Towards Practical Enforcement Theories*

Nataliia Bielova[1], Fabio Massacci[1], and Andrea Micheletti[2]

[1] Università degli Studi di Trento, Italy, lastname@disi.unitn.it
[2] Fondazione Centro San Raffaele del Monte Tabor, e-Services for Life & Health Unit, Milano, Italy, micheletti.andrea@hsr.it

Abstract. Runtime enforcement is a common mechanism for ensuring that program executions adhere to constraints specified by a security policy. It is based on two simple ideas: the enforcement mechanism should leave good executions without changes and make sure that the bad ones got amended. From the theory side, a number of papers [6, 10, 12] provide the precise characterization of good executions that can be captured by a security policy and thus enforced by a specific mechanism. Unfortunately, those theories do not distinguish what happens when an execution is actually bad (the practical case). The theory only says that the outcome of enforcement mechanism should be "good" but not how far should the bad execution be changed.
If we consider a real-life example of a drug dispensation process in a hospital the notion of security automata or even edit automata would stop all requests by all doctors on all drugs and all dispensation protocols, as soon as a doctor forgot to insert the research protocol number.
In this paper we explore a set of policies called iterative properties that revises the notion of good traces in terms of repeated iterations. We start discussing how an enforcement mechanism can actually deal with bad executions (and not just only the good ones).

1 Introduction

The last few years have seen a renewed interest in the theoretical and practical aspects of the run-time security enforcement mechanisms. After Schneider's paper [11] on security automata, a number of refinements have been proposed. For example Hamlen's work on rewriting [6], and Ligatti et al. works on edit automata [2, 12]. Most papers had an applied counterpart: systems actually capable of enforcing the security policies [4, 7, 10].

Yet, as we have shown already in [3], there is a gap between the theoretical constructions actually used in the papers and the practical implementation. The reasons behind this gap is the actual format of the definitions that have been so far used to formally describe the "good" properties of an enforcement mechanism: transparency and soundness.

Basically, soundness says that every output of enforcement mechanism should be valid and transparency says that in case of valid input, the output should be

* Research partly supported by the Project EU-FP7-IP-MASTER.

equal to the input. Most papers focused on kinds of good traces that be potentially enforced with particular enforcement mechanism, thus providing an initial classification. For practical applications, this is not enough. What distinguishes an enforcement mechanism is not what happens when traces are good, because nothing should happen! The interesting part is how precisely bad traces are converted into good ones. To this extent soundness only says they should be made good. The practical systems, being practical, will actually take care of correcting the bad traces. But this part is simply not reflected in the current theories. Not even Ligatti's own running example could be generated or accounted for by his theoretical constructions [3].

In order to be concrete and show the impressive width of the gap, we use a real, industrial level healthcare process set-up in some private and public hospitals for drug dispensation.

The classical enforcement mechanisms assume that as soon as some restricted operation happens, either the process must be immediately stopped [11] or the mechanism should wait until the execution becomes valid again by itself [10]. In practice a mechanism faithfully implementing security automata á la Schneider would stop the whole drug dispensation process because once a doctor did not insert a research protocol number. Following the theoretical construction actually used by Ligatti, Bauer and Walker the mechanism should wait, suppressing actions of doctors and nurses, until the doctor would insert that missing protocol number. Process becomes valid again by itself. This behavior is not inherent to the theoretical constructions that are possible with edit automata. It is just the only one that is actually provided in the cited papers. Therefore, in this paper we address the following challenge:

Challenge 1 *The enforcement mechanism should have a "plausible"/"believable" behavior when the requests (of the users) do not correspond to the policy.*

Contribution of the paper. In the next section, we describe our concrete running example: a health-care process of drug dispensation to outpatients. It will be used to show what kind of practical enforcement can be done whenever process execution violates the security policy. In §3 the basic notations (for edit automata, enforcement, etc.) that will be used in the paper are presented.

Our contributions are following: we introduce a new kind of property called iterative property. Loosely speaking, it captures the practical intuition of repeating good transactions. This property corresponds more accurately than renewal and safety property to the actual traces that are used in practice. In our practical example it covers all properties of interests with the exception of liveness.

Then, we also show the relation between iterative properties to all other classical security properties and show how to represent it with simple policy automata (§4). Finally, we show which mechanisms can concretely enforce this property and how to construct them (§6) and conclude (§7).

As a simple example, we have shown in [3] that the example of Fig.2 in [2] could not be generated from the proof of Thm.8. In this paper, this example can be automatically generated from the specific automata corresponding to the

