal{C}^\star$. Also, a is consistent with every member of H. Thus, for each $b \in H$, let $\mathcal{C}_b$ be a class containing both a and b. By (1), $\mathcal{C}^\circ = \bigcap_{b\in H} \mathcal{C}_b$ is a class. By assumption (25), there is an event $b^\star \in H$ such that $b^\star \notin \mathcal{C}^\circ$. By (1), $\mathcal{C}^\bullet = \mathcal{C}^\star \cap \mathcal{C}_{b^\star}$ is a class. Now, $a \in \mathcal{C}^\circ$, but $a \notin \mathcal{C}^\bullet$; $b^\star \in \mathcal{C}^\bullet$, but $b^\star \notin \mathcal{C}^\circ$. So $\mathcal{C}^\circ$ and $\mathcal{C}^\bullet$ are distinct, incomparable subsets of $\mathcal{C}_{b^\star}$, contradicting (2). □