Trust-Driven Policy Enforcement through Gate Automata

G. Costa, Ilaria Matteucci

Trust-Driven Policy Enforcement through Gate Automata

Gabriele Costa
Università di Pisa
and IIT-CNR
costa@di.unipi.it

Ilaria Matteucci
IIT-CNR
ilaria.matteucci@iit.cnr.it

***Abstract*—In this paper we introduce the notion of *gate automata* for describing security policies. This new kind of automata aim at defining a model for the specification of both security and trust policies.**

The main novelty of our proposal is a unified framework for the integration of security enforcement and trust monitoring. Indeed, gate automata watch the execution of a target program, possibly modifying its behaviour, and produce a feedback for the trust management system. The level of trust changes the environment settings by dynamically activating/deactivating some of the defined gate automata.

Keyword: Gate automata, interface automata, security automata, run-time enforcement, Security-by-Contract-with-Trust.

I. Introduction

The influence of the digital devices on our everyday life is increasing over and over. From a merely technological point of view, the new generation mobile devices are reducing the gap with the personal computers. They are becoming the main access point for the network and their standard software equipment can be easily extended by downloading and running new applications. Hence, the importance of protecting the mobile devices in the same way we protect our personal computer arises.

In the last few years a lot of work has been done for defining run-time enforcement mechanisms for securing applications coming from untrusted software providers, *e.g.*, [10], [15], [18]. However, at the best of our knowledge, few works deal with the full integration of trust management and policy enforcement for mobile code. Nevertheless, some work about policy enforcement environment extended with trust management module exists, *e.g.*, see [4], [11].

In this work, we refer to the Security-by-Contract-with-Trust paradigm (S×C×T) [8], [9] as run-time enforcement mechanism based on both security and trust notion. In particular, we propose a novel strategy for enforcing security properties. To do this, we introduce the notion of *gate automata*, a novel category of security automata that combines several well-known features. Indeed, the automata-based specification makes it simpler to reason about security requirements. For instance, *security automata* [17] are a major proposal for the specification of safety properties. A wider class of security properties can be modelled by passing to an enforcement environment that also *edits* the behaviour of its target, *e.g.*, see [13].

Furthermore, we present how gate automata can replace the enforcement mechanism of the S×C×T obtaining an optimization of the enforcement process based on trust levels. Basically, we show that this class of automata works as edit automata [13] for guaranteeing security properties and handles the trust levels of the applications. In this way, the users can customise their security requirements and the management of the trust levels in their systems.

Hence gate automata represent an integrated formalism for defining security and trust policies in an intuitive way. Indeed, gate automata allow to specify security policies that also affect the trust levels of the system. When we download an application, according to the provider identity, we associate it to a certain level of trust. The smaller is the level of trust the stronger, *i.e.*, the more restrictive, is the security policy we enforce on the application. While the application is executed, the run-time enforcement mechanism controls its behaviour. If a violation occurs, *i.e.*, the application tries to perform an action that is not allowed by the current security policy, the enforcement system reacts, possibly decreasing the level of trust of the application. This may lead to an automatic update of the run-time enforcement mechanism that, after the reconfiguration, enforces a different, *e.g.*, more restrictive, security policy.

This paper is organized as follows: in the next section we recall the Security-by-Contract-with-Trust paradigm. Section III formally introduces gate automata and compares them with other automata-based specifications. Section IV shows the novel run-time enforcement strategy obtained by using gate automata. Section V compares our approach with already existing related works and Section VI leads to the conclusion of the paper and provides some future research directions.

II. Security-by-Contract-with-Trust

The Security-by-Contract-with-Trust paradigm, S×C×T for short, has been introduced in [8], [9] as a unique framework for managing both security and trust at application

Work partially supported by EU-funded project FP7-231167 CONNECT, by EU-funded project FP7-257930 ANIKETOS and EU-funded project FP7-256980 NESSoS.

execution time. As in its previous version without trust, the S×C [6], it is based on the two concept of *contract* of an application and *policy* of a platform. Intuitively, the *contract* is associated with an application and consists in the description of the behaviour of the application itself. On the other hand, the *policy* is set on the platform that is in charge to run applications and it is a description of all possible application's behaviours allowed by the platform. It can be written by the owner of the platform or by the vendor.

The Security-by-Contract-with-Trust workflow is depicted in Figure 1. The basic idea is the following one: let us consider to have a platform, *e.g.*, a desktop, a laptop, a mobile phone, and so on, and we want to run on it an application, developed by others, possible unknown, developers, in a secure way. We assume that this application is accompanied by its contract provided by its developer and, according to [3], [5], we also assume that both contracts and policies are specified through the same formalism. The application lifecycle consists in the following steps:

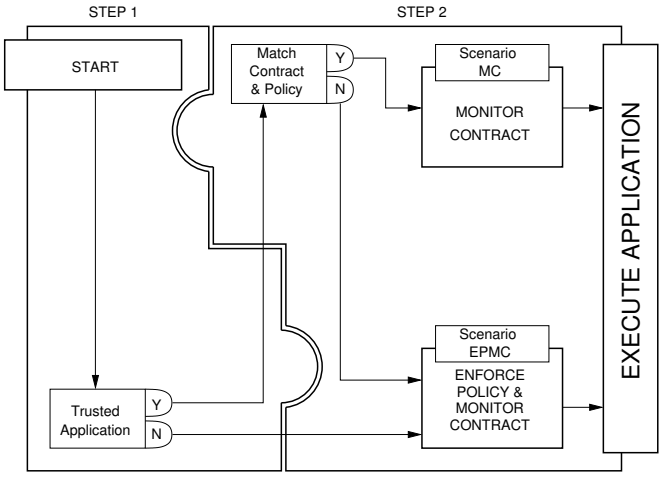


Figure 1: The Security-by-Contract-with-Trust Workflow.

