Optimizing IRM with Automata Modulo Theory [⋆]

F. Massacci [a,1] I. Siahaan [a,2]

[a] *Department of Information Engineering and Computer Science (DISI), University of Trento, Italy*

Abstract

Inlined Reference Monitor (IRM) is a flexible mechanism to enforce the security of untrusted applications. One of the shortcomings of IRM is that it might introduce a significant overhead in otherwise perfectly secure application. In this paper we propose six different framework models for IRM optimization with respect to components that are needed to be trusted or untrusted. Then, we describe an approach for IRM optimization using automata modulo theory. The key idea is that given a *policy* that represents the desired security behavior of a platform to be inlined, we compute an *optimized policy* with respect to the (trusted) claims on the security behavior of a application. The optimized policy is the one to be injected into the untrusted code.

Keywords: Access control, Language-based security, Malicious code, Security and privacy policies

1 Motivations

In the realm of mobile communication the strong push for fancy functionalities has brought us a wealth of communicating applications by more or less unknown sources ranging from P2P game clients to local search engines, each of them plowing through the user's platform, and springing back with services from and to the rest of the world.

The security problems arising when application developers and platform owners are not on the same (security) side are well known from the experience of Java web applications for the desktop. The confinement of Java applets [18] is a classical solution. Indeed, to deal with the untrusted code either .NET [27] or Java [18] can exploit the mechanism of permissions. Permissions are assigned to enable execution of potentially dangerous or costly functionality, such as starting various types of connections. The drawback of permissions is that after assigning a permission the user has very limited control over how the permission is used. Conditional permissions that allow and forbid use of the functionality depending on such factors as

⋆ Research partly supported by the Projects EU-FP7-IP-SECURECHANGE and EU-FP7-IP-MASTER.

1 Email:fabio.massacci@unitn.it

2 Email:siahaan@disi.unitn.it

bandwidth or the previous actions of the application itself (e.g. access to sensitive files) are also out of reach. The consequence is that either applications are sandboxed (and thus can do almost nothing), or the user decided that they come from a trusted source and then they can do almost everything. But coming from a trusted source is not a surrogate for trustworthiness[3].

In order to overcome the well-known limitation of the trusted signature or sandbox a number of techniques have been proposed and implemented. The most promising one is the notion of Inlined Reference Monitor (IRM): it works by inlining the code with the security policies. IRM has been implemented in several systems, for example the PoET/PSLang toolkit [13], enforcing security policies whose transitions pattern-match event symbols using regular expressions, or Polymer [7] based on edit automata. The shortcoming of traditional IRM is the huge overhead resulted from inlining.

Other extensions along this line have been further proposed. Sekar et al. [37] have proposed the notion of Model Carrying Code (MCC) but not practical working mechanism has been delivered. Later the Security-by-Contract (SxC) framework [11] built upon the MCC seminal idea have shown that an effective and comprehensive version of IRM can be deployed on mobile platforms [10]. Furthermore, Phung et al. [36] proposed lightweight version of IRM for JavaScript that does not modify browser or original code i.e. it adds new code only in the header of the page. An alternative approach to IRM is by using reflection [38], where policies are implemented as meta-objects bounded to application objects at load time, such that the code becomes self-protecting.

Even if current version of IRM can work on rich system such as today's smart phones, the overhead is still too much for the next frontier of web applications: Java cards. Indeed, the smart card technology [32] evolved with larger memories, USB and TCP/IP support and the development of the Next-Generation (for short NG) Java Card platform [1,2] with Servlet engine. This latter technology is a full fledged Java platform for embedded Web applications and opens new Web 2.0 opportunities such as NG Java Card Web 2.0 Applications. It can also serve as an alternative to personalized applications on remote servers so that personal data no longer needs to be transmitted to remote third-parties.

Thus, optimizing redundant monitoring without compromising security is needed. The key idea is that given a *policy* that represent the desired security behavior of a platform to be inlined, we compute an *optimized policy* with respect to the claims on the security behavior of a application (for short *contract*). Then, we use this *optimized policy* to inject the untrusted code. In the first work [39] proposed IRM optimization for a constrained history-based access control policy such as Chinese Wall policies using compiler optimization approach. Unfortunately, this approach is severely limited by the expressivity of the language: it only consider propositional conditions on policies. As a result even a simple policy such as "Only allows connections to urls starting with http" cannot be optimized. An earlier work [25,13] suggested to apply static program analysis as in compiler optimization to tame the overhead of code instrumentation.

[3] The (once) enthusiast installers of UK Channel 4 on demand services 4oD [3] might also tell that even a branded name is not a surrogate for trust[34].

An application is only allowed to perform a or m followed by b or n and then repeat the sequence of efg.

(a) Rule of a policy

An application never uses n, m. It performs a followed by b and then repeat the sequence of efg.

(b) Rule of a contract

An application is only allowed to perform a followed by b and then it can do whatever, i.e. following the contract.