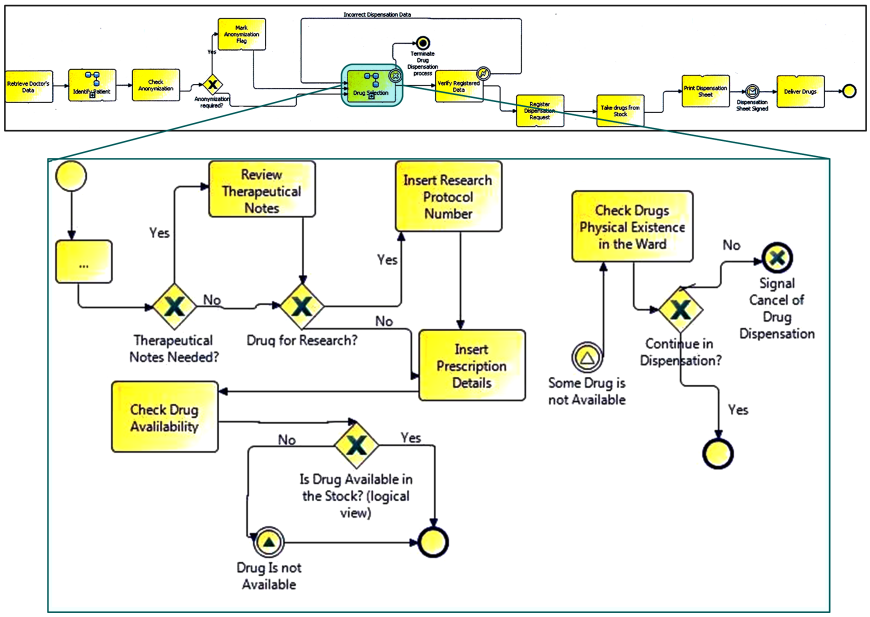


Fig. 1. BPMN model drug dispensation process

policy by our algorithm in §6 if, whenever there is an attempt to violate the policy, we emit a warning instead of silently suppressing the action.

2 Running example

The case study in this paper is based on a healthcare process of drug dispensation. Private Hospitals accredited with the Public National Health Service are charged with administering drugs or with providing diagnostic services to patients that use their structure and then are authorized to claim the cost of drug dispensation or diagnostic provisioning to the Regional Healthcare Authority.

In particular, there is a mechanism called File F that allows hospitals to refund the drugs administered and/or supplied in the hospitals' outpatient departments to the patients that are not hospitalized. Here we describe the drug dispensation process consisting of the following main steps: the doctor identifies the patient; the doctor selects the drugs, registers the dispensation, takes the drugs from the stock and prints the dispensation sheet; doctor signs the dispensation sheet and delivers drugs to the patient.

In Fig. 1 we present a BPMN diagram of drug dispensation process emphasizing the drug selection subprocess[3]. According to this subprocess, the doctor retrieves patient's prescription and dispensation historical data, selects the drugs

[3] We would like to thank ANECT for developing original BPMN diagrams of the full drug dispensation process.

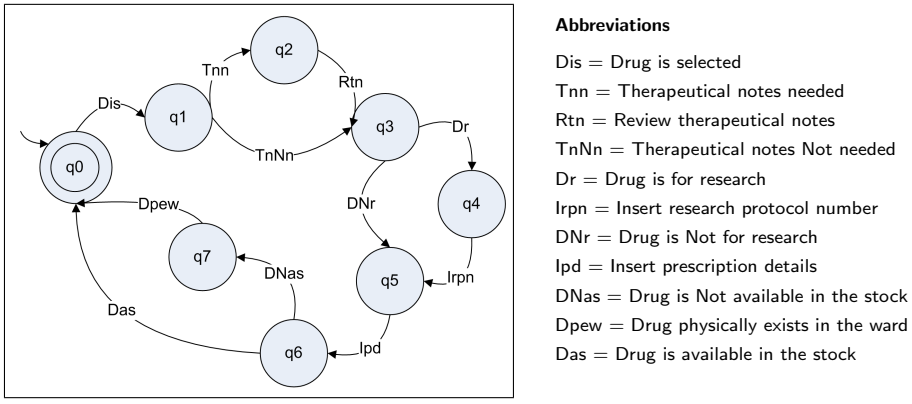


Fig. 2. Policy automaton for iterative drug selection subprocess

by existing prescription, by repeating dispensation or from the stock. Then, for each selected drug the following procedure holds:

1. First the doctor should select the drug.
2. If the therapeutical notes needed, doctor reviews them in this step.
3. If patient is using prescribed drug for the research program purposes (i.e., the patient has been enrolled in the clinical trial for the testing of that drug), doctor has to insert the research protocol number into the prescription form.
4. Doctor has to insert all the other prescription details.
5. The availability of the drug in the stock is checked. If it is not available the next check is made, otherwise the process succeeds.
6. Doctor checks the drug availability in the ward. If it is available the process succeeds, otherwise fails.

In this paper we want to define how the executions where "something locally bad may happen" can be enforced. Let us come back to the Fig. 1 and assume that each choice in BPMN diagram corresponds to the action in resulting process execution that communicates the choice, e.g., if the drug is for research then the corresponding event is "Drug is for research", similarly, if drug is not for research then the event is "Drug is not for research". To ease the comparison with other papers on enforcement [2, 12] we give in Fig. 2 automaton corresponding to the BPMN process. Formal definition will be given later in the section 3. Intuitively an acceptable iteration requires following the edges of the automaton and end in the state $q0$ without skipping steps.

Example 1. Let us assume the following execution of the process, which consists of 3 iterations for three different drugs. At first the drug is selected, therapeutical notes are not needed, the drug is not for research, we insert prescription details, the drug is available at the stock. The first part is correct. The second iteration is: drug is selected, therapeutical notes not needed, the drug is for research, insert prescription details, the drug is available at the stock. In this iteration the drug is for research but the research protocol number is not inserted, therefore it is not

correct. The third iteration is drug is selected, therapeutical notes needed, review therapeutical notes, the drug is not for research, insert prescription details, the drug is available at the stock. This iteration is also correct. ♢

Even if we accept the idea that incorrect execution should be dropped, the acceptable behavior for the administrators of the e-health system is just to drop the second part of the execution.

3 Security Properties

Following the standard notation on run-time security policies [11, 5, 2] we denote the set of observable program actions by $\mathcal{A}$. An *execution*, or a trace, is a finite or infinite sequence of actions; the set of all finite executions over $\mathcal{A}$ is denoted by $\mathcal{A}^*$, the set of infinite executions is $\mathcal{A}^\omega$, and the set of all executions is $\mathcal{A}^\infty$. We write $\tau;\sigma$ to denote concatenation of two sequences and τ must be finite. By $\tau \preceq \sigma$, or $\sigma \succeq \tau$ we denote that τ is a finite prefix of finite sequence σ.