Step 1. Once an application is downloaded on the platform, before executing it, the trust module decides if the application satisfies its contract according to a fixed trust threshold.
Step 2. According to this trust measure, the security module defines if just monitoring the contract or both enforce the policy and monitoring the contract going into one on the two scenarios described below. As a result of a contract monitoring strategy, the level of trust of the provider is updated. Our system penalizes the provider more when the contract does not specify application's behaviour correctly, rather when the application itself contradicts user's security policy. The two scenario we have works as follows:

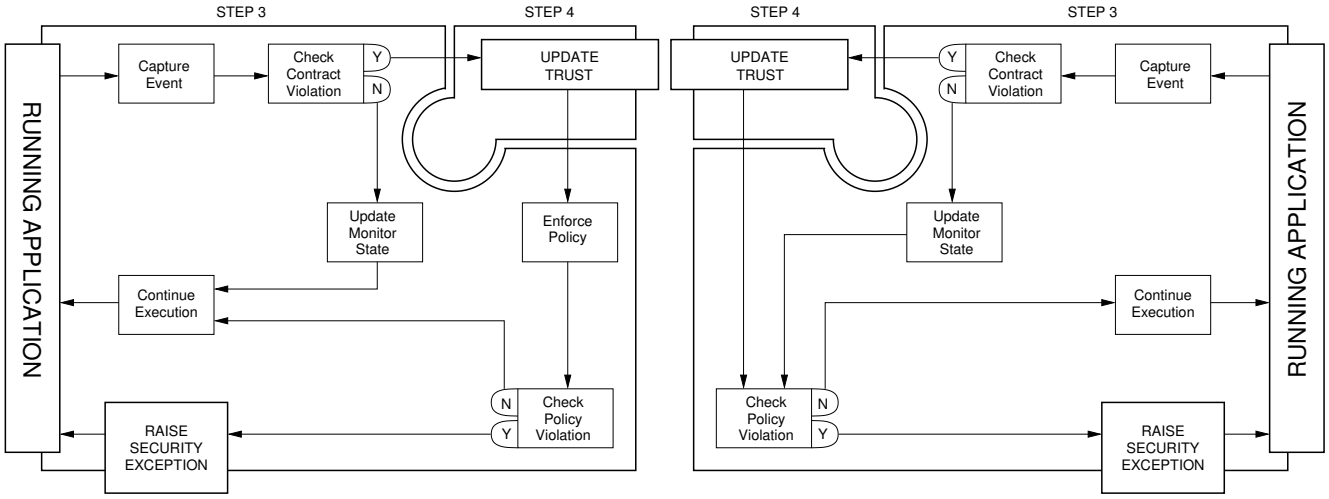
Scenario MC. *The contract satisfies the policy*. The monitoring/enforcement infrastructure is required to monitor only the application contract. Indeed, under these conditions, contract adherence also implies policy compliance. If no violation is detected then the application worked as expected. Otherwise, we discovered that a trusted party provided us with a fake contract. More in detail, the contract monitoring works according to the following strategy depicted in Figure 2a: the contract monitoring receives event signals from the executing code. The execution trace is kept in memory. When a signal arrives, its consistency with respect to the monitored contract is checked. If the contract is respected then its internal monitoring state is updated and the operation is allowed, and a good behaviour is logged (*i.e.*, contract respected). Otherwise, if a violation attempt happens, a security error occurs, and a bad feedback is trigged (*i.e.*, contract violation), and the system switches from contract monitoring to policy enforcement configuration in order to guarantee that the security policy is satisfied. Since an instance of the policy is always present, this operation does not imply a serious computational overhead.

Scenario EPMC. *The contract does not satisfy the policy*. Since the contract declares some potentially undesired behaviour, policy enforcement is turned on. Similarly to a pure enforcement framework, our system guarantees that executions are policy-compliant. However, monitoring contract during these executions can provide a useful feedback for better tuning the trust vector. Hence, in this scenario, both the policy enforcement and the contract monitoring are active. Indeed, the contract monitoring receives event signals from the executing code and keeps trace of the execution trace. When a signal arrives, its consistency with respect to the monitored contract is checked. If the contract is respected then its internal monitoring state is updated and the operation is allowed, and a good behaviour is logged (*i.e.*, contract respected). Otherwise, if a violation attempt happens, a security error occurs and a violation feedback is logged for the trust module. The policy enforcer is only in charge to following the execution of the application and whenever it attempts to violate the security policy of the device the enforcement mechanism halts the execution in such a way the security policy is satisfied. This configuration is activated on a statistical base (Figure 2b).