(c) Rule of an optimized policy

Fig. 1. Example of Optimization

To illustrate how an optimization works we use a simple alphabet $\{a, b, e, f, g, m, n\}$ that represent security relevant behaviors. For example we have a rule for a contract(Figure 1(b)) and a rule for a policy(Figure 1(a)), then the optimized rule of the policy is represented in Figure 1(c).

1.1 *The Contributions of the Paper*

We proposed to use more expressive policies for IRM optimization by using automata theory. First, we identified the different trust models for IRM optimization, i.e. the relative position of the optimizer and the inliner with respect to the trust border.

The formal model used for capturing contracts and policies is based on a concept called *Automata Modulo Theory* ($\mathcal{AMT}$). $\mathcal{AMT}$ generalizes the finite state automata of model-carrying code [37] and extends Büchi Automata (BA). It is suitable for formalizing systems with finitely many states but infinitely many transitions, by leveraging the power of satisfiability-modulo-theory (for short SMT) decision procedures. $\mathcal{AMT}$ enables us to define very expressive and customizable policies, by capturing the infinite transition into finite transitions labeled as expressions in suitable theories. We map the problem of optimizing IRM into automata theoretic construction of intersection and simulation.

In the next section, we introduce optimization models for IRM and discuss how the component of this model can be implemented by a number of related works (§2). We continue by briefly recapping the notion of $\mathcal{AMT}$ and the relevant operations in (§3). We continue with fair simulation for $\mathcal{AMT}$ (§4) as the basic block for our algorithm in (§5).

2 Security Models for Optimized IRM

In this section, we introduce our IRM trust models. Figure 2(b) illustrates our general optimization workflow model. This model is a modification of S×C workflow (Figure 2(a) [8]) by adding optimization step. First, a code is analyzed in order to extract contract out of it. This can be done by trusted or untrusted parties. If done by untrusted parties, then the claimed contract needs to be verified whether it complies to the code. If it complies, then we simulate the contract with the policy to verify if the policy is already enforced by the contract. On failure of simulation, we optimize policy by discharging behaviors which are already enforced by a contract and we inject this optimized policy to the code. The overall model (Figure 2(b)) consists of the following components:

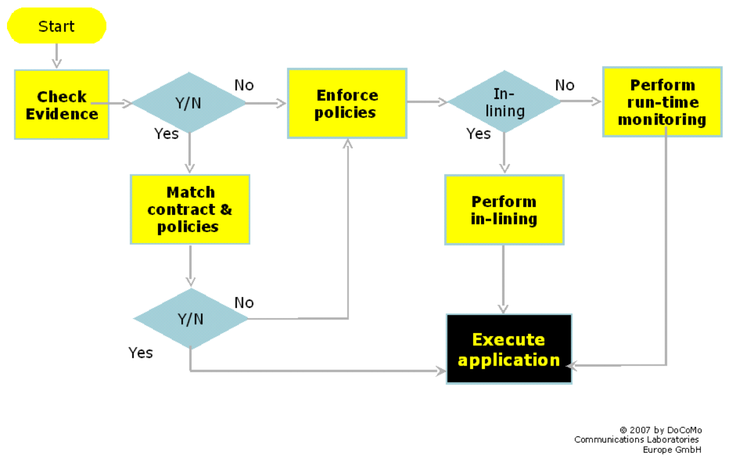
ContractExtractor and ClaimChecker The former extract policies from code based on control flow graphs and possibly annotation existing on the code [17]. The latter is the basic component of Proof-Carrying Code [33], where the untrusted code supplier must provide with the code a safety proof that attests to the code's safety properties. In mobile system domain [21] implements a linear decision algorithm verifying that annotated .NET binaries satisfy a class of policies using security and edit automata.

SimulationChecker uses fair simulation for $\mathcal{AMT}$ [31]. This key idea is based on symbolic simulation [14,26]. A system fairly simulates another system if and only if in the simulation game, there is a strategy that matches with each fair computation of the simulated system a fair computation of the simulating system. We can use this techniques to decide if the update is acceptable by different notion of simulation.

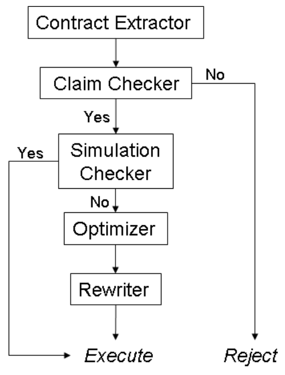
Rewriter We use rewriter instead of inliner because it is not necessary to actually inline the entire security automaton. Some example of works on rewriter are Naccio [15], PoET/Pslang [12], and Polymer [28]. These approaches compile policy language into plain Java and then into Java bytecode monitor which is injected into ordinary Java bytecode by inserting calls in all the necessary places. Other rewriter uses reflection [38] where policies are implemented as meta-objects bounded to application objects at load time through bytecode rewriting. This approach is implemented using Kava which provides a non-bypassable meta level. An alternative approach to rewriter is an inliner, for example [10] that only inlines hooks to the monitor with the monitor itself runs in a separate thread.

Optimizer can be performed by compiler optimization approach as in [39] or by our approach described in (§5).