A *security property* is a predicate $\widehat{P}$ over traces or, equivalently, a set of traces $\Sigma \subseteq \mathcal{A}^\infty$ such that $\forall \sigma \in \Sigma.\, \widehat{P}(\sigma)$. Schneider only considered infinite traces (by extending finite traces by the repetition of the last action) but we prefer to distinguish between finite and infinite traces. In the sequel the execution σ that satisfies the property $\widehat{P}$ is called *valid*, *legal* or *good*. Similarly, the execution that does not satisfy the property will be called *invalid*, *illegal* or *bad*.

There are several classes of properties. The property "nothing bad ever happens" is called *safety* property and formally defined by Lamport [8]. Additional to safety properties, there are *liveness* properties [1] that claim that "something good eventually happens during any execution" or in the other words any finite execution can always be extended to satisfy the property.

However, except for safety and liveness properties there are more general properties, which allow executions to alternate between satisfying and violating security property. *Renewal* property presented in [10] is such a property. According to it, every infinite-length sequence has infinite number of valid finite prefixes and every invalid sequence has only a finite number of valid prefixes.

Definition 1. *Property $\widehat{P}$ is* renewal *if the following holds:*

$$\forall \sigma \in \mathcal{A}^\omega :\ \widehat{P}(\sigma) \Longleftrightarrow (\forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \widehat{P}(\tau)) \tag{1}$$

According to [10] every decidable renewal property can be enforced by a Ligatti automata (a particular kind of edit automata [3]). These automata will always output only the longest valid prefix of the input. The renewal property, similar to liveness property, implicitly assumes that "nothing irremediably bad happens in any finite execution". It is obviously implied by the fact that the valid execution must have infinite number of valid prefixes. Therefore if the valid execution has something irremediably bad appeared in a finite prefix, then the number of valid prefixes is finite.

Example 2. Let us expand the Ex. 1. Result of the drug selection process execution for three drugs is a following trace, which consists of three iterations:

1. The first iteration is Dis; TnNn; DNr; Ipd; Das, which is a valid execution.
2. The second attempted iteration is Dis; TnNn; Dr; Ipd; Das, which is an invalid execution. It means that the drug is for research (Dr action) but the research protocol number is not inserted (there is no action Irpn after Dr).
3. The third iteration in the trace consists of Dis; Tnn; Rtn, DNr; Ipd; Das, which is a valid iteration.

The resulting trace is invalid since it has an invalid second part after which the trace can never become valid again. What kind of behavior is expected from the enforcement mechanism in this case? ♢

The Ex. 2 clearly presents a renewal property, since if the infinite sequence is valid (always contains action Irpn after Dr or contains action DNr) it has infinite number of valid prefixes. Moreover, it can be enforced by Ligatti Automaton by outputting the longest valid prefix. In this case the resulting execution will be a first iteration of the process execution.

However, the administrators of the e-health system might expect a resulting trace to be longer. Since the drug selection process for the third drug is valid by itself, they would like to have this iteration in the resulting trace. Therefore an excected behavior of the system is following: to delete the second invalid iteration of the execution, and to output the first iteration followed by a third iteration. However, this trace correction is not provided by existing techniques [10].

The iterative drug selection process used in Ex. 2 shows that there exists a class of properties that accept set of traces consisted of repeated executions. In our example the trace is legal if it consists of iterations that are representing all the paths from state $q0$ to state $q0$ in Fig. 2. We define this class as *iterative properties* (like in [10] assuming that empty trace is always valid).

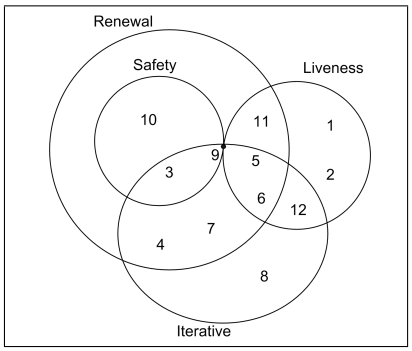
Definition 2. *Property* $\widehat{P}$ *is a* iterative property *iff*

$$\forall \sigma,\, \sigma' \in \mathcal{A}^*.\, \widehat{P}(\sigma) \wedge \widehat{P}(\sigma') \Longrightarrow \widehat{P}(\sigma;\sigma') \tag{2}$$

Notice that given an infinite sequence of the form $(\sigma)^\infty$, it can satisfy some iterative property, which accepts σ. By this example we want to emphasize, that iterative property is defined over finite and infinite sequences.

We present several examples of the iterative properties that are also renewal properties as well as some others that are not renewal and not even liveness. The relationship between safety, liveness and renewal properties is taken from Fig. 1 of [10]. We add their relationships with iterative properties in Fig. 3.

Properties 3-7 are iterative properties since by concatenating two valid sequences a valid sequence is always obtained. The trivial property 9 is iterative as well. The property 8, which was defined as non-renewal and non-liveness in [10], is an iterative property. If we concatenate two valid executions that are terminated and never access private files then the resulting execution will also be



Legend

1 Nontermination
2 Resource availability
3 Stack inspection
4 Log out and never open files
5 Property 4 on system without file-open actions
6 Eventually audits
7 Transaction property
8 Termination + file access control
9 Trivial
10 All sequences with fixed length
11 Increasingly longer sequences
12 Termination

Fig. 3. Relationships between security properties

valid. We similarly explain the property 12: since all finite sequences are valid, the resulting concatenated finite-length sequence is valid as well.