Summing up, both execution scenarios check contract violations through the contract monitoring strategy described above and update providers' trust level according to the contract monitoring feedback.

The goal of this paper is to propose a novel strategy for the two scenarios MC and EPMC based on a particular class of finite state automata, hereafter called *Gate automata*.

The basic idea is to define an integrated formalism for defining security and trust policies in a compositional and intuitive way. The main advantages of the proposed approach are the possibility of customizing the trust management and an optimization of the behaviours of both contract monitoring and policy enforcement.



(a) Scenario MC (b) Scenario EPMC

Figure 2: The contract monitoring configurations

III. Gate Automata

In this section we formally introduce *gate automata* and their properties. Moreover, we shall provide the reader with several examples showing how gate automata can be suitably used for specifying security policies.

A. Structure of the gate automata

We start by giving the formal definition of gate automata.

Definition 1. A gate automaton $\mathcal{G}$ is a 4-tuple $\langle V, \imath, A, T \rangle$ where:

- V is a finite set of states;
- $\imath \in V$ is the initial state;
- A is a set of actions;
- $T \subseteq V \times (A \cup \bar{A} \cup \{\blacktriangle, \blacktriangledown\}) \times V$ is a set of labelled transitions such that:
  1) $(v, a, u) \in T \wedge (v, b, w) \in T \wedge a = b \iff u = w$
  2) $\forall (v, a, u) \in T . a \in \bar{A} \cup \{\blacktriangle, \blacktriangledown\} \implies \nexists b, w . b \neq a \wedge (v, b, w) \in T$

Basically, a gate automaton slightly differs from a deterministic, finite state automaton. A gate automaton processes a sequence of actions possibly modifying it. The transitions of the automata can be labelled with input (*i.e.*, $\alpha \in A$) or output (*i.e.*, $\bar{\alpha} \in \bar{A}$) actions. An input action is generated by some actions source, *e.g.*, a running program, while output actions are fired by the automaton itself. Moreover, gate automata can perform two special operations, *i.e.*, $\blacktriangle$ and $\blacktriangledown$, that increase and decrease the trust weight corresponding to the source of the actions. Where it improves the readability we use $v \xrightarrow{\alpha} w$ in place of $(v, \alpha, w) \in T$ and $v \not\xrightarrow{\alpha}$ for $\nexists w \,.\, (v, \alpha, w) \in T$.

Example 1. Imagine a *file access* policy φ_{FA} saying "never read a file if it is not open". Intuitively, the gate automaton of Figure 3 represents φ_{FA}.

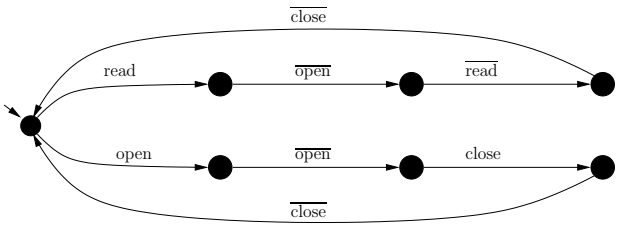


Figure 3: A gate automaton for file access.

Indeed, whenever an action read is going to be performed, *i.e.*, it is taken as input of the gate automata, the automaton inserts an action open before (and a close after) it, *i.e.*, it insert an open and a close as output actions. Instead, if an action open is performed, the automaton propagates it and moves to a state having no transitions labelled with read. This means that every read is left unchanged. The automaton leaves this state if it receives a close action. □

Example 2. Imagine a *Chinese Wall* policy φ_{CW} saying "never send network messages while accessing the file system".

We implement this policy though the gate automaton of Figure 4. The target can either open a file or send data. However, if the program tries to do both the operations, the second one is cancelled, the application's trust level is decreased ($\blacktriangledown$) and the automaton reaches one of the looping states, *i.e.*, *pit* states. The pit allows all the actions but open (send, respectively). □

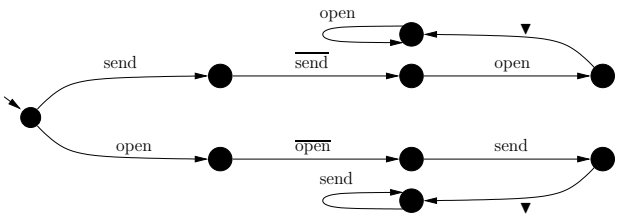


Figure 4: A gate automaton for the *Chinese Wall* policy.

Example 3. The security of mobile devices is based on software certification. Basically, an application is signed with a certificate provided by some trusted entity, *i.e.*, a certification authority. At install time, the signature is verified and, if it is valid, the application receives all the required access privileges. If the signature is corrupted or absent, the application has no access rights to security critical operations and every decision is delegated, time by time, to the user. Users allow or deny permissions to each single operation. If the user consider a program to be harmless, *i.e.*, she trusts the application, she can decide to always permit the action. Symmetrically, if the application is not trusted, the user can decide to never allow the access.