The IRM approach is facilitated by the trend toward using higher-level languages, especially type safe languages, for software development. Not only do those languages define application abstractions on which policies can be enforced, but they also provide strong guarantees that can be used to ensure a secured application cannot compromise its IRM. By leveraging these guarantees, an IRM security policy can provide a single cohesive description of both the intent and the means by which a policy is enforced. This potentially allows the IRM approach to give greater assurance, since enforcement now relies on a trustworthy component of moderate size

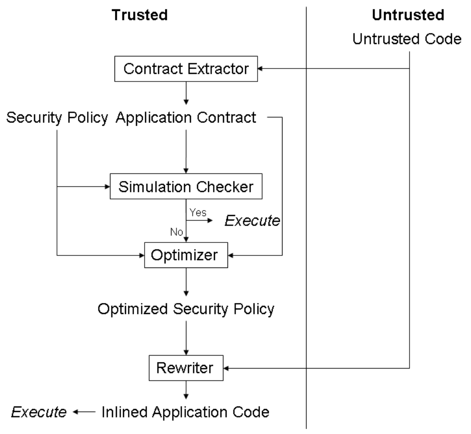

Start
Check Evidence
Y/N
No
Yes
Enforce policies
In-lining
No
Yes
Perform run-time monitoring
Match contract & policies
Perform in-lining
Y/N
No
Yes
Execute application
© 2007 by DoCoMo Communications Laboratories Europe GmbH

(a) S×C WorkFlow

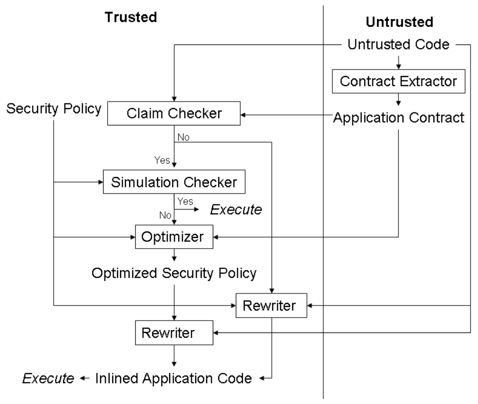


(b) Optimization WorkFlow

Fig. 2. S×C modified as Optimization WorkFlow



(a) Contract Extractor on Trusted part



(b) Contract Extractor on Untrusted part

Fig. 3. Rewriter on Trusted part

whose full specification can be studied in isolation.

The main consideration for our models is the trade off between moving more processes out of trusted part and the complexity of the whole process (inspired by model in [20]).

In the simplest model (Fig. 3(a)), the Untrusted part consists of only `Code`. First, we extract the application contract `Claim` using ContractExtractor on the trusted part. Then, we check whether this result can be simulated by the security policy `Policy` using the SimulationChecker. If the simulation succeed, then we can execute the code without further ado. Otherwise, we optimize the enforcement mechanism by using an Optimizer which gives result `OptPolicy`. The last procedure is the Rewriter which gives result an `SafeCode` that is ready to be executed.

We are interested in extracting *security relevant* behaviors [37], that can be performed by data flow analysis [4], control flow analysis [35], abstract interpretation [9], or model extraction [37]. After extraction, we check whether this result can be

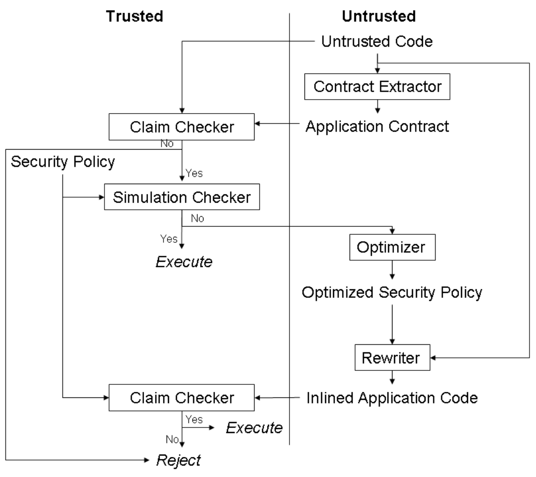


Fig. 4. Optimizer and Rewriter on Untrusted part: Contract Extractor on Untrusted part

simulated by `Policy`(P) using SimulationChecker, which can be implemented as automata simulation [31]. If the simulation succeed, then we can execute the code without further ado. Otherwise, we optimize the IRM using Optimizer which gives result `OptPolicy`$(OptP)$.

In the second model (Fig. 3(b)), the Untrusted part consists of UC and C which is extracted using ContractExtractor. We check C against UC using ClaimChecker. This step is needed because we cannot trust C. Some works have been developed along this line, for example the concept of type-checker in [21], signature verifier in [16], weakest precondition based annotation checker [5] specified with ConSpec language[6], or *Proof-Carrying Code* [33]. If C does not comply to the UC, then we will not make any optimization and will directly inline P into UC using Rewriter which gives result an IC that is ready to be executed (*Execute*). In the case that C complies, we run SimulationChecker and the path goes through as of the first model.