The property 10 is non-iterative. However, it is a safety property – if the sequence is invalid and its length is bigger than some fixed number n, then there exists a prefix of length $n+1$ such that any continuation of this prefix is an invalid sequence. The property 11 states that the sequence is valid iff it is infinite or its length belongs to the following set of numbers $\{F_i\}$: $F_0 = 1$, $F_{i+1} = 2F_i + 1$. This property is not iterative: by concatenating two valid sequences a new invalid sequence is obtained. Yet, this is a renewal property, because every valid infinite-length sequence has an infinite number of valid prefixes.

4 Iterative property representation

For finite representation of the security policies we use a variant of automata without loss of generality. Indeed, it was shown in Proposition 6.24 of [12] that edit automata can only enforce properties represented by Büchi automaton.

Definition 3 (Policy automaton). *A* Policy automaton *is a 5-tuple of the form $\langle \mathcal{A}, Q, q_0, \delta, F \rangle$, where $\mathcal{A}$ is a finite nonempty set of security-relevant program actions, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \mathcal{A} \to Q$ is a labeled partial transition function, and $F \subseteq Q$ is a set of accepting states. The initial state is always an accepting state ($q_0 \in F$) and Policy automaton satisfies the following properties:*

1. *It does not have any dead-ends.*
2. *All the states of automaton are reachable.*
3. *It is a deterministic automaton.*

For the drug selection subprocess from Fig. 1 we constructed the Policy automaton from Fig. 2. In the following we will use this Policy automaton.

The Policy automaton can have accepting and non-accepting states. In our example, if the action Dr ("Drug is for research") happens, then the subsequent

state must be non accepting and only after action Irpn ("Insert research protocol number") the next state may be accepting.

Definition 4 (Run of a Policy automaton). *Let $A = \langle \mathcal{A}, Q, q_0, \delta, F \rangle$ be a policy automaton. A* run *of A on a finite (respectively infinite) sequence of actions $\sigma = \langle a_0, a_1, a_2, \ldots \rangle$ is a sequence of states $q_{|\sigma|} = \langle q_0, q_1, q_2 \ldots \rangle$ such that $q_{i+1} = \delta(q_i, a_i)$. A* finite run is accepting *if the last state of the run is an accepting state. An* infinite run is accepting *if the automaton goes through some accepting states infinitely often.*

Definition 5 (Property represented as Policy automaton). *Some property $\widehat{P}_A$ is represented as a Policy automaton A if and only if:*

$$\forall \sigma \in \mathcal{A}^{\infty} : \widehat{P}_A(\sigma) \Longleftrightarrow A \text{ accepts } \sigma \tag{3}$$

The Policy automaton combines the acceptance conditions of Büchi Automata and finite state automata. Some iterative properties can be represented as a Policy automaton. For example, a predicate $\widehat{P}$ corresponding to drug selection subprocess is an iterative property and it is represented by Policy automaton.

5 Effective vs. Iterative enforcement

Once we have designed the security policy we can define an enforcement mechanism. Runtime enforcers transform the program executions of an untrusted application to ensure that they do not violate security property. For example, the enforcer can be formally modeled as security automata [11] (for safety properties), or as edit automata [10] (for renewal properties).

For the sake of simplicity, we only reason only about finite executions. We assume that enforcement mechanism E given some input execution sequence τ transforms it into the output sequence $E(\tau)$.

Traditionally, an enforcement mechanism should satisfy (at least) two properties: soundness and transparency. Soundness says that all the output of the enforcement mechanism should be legal. An enforcement mechanism obeys transparency if all the good sequences are output without changes. Enforcement mechanism provides "effective$_=$enforcement" [9] if it obeys both of these properties.

Definition 6. *An enforcement mechanism E* effectively$_=$enforces *a property $\widehat{P}$ on the system with action set $\mathcal{A}$ iff*

1. $\forall \sigma \in \mathcal{A}^* \,.\, \widehat{P}(E(\sigma))$
2. $\forall \sigma \in \mathcal{A}^* \,.\, \widehat{P}(\sigma) \Rightarrow E(\sigma) = \sigma$

The definition of effective$_=$enforcement is enough if we deal only with valid input. Formally speaking, a mechanism that transforms every invalid trace to an empty trace will also satisfy the property of "effective$_=$enforcement". However, none of our references from the hospital will consider an "effective enforcement"

mechanism that will stop drug selection subprocess and whole dispensation process because the doctor failed to insert a research protocol number. Therefore, sequences are not wholly good or wholly bad. They are composed by fragments that can be good or bad. The removal of a bad fragment from an otherwise good sequence can make it good.

In Thm. 4 of [3] it is shown that Ligatti automaton is specific type of edit automaton that has an all-or-nothing behavior and delayed precisely enforces given property. All-or-nothing means that it always outputs all the read actions that are not output so far or does not output anything. Delayed precise enforcement means that only the longest valid prefix is output. We give a formal definition of the longest valid prefix and type of enforcement that Ligatti automaton provides.