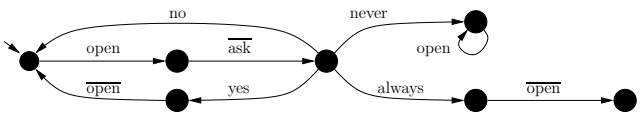


Figure 5: A gate automaton for the *ask user* policy.

The gate automaton of Figure 5 implements the policy that asks the user to decide whether to permit once (yes), always permit (always), deny once (no) or never permit (never) the open action. □

B. From gate automata to interface automata

In this section we introduce the relation between gate automata and interface automata. Basically, a gate automaton can be instantiated to a corresponding interface automaton through a simple transformation. Hence, we use interface automata for giving an operational semantics to the security policies defined through our gate automata.

Definition 2. An instantiation of a gate automaton $\mathcal{G}$ over a index k, denoted by $\mathbf{G}_k$, is an interface automaton $P = \langle V_P \cup V_{id}, \{\imath\}, A_k^{\mathcal{I}}, A_{k+1}^{\mathcal{O}}, \{\blacktriangle, \blacktriangledown\}, T_P\rangle$ where:

- $V_P = V \cup V_{id}$ is the finite set of states (where $V_{id} = \{v_{id}^{\alpha} : v \in V \ \wedge \ v \stackrel{\alpha}{\not\rightarrow} \ \wedge \forall \beta \in \bar{A} \cup \{\blacktriangle, \blacktriangledown\}. v \stackrel{\beta}{\not\rightarrow}\}$)
- $A_k^{\mathcal{I}} = \{\langle \alpha, k\rangle : \alpha \in A\}$ is the input alphabet;
- $A_{k+1}^{\mathcal{O}} = \{\langle \alpha, k+1\rangle : \alpha \in A\}$ is the output alphabet;
- T_P is a set of transitions defined as:

$$\begin{aligned} T_P \ = \ & \{(v, \langle \alpha, k\rangle, w) : (v, \alpha, w) \in T\} \\ \cup \ & \{(v, \langle \alpha, k+1\rangle, w) : (v, \bar{\alpha}, w) \in T\} \\ \cup \ & \{(v, \blacklozenge, w) : (v, \blacklozenge, w) \in T\} \\ \cup \ & \{(v, \langle \alpha, k\rangle, v_{id}^{\alpha}) : v_{id}^{\alpha} \in V_{id}\} \\ \cup \ & \{(v_{id}^{\alpha}, \langle \alpha, k+1\rangle, v) : v_{id}^{\alpha} \in V_{id}\} \end{aligned}$$

  where $\blacklozenge \in \{\blacktriangle, \blacktriangledown\}$

Example 4. Consider the gate automaton of Example 1. We instantiate it with index k and we obtain the interface automaton of Figure 6.

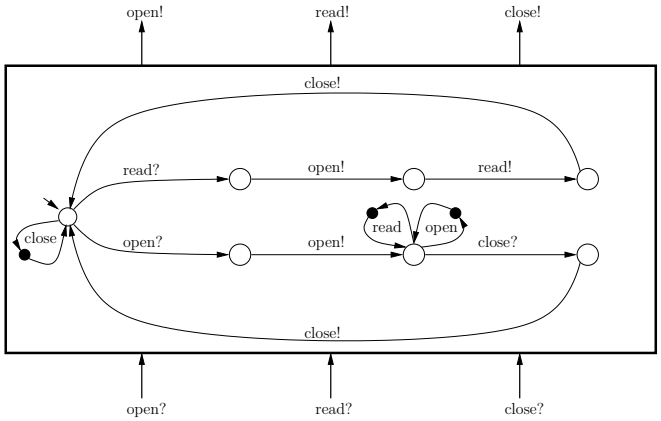


Figure 6: The instantiation of the gate automaton of Fig. 3.

For the sake of simplicity, we use $\alpha?$ and $\alpha!$ in place of α_k and α_{k+1}. Self loops labelled with an action α are a compact notation for the couple of transitions $(v, \alpha?, v_{id}^{\alpha})$ and $(v_{id}^{\alpha}, \alpha!, v)$, where v_{id}^{α} is the small black state in the loop. In the following, we apply this notation in order to have a more readable representation of the automata. □

For what concerns the semantics of an instantiation $\mathbf{G}_k$ of a gate automaton $\mathcal{G}$, we define it in terms of *reaction sequences*. Intuitively, a reaction sequence is a trace of output and internal actions fired by an interface automaton after reading one input symbol. We start by extending the definition of *execution fragment* in the following way.

Definition 3. An *execution fragment* of an interface automaton P is a possibly infinite, alternating sequence of states and actions $v_0, \alpha_0, v_1, \alpha_1, \ldots$ such that $(v_i, \alpha_i, v_{i+1}) \in T_P$.

Definition 4. Given an interface automaton $P = \langle V_P, V_P^{init}, A_P^{\mathcal{I}}, A_P^{\mathcal{O}}, A_P^{\mathcal{H}}, T_P\rangle$, an action $\alpha \in A_P^{\mathcal{I}}$ and a state $v \in V_P$, a *reaction sequence* to α in v is a possibly infinite trace of actions $\sigma = \alpha_0, \alpha_1, \ldots$ such that

- $\alpha_i \in A_P^{\mathcal{O}} \cup A_P^{\mathcal{H}}$,
- $\exists v, v_0, v_1, \ldots \in V_P$ such that $v, \alpha, v_0, \alpha_0, v_1, \alpha_1, \ldots$ is an execution fragment of P and
- if σ has finite length n then $\forall \beta \in A_P^{\mathcal{O}} \cup A_P^{\mathcal{H}} \ . \ v_n \stackrel{\beta}{\not\rightarrow}$.