The third model is similar to our first model by moving Rewriter to Untrusted part. This approach is similar to [39] where the IRM optimization framework can also be distributed with an untrusted code producer involves to make optimization effective. The Untrusted part consists of UC and IC. The fourth model is similar to our second model by moving Rewriter to Untrusted part. The Untrusted part consists of UC, C which is extracted using ContractExtractor, and IC. Moving more components out of the trusted computing base is desirable. Thus, we propose the fifth model which is similar to our third model by moving Optimizer also to Untrusted part. Now, the Untrusted part consists of UC, $OptP$ and IC.

On our sixth model (Fig. 4) we move most of the components out of the trusted domain. After running the ContractExtractor, we check the application contract `Claim` against the application `Code` using the ClaimChecker. If the `Claim` does not comply to `Code`, then we will reject `Code`. Rejection might be too restrictive, thus another option is similar to deploy directly the `Policy` object in charge on monitoring in `Code` by using the Rewriter which gives result an `SafeCode`.

3 Automata Modulo Theory

The security behaviors of a code (C) and the desired behaviors by platform (P), can be represented as automata, where transitions corresponds to invocation of APIs as suggested by Erlingsson [12, p.59] and Sekar et al. [37]. While this idea of representing the security-digest as an automaton is almost a decade old, the practical realization has been hindered by a significant technical hurdle: we cannot use the naive encoding into automata for practical policies. For example a policy "connect only to urls starting with https" leads to automata with infinitely many transitions.

To overcome this limitation we have introduced *Automata Modulo Theory* ($\mathcal{AMT}$) in [30]. $\mathcal{AMT}$ is a combination of the theory of Büchi Automata (BA) with the Satisfiability Modulo Theories (SMT) problem. We prefer to use BA instead of security automata because we are interested in verifying safety, liveness and renewable properties (see[29,22] for characterization of security policies enforceable by program rewriting).

To make this paper self contain, we briefly discuss $\mathcal{AMT}$ concept and $\mathcal{AMT}$ simulation and we refer to [30,31] for details.

Definition 3.1 [Automaton Modulo Theory ($\mathcal{AMT}$)] A tuple $A = \langle E, S, \mathbf{s_0}, \Delta, F\rangle$ where E is a set of Σ-formulas in Σ-theory $\mathcal{T}$, S is a finite set of states, $\mathbf{s_0} \in S$ is the initial state, $\Delta \subseteq S \times E \times S$ is a labeled transition relation, and $F \subseteq S$ is a set of accepting states.

To understand the semantics of an automaton modulo theory, we need to consider the corresponding concrete automaton. A concrete automaton is constructed by replacing each transition (labeled with an expression from the theory), with the infinitely many transitions (labeled by the corresponding satisfying assignments). The concrete runs of the system are the actual system traces of values of invoked APIs which are represented by assignments.

Definition 3.2 [$\mathcal{AMT}$ concrete run] Let $A = \langle E, S, \mathbf{s_0}, \Delta, F\rangle$ be an $\mathcal{AMT}$. A *concrete run* of A is a sequence of states alternating with assignments $\sigma_C = \langle s_0\alpha_0 s_1\alpha_1 s_2\alpha_2 \ldots\rangle$, such that: i) $s_0 = \mathbf{s_0}$, ii) there exists expressions $e_i \in E$, where $(s_i, e_i, s_{i+1}) \in \Delta$ and $(\mathcal{A}, \alpha_i) \models e_i$ holds for all $i \in [0 \ldots |w|]$. A finite run is accepting if $s_{|w|}$ is an accepting state and an infinite run is accepting if the automaton goes through some accepting states infinitely often. [4]

Our notion of symbolic run corresponds to the traditional notion of run in automata.

Definition 3.3 [$\mathcal{AMT}$ symbolic run] Let $A = \langle E, S, \mathbf{s_0}, \Delta, F\rangle$ be an automaton modulo theory $\mathcal{T}$. A *symbolic run modulo* $\mathcal{T}$ of A is a sequence of states alternating with expressions $\sigma_, = \langle s_0 e_0 s_1 e_1 s_2 e_2 \ldots\rangle$ such that: i) $s_0 = \mathbf{s_0}$, ii) $(s_i, e_i, s_{i+1}) \in \Delta$ and $(\mathcal{A}, \alpha_{ij}) \models e_i$ holds for some j, with runs acceptance as in concrete runs.

The transition function of A may have many possible transitions for each state and expression, hence A may be non-deterministic.

[4] We use definition of run as in [14] which is slightly different from the one we use in [30], where we use only states.

Definition 3.4 [Deterministic $\mathcal{AMT}$] $A = \langle E, S, \mathbf{s_0}, \Delta, F\rangle$ is a *deterministic automaton* modulo theory $\mathcal{T}$, if and only if, for every $s \in S$ and every $s_1, s_2 \in S$ and every $e_1, e_2 \in E$, if $(s, e_1, s_1) \in \Delta$ and $(s, e_2, s_2) \in \Delta$, where $s_1 \neq s_2$ then $e_1 \wedge e_2$ is unsatisfiable modulo $\mathcal{T}$.