Definition 7. *The prefix σ_o (o for "output") is a* longest valid prefix *of the sequence of actions σ with respect to the property $\widehat{P}$ iff*

$$\widehat{P}(\sigma_o) \wedge (\forall \sigma^* . \sigma^* \succeq \sigma_o \wedge \sigma^* \preceq \sigma \Longrightarrow \neg \widehat{P}(\sigma^*)) \tag{4}$$

Definition 8. *An enforcement mechanism E* all-or-nothing delayed precisely enforces *a property $\widehat{P}$ on the system with action set $\mathcal{A}$ iff*

1. $\forall \sigma \in \mathcal{A}^* \,.\, \widehat{P}(\sigma) \Rightarrow E(\sigma) = \sigma$
2. $\forall \sigma \in \mathcal{A}^* \,.\, \neg \widehat{P}(\sigma) \Rightarrow E(\sigma) = \sigma_o$, *where σ_o is a longest valid prefix of σ.*

Let us define another type of enforcement, which is able to output longer acceptable sequences than just a longest valid prefix.

Definition 9. *An enforcement mechanism E* iteratively enforces by suppression *an iterative property $\widehat{P}$ on the system with action set $\mathcal{A}$ iff*

1. $\forall \sigma \in \mathcal{A}^* . \widehat{P}(\sigma) \Rightarrow E(\sigma) = \sigma$, *and*
2. $\forall \sigma \in \mathcal{A}^* . \neg \widehat{P}(\sigma) \wedge \exists \sigma^* \succeq \sigma \,.\, \widehat{P}(\sigma^*) \Rightarrow E(\sigma) = \sigma_o$, *where σ_o is the longest valid prefix of σ, and*
3. $\forall \sigma \in \mathcal{A}^* . \neg \widehat{P}(\sigma) \wedge \forall \sigma^* \succeq \sigma \,.\, \neg \widehat{P}(\sigma^*) \wedge \exists \sigma_b . \sigma = \sigma_o ; \sigma_b ; \sigma_r \wedge \sigma_b$ *is the smallest sequence s.t. after deleting it from σ the resulting trace can become good again $\Rightarrow E(\sigma) = \sigma_o ; E(\sigma_r)$, where σ_o is the longest valid prefix of σ.*

The example of market policy (Fig. 2 of [2]) shows exactly iterative enforcement by suppression. The only difference is that the edit automaton from Fig. 2 does not simply suppress the bad sequences but gives a warning to the user. Hence, the Ligatti automaton [3] provides all-or-nothing delayed precise enforcement, while edit automaton from Fig. 2 of [2] provides iterative enforcement by suppression. However, it is not clear how the latter automaton can be automatically constructed for given policy.

6 Iterative enforcement mechanism

We take the definition of edit automata from [3]. Generally speaking, edit automaton can suppress actions without further insertion, but this power of edit

automaton was not used to enforce renewal properties. It happened because the sequences corresponding to renewal property do not contain any bad parts that make them not able to become good again in the future. We present such sequence in Ex. 2, where bad part is a second iteration.

Definition 10. *An* edit automaton E *is a 5-tuple of the form* $\langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ *with respect to some system with actions set* $\mathcal{A}$*. Q specifies possible states, and* $q_0 \in Q$ *is the initial state. The partial function* $\delta : (Q \times \mathcal{A}) \to Q$ *specifies the transition function; the partial function* $\gamma_o : (Q \times \mathcal{A}^* \times \mathcal{A}) \to \mathcal{A}^*$ *defines the output of the transition according to the current state, the sequence of actions that has been read before the current action and the current input action; the partial function* $\gamma_k : (Q \times \mathcal{A}^* \times \mathcal{A}) \to \mathcal{A}^*$ *defined the sequence that will be kept after committing the transition. The dependence between the transition, output and keep function is following: if* $\delta(q, a)$ *is defined then* $\gamma_o(q, \sigma, a)$ *and* $\gamma_k(q, \sigma, a)$ *must be defined for all* σ*.*

Considering σ as a sequence of actions read so far and a as input action, we write an assignment of a transition from state q to state q':

$$\boxed{\langle \delta, \gamma_o, \gamma_k \rangle (q, a) = q' \,|\, \sigma' \,|\, \sigma''}$$

where $\delta(q, a) = q'$, $\gamma_o(q, \sigma, a) = \sigma'$, $\gamma_k(q, \sigma, a) = \sigma''$.

In order for the enforcement mechanism to be effective all functions δ, γ_k and γ_o should be decidable.

Definition 11. *Let* $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ *be an edit automaton. A* run *of automaton A on an* input *sequence of actions* $\sigma = \langle a_1, a_2, \ldots \rangle$ *is a sequence of pairs* $\langle (q_0, \epsilon), (q_1, \sigma_1), (q_2, \sigma_2), \ldots \rangle$ *such that* $q_{i+1} = \delta(q_i, a_{i+1})$ *and* $\sigma_{i+1} = \gamma_k(q_i, a_{i+1}, \sigma_i)$*. The* output *of automaton A on input* σ *is sequence of actions* $\sigma_o = \langle \sigma_1^o, \sigma_2^o, \ldots \rangle$ *such that* $\sigma_{i+1}^o = \gamma_o(q_i, a_{i+1}, \sigma_i)$*.*

In the sequel we use $*$ as an abbreviation for the sequence of actions kept in memory so far (actions that were read but not output yet). Then we use $*; a$ as an abbreviation for the concatenation of the current memory and action a. So $\langle \delta, \gamma_o, \gamma_k \rangle (q, a) = q' \,|\, \epsilon \,|\, *; a$ means that $\delta(q, a) = q'$, $\gamma_o(q, \sigma, a) = \epsilon$ for all $\sigma \in \mathcal{A}^*$ and $\gamma_k(q, \sigma, a) = \sigma; a$ for all $\sigma \in \mathcal{A}^*$. In order to farther simplify the reading of the figures we use boolean expressions on the edges of the automaton. So that if we label an edge with $!(a \vee b)$ it means that it can only take any action different from a or b. We use $\top$ for any action; ADD for adding an input action to the memory: $a \,|\, \epsilon \,|\, \text{ADD}$ is an abbreviation for $a \,|\, \epsilon \,|\, *; a$.