We say that α is an *activator* of σ in v and denote in with $v \underset{\alpha}{\overset{\sigma}{\Longrightarrow}} v_n$ if σ is finite or $v \underset{\alpha}{\overset{\sigma}{\Longrightarrow}} \uparrow$ otherwise.

C. Trace validity

In this section we provide a formal definition of compliance of a trace with respect to a gate automaton. Intuitively, we can imagine that a sequence of actions is allowed by a gate automaton if, passing it as the input of the (instantiation of the) automaton, the output is the unchanged sequence. We formally define this notion in terms of reactions sequences in the following way.

Definition 5. Given a finite trace of actions $\sigma = \alpha_1, \ldots, \alpha_n$ and a gate automaton $\mathcal{G} = \langle V, \imath, A, T \rangle$ we say that σ is *weakly compliant* with $\mathcal{G}$, in symbols $\sigma \vdash \mathcal{G}$, if and only if for any instantiation $\mathbf{G}_k$ of $\mathcal{G}$ we have

$$\imath \underset{\langle \alpha_1, k \rangle}{\overset{\sigma_1^{k+1}}{\Longrightarrow}} v_1 \ldots \underset{\langle \alpha_n, k \rangle}{\overset{\sigma_n^{k+1}}{\Longrightarrow}} v_n$$

such that $\sigma_i^{k+1} = \langle \beta_{i,1}, k+1 \rangle \ldots \langle \beta_{i,m_i}, k+1 \rangle$ and

$$f_{out}(\sigma_1^{k+1} \ldots \sigma_n^{k+1}) = \sigma$$

where f_{out} is the function recursively defined as

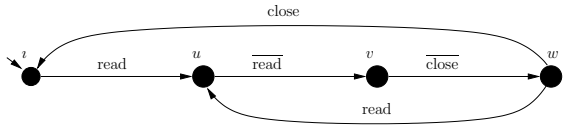
$$f_{out}(\sigma\sigma') = f_{out}(\sigma) f_{out}(\sigma')$$

$$f_{out}(\langle \alpha, h \rangle) = \alpha \qquad f_{out}(\langle \blacklozenge, h \rangle) = \cdot$$

being $\cdot$ the empty trace and $\blacklozenge \in \{\blacktriangle, \blacktriangledown\}$.

Beyond the technical definition, the weak compliance of a trace with respect to a gate automaton is quite intuitive. In particular, we can see the weak compliance as the dual of *transparency*. That is, a trace weakly complies with a gate automaton if and only if an external observer cannot understand whether the trace has been processed by (the instantiation of) the automaton or not.

Example 5. Imagine a policy φ_{RC} saying "all files must be closed after reading". We represent this policy through the 4-states gate automaton $\mathcal{G}$ depicted below.



Basically, the $\mathcal{G}$ allows a read action and immediately enqueues a close reaching the rightmost state w. From this state two branches are possible. If the next action is a close, the automaton cancels it and returns to the initial state $\imath$. Instead, if it receives a read action, it loops on the second state u and repeats the first behaviour.

Consider now the trace $\sigma = \mathrm{read}, \mathrm{close}$. It is easy to verify that interface automaton obtained instantiating the gate automaton for φ_{RC} reacts to σ in the following way:

$$\imath \underset{\mathrm{read?}}{\overset{\mathrm{read!,close!}}{\Longrightarrow}} w \underset{\mathrm{close?}}{\overset{\cdot}{\Longrightarrow}} \imath$$

where we used the notation of Example 4 for input and output actions.

Since $f_{out}(\mathrm{read!}, \mathrm{close!}) = \sigma$, the trace is weakly compliant with $\mathcal{G}$, i.e. $\sigma \vdash \mathcal{G}$.

Clearly, weak compliance does not correspond to a full transparency. Indeed, the transitions of the automaton can introduce and delete actions in such a way that a trace is kept unchanged as a whole, but its prefixes are modified. For instance this can happen when the automaton anticipates an action, *e.g.*, close in the previous example, or postpones it.

For characterising sequences that are not modified at all by a gate automaton we use the notion of *strong compliance*.

Definition 6. Given a finite trace of actions $\sigma = \alpha_1, \ldots, \alpha_n$ and a gate automaton $\mathcal{G} = \langle V, \imath, A, T \rangle$ we say that σ is *strongly compliant* with $\mathcal{G}$, in symbols $\sigma \models \mathcal{G}$, if and only if for any prefix σ' of σ holds that $\sigma' \vdash \mathcal{G}$.