$\mathcal{AMT}$ intersection is defined over symbolic level, hence we use satisfiability on formulas $(DP(e^a \wedge e^b) = SAT)$ which corresponds to equality of alphabet in concrete level.

Definition 3.5 [$\mathcal{AMT}$ Intersection] Let $\langle E, S^a, \mathbf{s_0}^a, \Delta^a_{\mathcal{T}}, F^a\rangle$ and $\langle E, S^b, \mathbf{s_0}^b, \Delta^b_{\mathcal{T}}, F^b\rangle$ be (non) deterministic $\mathcal{AMT}$, the $\mathcal{AMT}$ *intersection automaton* $A^{ab} = \langle E, S, \mathbf{s_0}, \Delta, F\rangle$ is defined as follows:

$$S = S^a \times S^b \times \{1,2\} \tag{1}$$

$$\mathbf{s_0} = \left\langle \mathbf{s_0}^a, \mathbf{s_0}^b, 1 \right\rangle \tag{2}$$

$$F = F^a \times S^b \times \{1\} \tag{3}$$

$$\Delta = \left\{ \left\langle (s^a, s^b, x), e^a \wedge e^b, (t^a, t^b, y) \right\rangle \middle| \begin{array}{l} (s^a, e^a, t^a) \in \Delta^a \text{ and} \\ (s^b, e^b, t^b) \in \Delta^b \text{ and} \\ DP(e^a \wedge e^b) = SAT \text{ and} \\ \textbf{marker condition} \end{array} \right\} \tag{4}$$

where **marker condition** is defined as follows: if $s^a \in F^a \wedge x = 1$, then $y = 2$; if $s^a \notin F^a \wedge x = 1$, then $y = 1$; if $s^b \in F^b \wedge x = 2$, then $y = 1$, otherwise $x = y$.

4 Simulation

The SimulationChecker and Optimizer use fair simulation defined in game graph. Therefore, in this section we introduce the notion of simulation at the concrete level, among assignments i.e. API calls, and we continue giving the notion of symbolic simulation as in [23]. The actual notion of fair simulation is adapted from [14,19,24].

Definition 4.1 [Concrete Fair Compliance Game] Let A^c and A^p be $\mathcal{AMT}$ with initial states s_0 and t_0 respectively. A *Concrete Fair Compliance Game* $G^C_{A^c,A^p}(s_0, t_0)$ is played by two players, C and P, in rounds.

(i) In the first round C is on the initial state $s_0 \in S^c$ and P is on the initial state $t_0 \in S^p$.

(ii) C chooses a transition $\langle s_i, e^c_i, s_{i+1}\rangle \in \Delta^c_{\mathcal{T}}$ and an assignment α_i such that $(\mathcal{A}, \alpha_i) \models e^c_i)$ and moves to s_{i+1}.

(iii) P responds by a transition $\langle t_i, e^p_i, t_{i+1}\rangle \in \Delta^p_{\mathcal{T}}$ such that $(\mathcal{A}, \alpha_i) \models e^p_i)$ and moves to t_{i+1}.

The winner of the game is determined by the following rules:

- If the C cannot move then P wins.
- If the P cannot move then C wins.

- Otherwise there are two infinite concrete runs $\overrightarrow{s} = \langle s_0\alpha_0 s_1\alpha_1 s_2\alpha_2,\ldots\rangle$ and $\overrightarrow{t} = \langle t_0\alpha_0 t_1\alpha_1 t_2\alpha_2,\ldots\rangle$ respectively of A^c and A^p. If $\overrightarrow{s} = \langle s_0\alpha_0 s_1\alpha_1 s_2\alpha_2,\ldots\rangle$ is an accepting concrete run for A^c and $\overrightarrow{t} = \langle t_0\alpha_0 t_1\alpha_1 t_2\alpha_2,\ldots\rangle$ is not an accepting concrete run for A^p then C wins. In other cases, P wins.

Intuitively in the compliance game, C tries to make a move and the P follows accordingly to show that the C move is allowed. If the P cannot move then C is not compliant: there is a move that the P could not do and that particular action is a violation. Next, we generalize the notion of simulation to symbolic level, among expressions.

Definition 4.2 [$\mathcal{AMT}$ Fair Compliance Game] A *Fair Compliance Game* $G_{A^c,A^p}(s_0,t_0)$ is played by two players, C and P, in rounds.

(i) In the first round C is on the initial state $s_0 \in S^c$ and P is on the initial state $t_0 \in S^p$.

(ii) C chooses a transition $\langle s_i, e^c_i, s_{i+1}\rangle \in \Delta^c_{\mathcal{T}}$ where e^c_i is satisfiable.

(iii) P responds by a transition $\Delta^p_{\mathcal{T}}(t_i, e^p_i, t_{i+1})$ such that $e^c_i \rightarrow e^p_i$ is valid.

The winner of the game is determined by the rules as in Definition 4.1 with the difference in run where we define run over expressions instead of assignments.

In $\mathcal{AMT}$ Fair Compliance Game C tries to make a symbolic move and the P follows suit in order to show that the C move is allowed. If the P cannot move this means that the C may not be compliant because there is a symbolic move that the P could not do. As in [31], for simulation algorithm we adapts the Jurdzinski's algorithm on parity games [26].