Let us come back to our Ex. 1 and to drug selection process representation in Fig. 2. First we will show how it can be all-or-nothing delayed precisely enforced and iteratively enforced by suppression. Then we compare the input and output of both types of enforcement.

Algorithm 1 Ligatti automaton Algorithm

Input: Policy automaton $A^P = \langle \mathcal{A}, Q^P, q_0^P, \delta^P, F^P \rangle$;
Output: Ligatti automaton $E = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$;
1: $q_0 = q_0^P$; $Q = Q^P \cup \{q_\perp\}$;
2: **for all** $q \in Q^P \cup \{q_\perp\}$ **do**
3: **for all** actions $a \in \mathcal{A}$ **do**
4: **if** $\delta^P(q, a)$ is not defined **then**
5: $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) := q_\perp \ |\epsilon|*; a$;
6: **else**
7: **if** $\delta^P(q, a) \notin F^P$ **then**
8: $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) := \delta^P(q, a) \ |\epsilon|*; a$;
9: **else**
10: $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) := \delta^P(q, a) \ |*; a|\epsilon$;

6.1 Ligatti automaton construction

Ligatti automaton is a specific kind of edit automaton that is constructed according to the proof of the theorem about effective$_=$enforcement (Thm. 8 of [2]). In this section we want to show how to build such automaton, which moreover provides all-or-nothing delayed precisely enforcement.

In Alg. 1 we present our construction. As a result, we obtain an automaton which has the same behavior as an automaton constructed by the proof of Thm. 8 [2]. This proof gives a construction of infinite state automaton that keeps in the state the sequence read so far but not valid yet. Since we represent the policy as a Policy automaton, we can know whether the sequence can ever become good again. It can if transition function is defined for current state and incoming action. Otherwise the trace can never become valid again. Considering this, we show an algorithm of automaton construction that always outputs the longest valid prefix, thus providing all-or-nothing delayed precise enforcement.

Alg. 1 works as follows. Suppose the state of the automaton after executing sequence σ is q, the next incoming action is a. In line 4 we check whether there is a path in the Policy automaton from the state q on action a. If there is no such path ($\delta^P(q, a)$ is not defined) it means that there is no way to reach the accepting state of the policy, therefore the sequence can never become valid again. Hence the output done so far is the only output produced on σ and all its continuations, therefore the next state is an error state: $\delta(q, a) := q_\perp$.

Otherwise (line 6) we can reach the next state. If the next state is non-accepting, it means that possibly there is a path to the accepting state, so the currently read sequence can become good. Therefore we simply keep the current action in the memory (line 8). If the next state is accepting, then currently read input $\sigma; a$ is accepted by the Policy Automaton. Therefore, we output all the actions read so far followed by the current action (line 10). The resulting Ligatti automaton for the Policy presented in Fig. 2 is partially shown in Fig. 4 (for the sake of readability we only show outcoming edges from the states $q_4, q_6, q_\perp$).

The behavior of the Ligatti automaton constructed by this algorithm is exactly the same as of one constructed by the proof of Thm. 8. The only difference

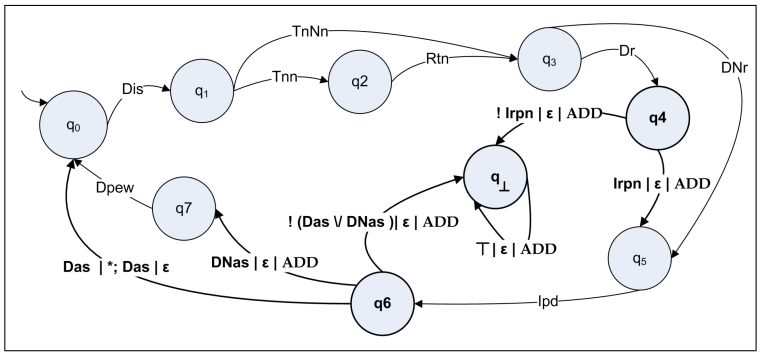


Fig. 4. Resulting Ligatti automaton for the Policy automaton from Fig. 2

is that in Thm. 8 the state contains all the read actions and if the trace can never become good again there will be as many states as the length of the trace. While in our construction, as soon as the trace cannot become good again, the next state will be an error state and all the following input actions will be kept.

Theorem 1. *A property $\widehat{P}$ represented by Policy automaton A^P can be* all-or-nothing delayed precisely enforced

6.2 Iterative enforcement by suppression mechanism

We will use a following assumption for an iterative property $\widehat{P}$ represented by Policy automaton. Let's take all possible sequences of actions corresponding to all the paths form one accepting state to another accepting state such that they don't contain any accepting states in the middle. Then we assume that each of this sequences starts with a unique starting action which never appears in the sequence again. Note that several sequences can have the same unique starting action. Indeed, in the example of drug selection process (Fig. 1) every valid sequence starts as soon as doctor chooses the drug and he can choose the drug again only in the beginning of next sequence.

We can find a lot of real life examples of this kind of iterative process that start with unique starting action. For instance, remaining in the healthcare world we can consider the case when a nurse has to prepare a therapy for a hospitalized patient: once identified the specific therapy (unique starting action) the nurse starts an iterative process. Another example is the reservation of a medical examination: once the administrative personnel receives a call from a person (unique starting action), he starts a process in order to examine the request, to evaluate availability and to confirm the reservation. All these processes are repeated several times for different therapies, patients, etc.