Example 6. Consider again the gate automaton $\mathcal{G}$ of Example 5. We already discussed the weak compliance of the trace $\sigma = \mathrm{read}, \mathrm{close}$ with respect to $\mathcal{G}$. However, we also observed that

$$\imath \underset{\mathrm{read?}}{\overset{\mathrm{read!,close!}}{\Longrightarrow}} w$$

and $f_{out}(\mathrm{read!}, \mathrm{close!}) \neq \mathrm{read}$. Hence, σ is not strongly compliant with $\mathcal{G}$.

D. Comparing gate automata with Edit automata

Ligatti *et al.* [13], extending the definition of security automaton given by Schneider in [17], have defined a new category of deterministic security automata, namely *edit automata*.

An edit automaton is defined as $(\mathcal{Q}, q_0, \delta, \gamma, \omega)$, where $\delta : A \times \mathcal{Q} \to \mathcal{Q}$ is the transition function, $\gamma : A \times \mathcal{Q} \to A \times \mathcal{Q}$ specifies the insertion of an action into the program actions sequence and $\omega : A \times \mathcal{Q} \to \{-, +\}$ indicates whether or not the action in question must be suppressed (-) or emitted (+). The functions ω and δ have the same domain, while the domains of γ and δ are disjoint. Note that this conditions guarantee the resulting automaton to be deterministic, as stated by the following rules.

if $\sigma = a; \sigma'$ and $\delta(a, q) = q'$ and $\omega(a, q) = +$

$$(\sigma, q) \xrightarrow{a}_E (\sigma', q') \qquad \text{(E-StepA)}$$

if $\sigma = a; \sigma'$ and $\delta(a, q) = q'$ and $\omega(a, q) = -$

$$(\sigma, q) \xrightarrow{\tau}_E (\sigma', q') \qquad \text{(E-StepS)}$$

if $\sigma = a; \sigma'$ and $\gamma(a, q) = (b, q')$

$$(\sigma, q) \xrightarrow{b}_E (\sigma, q') \qquad \text{(E-Ins)}$$

otherwise

$$(\sigma, q) \xrightarrow{\cdot}_E (\cdot, q) \qquad \text{(E-Stop)}$$

Also note that the single-step rules can be generalised to sequences of actions by computing the transitive closure of the above transitions.

We can observe that the class of security properties defined through gate automata can be mapped into edit policies.

Proposition 1. For each gate automaton $\mathcal{G}$ there exists an edit automaton $E_{\mathcal{G}}$ enforcing the same property of $\mathcal{G}$.

Proof. (Sketch) Starting from a gate automaton $\mathcal{G}$ we can build a corresponding edit automaton $E_{\mathcal{G}}$. Then we observe that for each input trace σ the two automata produce two outputs that share every finite prefix[1]. □

IV. Runtime enforcement

In this section we present the enforcement environment based on our gate automata. Basically, we describe how gate automata drive the enforcement mechanism in a security framework based on S×C×T. This approach represents an extension of the original S×C×T model [9] in which the policy enforcement is limited to target truncation. Moreover, we improve S×C×T, where the trust management is triggered only through contract violations, by integrating trust-oriented actions in the policies specification.

We saw in Section III that a gate automata are instantiated to corresponding interface automata. We use interface automata to drive the enforcement process of a target. For our purposes, a target R can be seen as a generic agent that fires security-relevant actions as side effects of its execution. Often, we refer to the target as a running program. However, a similar reasoning also applies to other scenarios, *e.g.*, users. Moreover, we assume the enforcement environment to be effective, that is R can be suspended (or asked to wait) before the actual execution of the operation corresponding to the guarded action. For instance, R tries to access a resource, so generating a corresponding action, but it actually obtains the permission only after checking its privileges.

Figure 7 shows the schematic representation of an enforcement environment using gate automata. The first component of the enforcement environment is the *trust management system* (TMS). This component handles the trust weights associated to each agent and provides an implementation of the two internal actions ▲ and ▼. While following the execution of its target, the enforcement environment can perform one or more actions of type ▲ and ▼. The TMS receives these signals and increases (decreases) the target trust level. Note that some trust management systems use a finer characterisation of rewards and penalties, *i.e.*, more than two actions. Nevertheless, this behaviour is fully compatible with our model. Indeed, we can extend the set of internal actions or simulate it by adding new consecutive transitions.

[1]The complete technical proof of Proposition 1 can be found in [16].

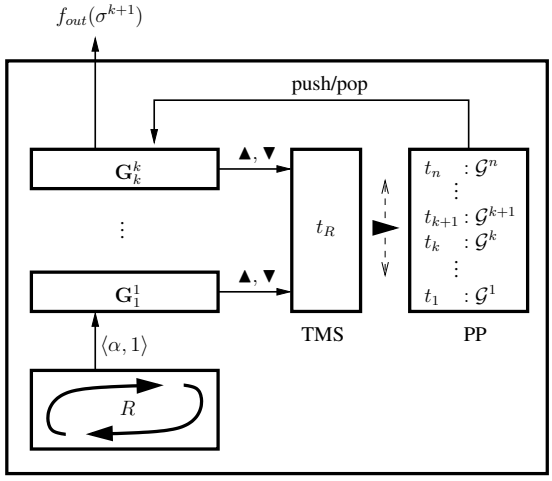


Figure 7: The enforcement environment based on gate automata.