Definition 4.3 [Compliance Graph] Let V_0 and V_1 be two disjoint sets, a *compliance graph* G is a tuple $\langle V_1, V_0, E, l\rangle$, where $V = V_0 \cup V_1$, $E \subseteq V \times V$, and $l : V \rightarrow \{0,1,2\}$.

Intuitively the compliance level l(v) is 0 when the simulating automaton accepts, 1 when the simulated automaton accepts (but the simulating automaton has not accepted yet) and 2 when neither of them accepts.

A compliance game $P(G, v_0)$ on G starting at $v_0 \in V$ is played by two players P and C. The game starts by placing pebble on v_0. At round i with pebble on v_i, $v_i \in V_0(V_1)$, P (C resp.) plays and moves the pebble to v_{i+1} such that $(v_i, v_{i+1}) \in E$. The player who cannot move loses. For infinite play $\pi = v_0 v_1 v_2 \ldots$, the winner defined as the minimum compliance level that occurs infinitely often, namely if the minimum compliance level is 0 or 2 then P wins, otherwise C wins.

We apply this compliance game to $\mathcal{AMT}$ such that given $\langle E, S^c, \mathbf{s_0}^c, \Delta^c_{\mathcal{T}}, F^c\rangle$ and $\langle E, S^p, \mathbf{s_0}^p, \Delta^p_{\mathcal{T}}, F^p\rangle$, we construct a $\langle V_1, V_0, E, l\rangle$ as follows:

- $V_1 = \{v_{(s^c,s^p)} | s^c \in S^c, s^p \in S^p\}$
- $V_0 = \{v_{(s^c,s^p,e^c)} | s^c \in S^c, s^p \in S^p, \exists r^c.s^c \in \Delta^c_{\mathcal{T}}(r^c, e^c)\}$
- $E = \{(v_{(s^c,s^p,e^c)}, v_{(s^c,t^p)}) | t^p \in \Delta^c_{\mathcal{T}}(s^p, e^p) \wedge VALID(e^c \rightarrow e^p)\} \cup \{(v_{(s^c,s^p)}, v_{(t^c,s^p,e^c)}) | t^c \in \Delta^c_{\mathcal{T}}(s^c, e^c)\}$

Algorithm 1 Simulation Algorithm

```
Input: two $\mathcal{AMT}$ automata (policy $C$, policy $P$)
1:  Construct compliance game graph $G = \langle V_1, V_0, E, l\rangle$
2:  for all $v \in V$ do
3:      $\mu(v) := \mu_{\mathtt{new}}(v) := 0$
4:  repeat
5:      $\mu := \mu_{\mathtt{new}}$
6:      for all $v \in V_0$ do
7:          $\mu_{\mathtt{new}}(v) := \begin{cases} \infty & \text{if } \{\mu(w)|(v,w)\} = \emptyset \\ min\,\{\mu(w)|(v,w)\} & \text{otherwise} \end{cases}$
8:      for all $v \in V_1$ do
9:          $max_v := max\,\{\mu(w)|(v,w) \in E\}$
10:         $\mu_{\mathtt{new}}(v) := \begin{cases} \infty & \text{if } max_v = \infty \\ 0 & \text{if } l(v) = 0 \\ max_v + 1 & \text{if } l(v) = 1 \\ max_v & \text{if } l(v) = 2 \end{cases}$
11: until $\mu = \mu_{\mathtt{new}}$
12: if $\mu(v_{(\mathbf{s_0}^c,\mathbf{s_0}^p)}) < \infty$ then
13:     Simulation exists
```

•

$$l(v) = \begin{cases} 0 \text{ if } v = v_{(s^c,s^p)} \text{ and } s^p \in F^p \\ 1 \text{ if } v = v_{(s^c,s^p)} \text{ and } s^c \in F^c \text{ and } s^p \notin F^p \\ 2 \text{ otherwise} \end{cases}$$

Next, we define a *compliance measure* $\mu : V \to \{x|x \leq |l^{-1}(1)|\} \cup \{\infty\}$. μ ranges from 0 to $|l^{-1}(1)|$ because at l(v)=1 the simulated automaton (contract) accepts but the simulating automaton (policy) has not accepted yet. Thus, progressing the measure has the analogy of computing the fix point where the C remains winning and ∞ shows that the μ keeps progressing beyond this limit, meaning C wins the game. If $l(v) = 1$, then $\mu(v) > \mu(w)$, where $|l^{-1}(1)|+1 = \infty$. If $l(v) = 2$ or $l(v) = 0$, then $\mu(v) \geq \mu(w)$.

The compliance measure for each node is the number of potential bad nodes, namely nodes where the contract accepts but the policy does not, that it can reach. Thus, $\mu(v) = \infty$ means that there is an infinite path where policy cannot return to compliance level 0.

5 A Search Procedure for IRM Optimization

The problem of searching an optimized policy can be stated intuitively as follows: given two automata C and P representing respectively the formal specification of a contract and of a policy, we have an efficient IRM *OptP* derived from P with respect to C when:

(i) every APIs invoked by the intersection of $OptP$ and C can also be invoked by P, and

(ii) $OptP$ has smaller number of transitions or states than P with respect to C.