Since every accepting sequence starts with a unique starting action, we can define the smallest bad sequence σ_b from the definition 9. Suppose, that sequence σ is executing and it corresponds to the 3rd clause of the definition. Then, as soon as the longest valid prefix σ_o executes, it is output without changes. Then, there are two possible options. First is if the next action is a unique starting action,

Algorithm 2 Suppressing bad parts Algorithm

Input: Policy automaton $A^P = \langle \mathcal{A}, Q^P, q_0^P, \delta^P, F^P \rangle$;
Output: Edit automaton $E = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$;
1: $q_0 = q_0^P$; $Q = Q^P \cup \{q_\perp\}$;
2: **for all** $q \in Q^P \cup \{q_\perp\}$ **do**
3: **for all** actions $a \in \mathcal{A}$ **do**
4: **if** $\delta^P(q, a)$ is not defined **then**
5: **if** $\exists q^A \in F^P . \delta^P(q^A, a) = q'$ **then**
6: **if** $q' \in F^P$ **then**
7: $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) := q' \ \ |a|\epsilon$;
8: **else**
9: $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) := q' \ \ |\epsilon|a$;
10: **else**
11: $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) := q_\perp \ \ |\epsilon|\epsilon$;
12: **else**
13: **if** $\delta^P(q, a) \notin F^P$ **then**
14: $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) := \delta^P(q, a) \ \ |\epsilon|*; a$;
15: **else**
16: $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) := \delta^P(q, a) \ \ |*; a|\epsilon$;

then σ_b starts with this action; the enforcement mechanism keeps in σ_b all the actions coming after it until the new unique starting action appears in the input. At this point the new accepting sequence can start. Therefore, by deleting σ_b we make the smallest possible suppression. Second option is that the next action is not a unique starting action, then the enforcement mechanism keeps in σ_b all the actions coming after last valid input. It happens until the new unique starting action appears in the input, at this point the new accepting sequence can start. Therefore, by deleting σ_b we make the smallest possible suppression.

In this section we show how to construct an edit automaton that provides iterative enforcement by suppression of iterative property represented by Policy automaton. We propose to construct an edit automaton $E = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ for iterative property $\widehat{P}$ represented by a Policy automaton $A^P = \langle \mathcal{A}, Q^P, q_0^P, \delta^P, F^P \rangle$ as shown in the Alg. 2. This algorithm is obtained from the Alg. 1 by changing the line 5 of the former algorithm to the lines 5-11 of the latter one. These changes correspond to the case when the trace cannot become good again since there are no transitions in the Policy automaton for the next incoming action.

Condition on line 5 corresponds to the case when the next incoming action a is a unique starting action. Then, if only this action is accepted by the Policy automaton, which means that $\widehat{P}(a)$ (line 6), this action is immediately output and memory is empty; next state is an accepting state corresponding to acceptance of this action. If unique starting action a is not accepted, then the memory is updated with the only this action and output empty sequence (line 9). If a is not a unique starting action (line 10), there is a transition to an error state and incoming action a is not kept in the memory. Notice, that an assumption about

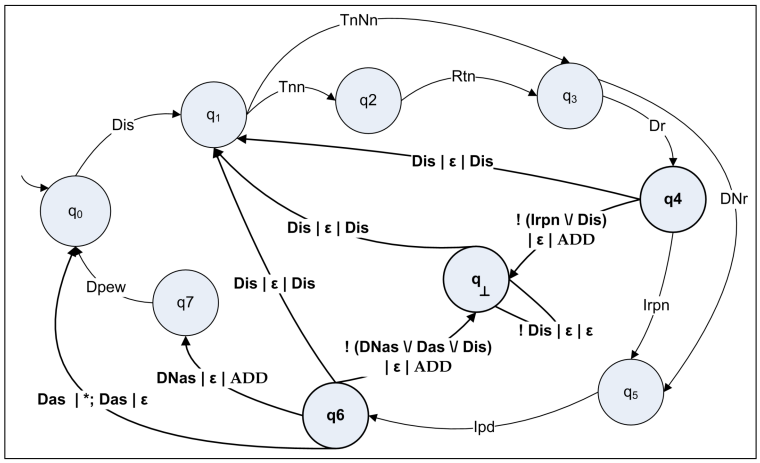


Fig. 5. Resulting edit automaton for the Policy automaton from Fig. 2

unique starting action is critical here, without this assumption in general case the algorithm will not satisfy the 3rd clause of the definition 9.

One of the differences is that the constructed automaton will be able to come back to the Policy automaton state from the error state as if no error happened. By doing so we simply delete smallest bad sequences parts (σ_b in the definition 9, which constructed automaton skips while being in the error state.

For the Policy automaton from our example an edit automaton is constructed following the Alg. 2. This automaton has 9 states but 99 transitions. For the sake of brevity we will use our abbreviations for reading the figures and show only outcoming transitions of the states $q_4, q_6, q_\perp$ in Fig. 5.

Theorem 2. *An iterative property $\widehat{P}$ represented by Policy automaton A^P can be* iteratively enforced by suppression *by the edit automaton E constructed by the Alg. 2.*

Let us show in Fig. 6 the output of the Ligatti automaton (that all-or-nothing delayed precisely enforces) and edit automaton (that iteratively enforces by suppression) for the same Policy automaton. The input in the figure shows 5 iterations corresponding to the drug selection process. The whole input is not valid and can never become valid again because there is a second iteration inside the trace that can never become valid again. However, third and fifth iterations are valid, which means that doctor could successfully select the drugs he wanted.