The enforcement environment also contains a set of gate automata $\mathcal{G}^1, \ldots, \mathcal{G}^n$ composing the *policy pool* (PP). The automata in the policy pool are associated to a certain level of trust $0 \leqslant t \leqslant 1$ on which they are inversely ordered, *i.e.*, $1 \leqslant i < j \leqslant n$ implies that $t_i > t_j$. When a target R, having trust level t_R, interacts with the system the policy pool instantiates all the gate automata $\mathcal{G}^i$ such that $t_i \geqslant t_R$ to a corresponding interface automaton $\mathbf{G}_i^i$. Then, the resulting interface automata are composed to create a *interface automata stack* that is applied to R.

The stack receives the actions performed by R and processes them by passing the reaction sequences of each automaton to the layer above. More in detail, assuming that the current state of each interface automaton $\mathbf{G}_i^i$ is v_i, every layer of the stack follows this procedure:

1) receives a trace σ^i from the level below;
2) for each element $\langle \bullet, i \rangle$ of σ^i executes the following sub steps:
   a) if $\bullet = ▲$ (▼) then requires the TMS to increase (decrease) t_R.
   b) otherwise, if $\bullet = \alpha$ computes $v_i \overset{\sigma^{i+1}}{\underset{\langle \alpha, i \rangle}{\Longrightarrow}} v_i'$ and passes the control to the layer above (by invoking this procedure);
3) returns the control to the level below.

When R fires some action α, the previous steps are executed starting from the first layer, where $\sigma^1 = \langle \alpha, 1 \rangle$. The output of the last layer is a sequence or reactions traces that have been stimulated by the action α. Finally, the system evaluates the trust feedbacks contained in these traces, *i.e.*, by passing them to the TMS, and emits the other actions (after removing the index k).

As the actions pass through the stack, the TMS can receive trust adjustment signals. As a consequence, the system can

decide to add or remove one or more automata from the stack. The following example can clarify this behaviour.

Example 7. Consider a policy pool containing $\{0.5 : \mathcal{G}^1, 0.2 : \mathcal{G}^2\}$ where $\mathcal{G}^1$ and $\mathcal{G}^2$ are the gate automata of Example 2 and 3, respectively. According to our model, a program R such that $t_R = 0.3$ is monitored using a stack containing only $\mathbf{G}_1^1$, *i.e.*, the instantiation of $\mathcal{G}^1$. In other words, R is in the scope of a Chinese Wall policy. Imagine now that R tries to fire the actions trace $\mathrm{open}; \mathrm{send}; \mathrm{open}$. At the first step, the stack, *i.e.*, only $\mathbf{G}_1^1$, receives the action $\langle \mathrm{open}, 1 \rangle$ and returns $f_{out}(\langle \mathrm{open}, 2 \rangle) = \mathrm{open}$. When processing send the automaton reacts by suppressing it, decreasing the trust level of R, *e.g.*, $t_R = 0.1$, and moving to the pit state (see Figure 4). Then, as a consequence of the trust level reduction, the system instantiates $\mathcal{G}^2$ and pushes it on the stack. Since there are no actions that the new interface automaton can evaluate, the execution of R continues. Now R tries to perform the second open. Having $\mathcal{G}^1$ no transitions for this action in the current state, the output of $\mathbf{G}_1^1$ is $\sigma^2 = \langle \mathrm{open}, 2 \rangle$. Finally, $\mathbf{G}_2^2$ suppresses the action and generates the final output ask. □

V. Related Work

In [17], security automata have been introduced for expressing security requirements and for enforcing them on a target execution. A security automaton processes possible infinite execution traces. It enforces a policy by stopping the target execution whenever it attempts to violate the corresponding policy, *i.e.*, whenever the target is going to perform an action that is not allowed by the automaton. The enforcement strategy of gate automata not only halts the execution of the target if something goes wrong but is also able to add and suppress actions for correcting the target behaviour when possible. For this reason, as we have underlined in Section III-D, their behaviour is comparable to edit automata [13]. Gate automata differ from edit automata mainly because they manage trust. Moreover, the trust management process is integrated in the enforcement strategy as it automatically drives the monitoring environment.

Also [2] advocates an automaton-based specification of security policies, namely *usage automata*. Usage automata slightly differ from security automata. Roughly, an execution trace complies with a usage policy if and only if it is not accepted as an input word by the corresponding automaton. Moreover, usage policies are applied directly to the source code through proper syntactical operators that also allows for composing the policies in a natural way via scope nesting. The main differences with respect to our automata are that usage automata do not change their target behaviour and they do not consider trust at all. Furthermore, in the environment using gate automata the scope of a policy is not predetermined but is activated/deactivated according to the trust values.

In [14] the authors present how security automata can be modelled through process algebra operators. This allows compositionality and permits to application of existing results on process algebras to the analysis, verification and synthesis of secure systems. In general, process algebras are more expressive than gate automata. However, we propose gate automata as a novel strategy for the enforcement mechanism of the S×C×Tparadigm. Indeed, in literature, several work deal with automata-based specifications of enforcement strategy, *e.g.*, [17], [13]. Gate automata allow to represent and enforce a wide class of properties of interest. Moreover, in addition to security automata, they are directly composable since they are a particular class of interface automata [1].