The most relaxed P can be obtained by allowing anything to happen, we call it P^*. Prior to construction of P^*, we remove edges of non existing alphabet from contract. To construct P^*, for each edge in P we replace edge with $*$ (corresponding to true) and check that it remains compliant, i.e. P can simulate the intersection (3.5) between C and P^*. If simulation fails, we put the original edge in P^*. This step fulfills the property i and to fulfill property ii we use $\mathcal{AMT}$ minimization with input P^* and use the approach as in [19]. First, we try to merge equivalence states and then remove redundant edges in P^*.

Merge equivalence states. We initialize the set of states to be merged as the set of all states of P^*. Then, we construct a game graph where C and P are the same automaton, i.e. P^*. We try to merge states by picking two states s_{opt}, s'_{opt}. Two states can be merged when they have the same incoming transitions and outgoing transitions and behave the same with the same input. Thus, we add outgoing transitions of s_{opt} to s'_{opt} and vice versa. We also add incoming transitions of s_{opt} to s'_{opt} and vice versa. Then, we recompute game graph by adding these transitions and re-run game. If succeed then we merge $s'_{opt}$and s_{opt}.

Remove redundant transitions. We initialize the set of transitions to be removed as the set of all transitions from the automaton $P^{*'}$, i.e. P^* after merging equivalent states. Then, we construct a game graph where C and P are the same automaton, i.e. $P^{*'}$. We try to remove a transition by picking a transition (u, e, v) in $P^{*'}$ where there exists a w that simulates v and (u, e, w) is also in $P^{*'}$. When we remove a transition in $P^{*'}$, we also remove some edges in the game graph that correspond to the removed transition. Thus we need not re-run simulation from scratch, but we can compute our game incrementally. So, we save the resulted μ in a vector and each time we re-run simulation we initialize our game from this computed μ instead of 0.e recompute game graph by removing these transitions and re-run game. If simulation succeed then we remove (u, e, v).

6 Conclusions

The main goal of this work has been to provide an answer to the following question: *given an untrusted code and a policy that a platform specifies to be inlined, how can we obtain an optimized IRM ?* To address this issue we have proposed six different framework models for IRM optimization with respect to components that are needed to be trusted or untrusted. We have also described an approach for IRM optimization based on automata theory. The key idea is that given a *policy* that represent the desired security behavior of a platform to be inlined, we compute an *optimized policy* with respect to the claims on the security behavior of a application that we inject to the untrusted code.

Future work will include comparative study of IRM with or without optimization and the effect of changes both in frequency (how often a code modified) and size (how much a code modified).

References

[1] *Card specification version 2.2*, Technical report, GlobalPlatform (2006), report available at www.globalplatform.org.

[2] *Confidential card content management card specification v 2.2 - amendment a*, Public Release GPC_SPE_007, GlobalPlatform (2007), report available at www.globalplatform.org.

[3] 4, C., *4od*, Available on the web http://www.channel4.com/4od/index.html (2008).

[4] Aho, A., R. Sethi and J. Ullman, "Compilers: principles, techniques, and tools," Addison-Wesley, 1986.

[5] Aktug, I., M. Dam and D. Gurov, *Provably correct runtime monitoring*, J. of Logic and Algebraic Programming (2009).

[6] Aktug, I. and K. Naliuka, *Conspec - a formal language for policy specification*, Proc. of the 1st Int. Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM2007) (2007).

[7] Bauer, L., J. Ligatti and D. Walker, *Edit automata: Enforcement mechanisms for run-time security policies*, Int. J. of Inform. Sec. **4** (2005), pp. 2–16.
URL `http://www.ece.cmu.edu/~lbauer/papers/editauto-ijis05.pdf`

[8] Bielova, N., N. Dragoni, F. Massacci, K. Naliuka and I. Siahaan., *Matching in security-by-contract for mobile code*, J. of Logic and Algebraic Programming (2009), to Appear.

[9] Cousot, P. and R. Cousot, *Abstract interpretation frameworks*, J. of Logic and Computation **2** (1992), pp. 511–547.

[10] Desmet, L., W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan and D. Vanoverberghe, *Security-by-contract on the .NET platform*, Information Security Tech. Rep. **13** (2008), pp. 25 – 32.

[11] Dragoni, N., F. Massacci, K. Naliuka and I. Siahaan, *Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code*, in: *Proc. of the 4th European PKI Workshop Theory and Practice (EUROPKI'07)* (2007), p. 297.

[12] Erlingsson, U., "The Inlined Reference Monitor Approach to Security Policy Enforcement," Ph.D. thesis, Department of Computer Science, Cornell University (2004).

[13] Erlingsson, U. and F. Schneider, *IRM enforcement of Java stack inspection*, in: *Proc. of the 2000 IEEE Symp. on Security and Privacy*, 2000, pp. 246–255.

[14] Etessami, K., T. Wilke and R. Schuller, *Fair simulation relations, parity games, and state space reduction for büchi automata*, SIAM J. on Comp. **34** (2005), pp. 1159–1175.