First three input iterations in the Fig. 6 are iterations from the Ex. 2. Their validity can be checked by the Policy automaton shown in Fig. 2. The Ligatti automaton shown in Fig. 4 outputs the longest valid prefix, hence only the first iteration. It means that doctor will successfully complete selection process only for the first drug. Edit automaton shown in Fig. 5 will output all the successful iterations, which means it outputs first, third and fifth iterations.

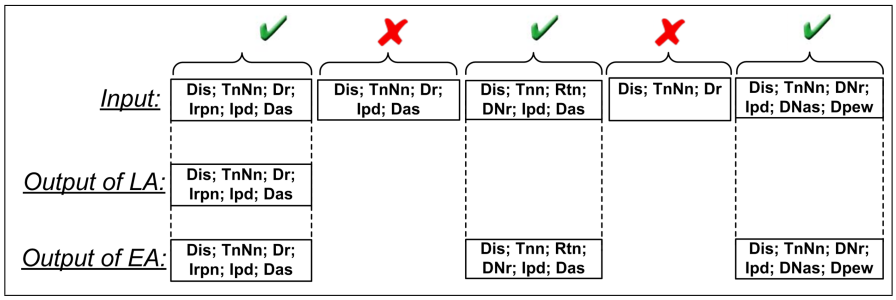


Fig. 6. Output of Ligatti (LA) and edit (EA) automata

If in Alg. 2 at lines 7, 9 and 11 we prefix the current output with the "warning" string we would get the edit automaton from Fig. 2 in [2]. This time it's made not manually but automatically.

7 Conclusions

Runtime enforcement is based on two simple ideas: the enforcement mechanism should leave good traces without changes and make sure that the bad ones got amended. From the theory side, a number of papers [6, 10, 12] provide the precise characterization of good executions that can be captured by a security policy and thus enforced by a specific mechanism. Unfortunately, those theories do not distinguish what happens when an execution is actually bad (the practical case). The theory only says that the outcome of enforcement mechanism should be "good" but not how far should the bad execution be changed.

If we consider a real-life example of a drug dispensation process in a hospital the notion of security automata or even edit automata would stop all requests by all doctors on all drugs and all dispensation protocols, as soon as a doctor forgot to insert the research protocol number.

In this paper we have shown a notion of enforcement that makes it possible to overcome the distinction between bad traces and good traces. The theoretical research on enforcement [6, 10, 12] only offered two properties: the mechanism should leave good traces untouched and make sure bad traces got amended. How they are amended was never formally specified. Notice that this was in contrast to the actually implemented systems. For instance Polymer did allow traces to be amended. But this was not reflected in the formal theory.

By revising the notion of good traces in terms of iterations we can offer a formal characterization of how enforcement mechanism deals with the bad traces.

These are the first steps towards closing the gap between the current theoretical works and their practical implementations. As a modest, but still telling example, the running example of Fig. 2 in [2] could not be generated from the policy by any of the formal construction appeared in that paper, nor in the constructions appearing in later papers. In contrast, it can be obtained by our second algorithm if, instead of suppressing every bad iteration, we emit a warning.

As a future work we consider the following. There are some actions that cannot be fixed in real life. We call these actions *observable* actions. This aspect is very important in real life when it involves the interactions of an organization with another one; for instance we can refer to the cases of outsourcing services or to the cases in which some requests are done to external parties. Moreover it is important to know that it is possible to have observable actions also within an organization; for example there are some physical actions that cannot be deleted. For instance when a doctor or a nurse, preparing a set of therapeutical drugs for a specific patient, takes a wrong drug from a stock it is not possible to delete this physical action with a given theoretical approach.

In the future a model for observable and fixable actions can be built, then it should be decided how the enforcement can be done under this assumption. For example, edit automaton can provide enforcement by insertion but cannot suppress observable actions.

We will also consider the case of multiple users and behavior of enforcement mechanism in that case. Indeed, many doctors may try to dispense drugs at the same time, and construction of enforcement mechanism can be different. We leave this problem for the future work.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Inform. Processing Letters*, 21(4):181–185, Cornell University, 1985.
2. L. Bauer, J. Ligatti, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *Int. J. of Inform. Sec.*, 4(1-2):2–16, Springer, 2005.
3. N. Bielova and F. Massacci. Do you really mean what you actually enforced? In *Proc. of the FAST'09 workshop*, volume 5491, pages 287–301, Springer, 2008.
4. U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. Technical report 2003-1916, Department of Computer Science, Cornell University, 2003.
5. P.W.L. Fong. Access control by tracking shallow execution history. *Proc. of Symp. on Sec. and Privacy*, pages 43–55, IEEE, 2004.
6. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *TOPLAS*, 28(1):175–205, ACM, 2006.
7. K.W. Hamlen, G. Morrisett, and F.B. Schneider. Certified in-lined reference monitoring on .net. In *Proc. of the workshop on Prog. Lang. and analysis for security (PLAS'06)*, pages 7–16, ACM Press, 2006.
8. L. Lamport. Proving the correctness of multiprocess programs. *TSE*, SE-3(2):125–143, IEEE, 1977.
9. J. Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, 2006.
10. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Inf. and Sys. Sec. (TISSEC)*, 12(3):1–41, ACM, 2009.
11. F.B. Schneider. Enforceable security policies. *ACM Transactions on Inf. and Sys. Sec. (TISSEC)*, 3(1):30–50, ACM, 2000.
12. C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Inform. and Comp.*, 206(2-4):158–184, Academic Press Inc., 2007.