Referring to interface automata [1], at the best of our knowledge, there are no proposals for applying them to the specification of security or trust properties. Instead, our approach allows for managing in an unique way both security and trust.

The integration between trust management and security enforcement is still an open issue. In [11] the authors propose an access control system that enhances the Globus toolkit with a number of features. In particular, their proposal deals with the integration of access control policies and access rights management in a Grid architecture. Along this line of research, Colombo et al. [4] present an integrated architecture, extending the previous one, with an inference engine that manages reputation and trust credentials. This framework is also extended in [12] a mechanism for trust negotiating credential is introduced to overcome scalability problems. In this way the framework preserves privacy credentials and security policy of both users and providers. Even if these works are deeply biased to the Grid architecture, the basic idea consists of considering the trust as a metrics for deciding the reliability of a resources provider. Nevertheless, these proposals rely on a monitoring environment that simply halts the execution when something wrong happens. In this sense, our model allows for defining edit properties instead of just truncation ones.

VI. Conclusion and Future Work

In this paper we introduced *gate automata* for specifying security and trust policies. Then, we presented a monitoring framework that implements the trust-driven enforcement of security policies in a S×C×T fashion. In this way, we have defined an integrated framework for defining and applying edit policies also dealing with the trust management issues in a intuitive way. During the monitoring process, the policy enforcement provides a feedback to the trust management system while the trust level modifies the enforcement settings by adding/removing security restrictions.

Several directions can be followed as future work. Firstly, we aim at investigating the expressive power of gate automata. In particular, we would like to point out which is

the class of security properties that can be expressed through them.

Another possible research stream is the study of different compositional strategies for the gate automata. Indeed, in this paper we showed how to compose them *vertically* in order to obtain a stack that defines the enforcement environment. However, as their semantics is defined through interface automata, gate automata can be also composed in a *horizontal* way. Horizontal composition, consists of pairing and composing the layers of two different automata stack. In this way we can model the composition of the security policies defined by different entities, *e.g.*, two or more parties involved in a unique computation.

REFERENCES

[1] Luca de Alfaro and T.A. Henzinger. Interface automata. In ACM, editor, *ESEC/FSE*, 2001.

[2] Massimo Bartoletti. Usage automata. In *Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security*, volume 5511 of *Lecture Notes in Computer Science*, pages 52–69, March 2009.

[3] Alessandro Castrucci, Fabio Martinelli, Paolo Mori, and Francesco Roperti. Enhancing Java ME security support with resource usage monitoring. In *ICICS*, pages 256–266, 2008.

[4] Maurizio Colombo, Fabio Martinelli, Paolo Mori, Marinella Petrocchi, and Anna Vaccarelli. Fine grained access control with trust and reputation management for globus. In *OTM Conferences (2)*, pages 1505–1515, 2007.

[5] Gabriele Costa, Fabio Martinelli, Paolo Mori, Christian Schaefer, and Thomas Walter. Runtime monitoring for next generation Java ME platform. *Computers & Security*, July 2009.

[6] N. Dragoni, F. Martinelli, F. Massacci, P. Mori, C. Schaefer, T. Walter, and E. Vetillard. Security-by-Contract (SxC) for software and services of mobile systems. In *At your service - Service-Oriented Computing from an EU Perspective.* MIT Press, 2008.

[7] N. Dragoni, F. Massacci, T. Walter, and C. Schaefer. What the heck is this application doing? - a Security-by-Contract architecture for pervasive services. To appear in *Computers & Security*, Elsevier.

[8] A. Lazouski F. Martinelli F. Massacci G. Costa, N. Dragoni and I. Matteucci. Extending Security-by-Contract with quantitative trust on mobile devices. In *Proceeding of CISIS 2010, The Fourth International Conference on Complex, Intelligent and Software Intensive Systems*, pages 872–877, 2010.

[9] V. Issarny A. Lazouski F. Martinelli F. Massacci I. Matteucci G. Costa, N. Dragoni and R. Saadi. Security-by-Contract-with-Trust for mobile devices. *Journal of Wireless Mobile Networks, Ubiquitous Computing and Dependable Applications*, 2010. To appear.

[10] Li Gong. Java Security: Present and Near Future. *IEEE Micro*, 17(3):14–19, 1997.

[11] H. Koshutanski, F. Martinelli, P. Mori, L. Borz, and A. Vaccarelli. A fine grained and x.509 based access control system for globus. In *OTM*, pages 1336–1350. Springer, 2006.

[12] Hristo Koshutanski, Aliaksandr Lazouski, Fabio Martinelli, and Paolo Mori. Enhancing grid security by fine-grained behavioral control and negotiation-based authorization. *Int. J. Inf. Sec.*, 8(4):291–314, 2009.

[13] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2), February 2005.

[14] Fabio Martinelli and Ilaria Matteucci. Through modeling to synthesis of security automata. *Electr. Notes Theor. Comput. Sci.*, 179:31–46, 2007.

[15] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, January 1997.

[16] Gabriele Costa's Web Page. www.di.unipi.it/∼costa, 2010.

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

[18] R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 15–28, 2003.