[15] Evans, D., "Policy-Directed Code Safety," Ph.D. thesis, MIT (1999).

[16] Ghindici, D., G. Grimaud and I. Simplot-Ryl, *An information flow verifier for small embedded systems*, in: D. S. et al., editor, *Proc. Workshop in Information Security Theory and Practices: Smart Cards, Mobile and Ubiquitous Computing Systems (WISTP'07)*, LNCS **4462** (2007), pp. 189–201.

[17] Ghindici, D., I. Simplot-Ryl and J.-M. Talbot, *A sound analysis for secure information flow using abstract memory graphs*, in: *The 3rd Int. Conf. on Fundamentals of Sw. Eng. (FSEN'09)*, 2009.

[18] Gong, L., G. Ellison and M. Dageforde, "Inside Java 2 Platform Security: Architecture, Api Design, and Implementation," Addison-Wesley Professional, 2003.

[19] Gurumurthy, S., R. Bloem and F. Somenzi, *Fair simulation minimization*, in: *Proc. of the 14th Int. Conf. on Computer Aided Verification (CAV'02)* (2002), pp. 610–624.

[20] Hamlen, K., "Security policy enforcement by automated program-rewriting," Ph.D. thesis, Cornell University (2006).

[21] Hamlen, K., G. Morrisett and F. Schneider, *Certified in-lined reference monitoring on .net*, in: *Proc. of the 2006 workshop on Prog. Lang. and analysis for security* (2006), pp. 7–16.

[22] Hamlen, K. W., G. Morrisett and F. B. Schneider, *Computability classes for enforcement mechanisms*, ACM Trans. Program. Lang. Syst. **28** (2006), pp. 175–205.

[23] Hennessy, M. and H. Lin, *Symbolic bisimulations*, in: *MFPS'92: Selected papers of the meeting on Math. Foundations of Programming Semantics* (1995), pp. 353–389.

[24] Henzinger, T., O. Kupferman and S. Rajamani, *Fair simulation*, in: *Proc. of of the 8th Int. Conf. on Concurrency Theory* (1997), pp. 273–287.

[25] Jeffery, C., W. Zhou, K. Templer and M. Brazell, *A lightweight architecture for program execution monitoring*, ACM SIGPLAN Notices **33** (1998), pp. 67–74.

[26] Jurdzinski, M., *Small progress measures for solving parity games*, in: *STACS '00: Proc. of the 17th Annual ACM Symposium on Theoretical Aspects of Computer Science* (2000), pp. 290–301.

[27] LaMacchia, B. and S. Lange, ".NET Framework security," Addison Wesley, 2002.

[28] Ligatti, J., "Policy Enforcement via Program Monitoring." Ph.D. thesis, Princeton University (2006).

[29] Ligatti, J., L. Bauer and D. Walker, *Run-time enforcement of nonsafety policies*, ACM Trans. on Inf. and Syst. Security **12** (2009), pp. 1–41.

[30] Massacci, F. and I. Siahaan., *Matching midlet's security claims with a platform security policy using automata modulo theory*, in: *Proc. of the 12th Nordic Workshop on Secure IT Systems (NordSec'07)*, 2007.

[31] Massacci, F. and I. Siahaan., *Simulating midlet's security claims with automata modulo theory*, in: *Proc. of the 2008 workshop on Prog. Lang. and analysis for security*, 2008, pp. 1–9.

[32] Mayes, K. and K. Markantonakis, "Smart Cards, Tokens, Security and Applications," Springer-Verlag, 2008.

[33] Necula, G., *Proof-carrying code*, in: *Proc. of the 24th ACM SIGPLAN-SIGACT Symp. on Princ. of Prog. Lang.* (1997), pp. 106–119.

[34] Networks, C., *Channel 4's 4od: Tv on demand, at a price.*, Crave Webzine (2007).

[35] Nielson, F. and H. Nielson, *Flow logic for imperative objects*, in: *Proc. of the 23rd Int. Symp. on Math. Foundations of Comp. Scie.* (1998), pp. 220–228.

[36] Phung, P., D. Sands and A. Chudnov, *Lightweight Self-Protecting JavaScript*, in: *Proc. of the 4th ACM Symposium on Information Comp. and Comm. Sec. (ASIACCS 2009)*, 2009, pp. 10–12.

[37] Sekar, R., V. Venkatakrishnan, S. Basu, S. Bhatkar and D. DuVarney, *Model-carrying code: a practical approach for safe execution of untrusted applications*, in: *Proc. of the 19th ACM Symp. on Operating Syst. Princ.* (2003), pp. 15–28.

[38] Welch, I. and R. Stroud, *Using reflection as a mechanism for enforcing security policies on compiled code*, J. of Comp. Sec. **10** (2002), pp. 399–432.

[39] Yan, F. and P. W. L. Fong, *Efficient IRM Enforcement of History-Based Access Control Policies*, in: *Proc. of the 4th ACM Symposium on Information Comp. and Comm. Sec. (ASIACCS 2009)*, 2009, pp. 35–46.