# Do you really mean what you actually enforced? *

## Edit Automata revisited

Nataliia Bielova and Fabio Massacci

DISI - University of Trento, Italy,
surname@disi.unitn.it

**Abstract.** In the landmark paper on the theoretical side of Polymer, Ligatti and his co-authors have identified a new class of enforcement mechanisms based on the notion of edit automata, that can transform sequences and enforce more than simple safety properties.
We show that there is a gap between the edit automata that one can possibly write (e.g. by Ligatti himself in his running example) and the edit automata that are actually constructed according the theorems from Ligatii's IJIS paper and IC follow-up papers by Talhi et al. "Ligatti's automata" are just a particular kind of edit automata.
Thus, we re-open a question which seemed to have received a definitive answer: you have written your security enforcement mechanism (aka your edit automata); does it really enforce the security policy you wanted?

**Key words:** Formal models for security, trust and reputation, Resource and Access Control, Validation/Analysis tools and techniques

## 1 Introduction

The explosion of multi-player games, P2P applications, collaborative tools on Web 2.0, and corporate clients in service oriented architectures has changed the usage models of the average PC user: users demand to install more and more applications from a variety of sources. Unfortunately, the full usage of those applications is at odds with the current security model.

The first hurdle is certification. Certified application by trusted parties can run with full powers while untrusted ones essentially without any powers. However, certification just says that the code is trusted rather than trustworthy because the certificate has no semantics whatsoever. Will your apparently innocuous application collect your private information and upload it to the remote server [16]? Will your corporate client developed in out-sourcing dump your hard disk in a shady country? You have no way to know.

Model carrying code [18] or Security-by-Contract [4] which claim that code should come equipped with a security claims to be matched against the platform policies could be a solution. However this will only be a solution for certified code.

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

To deal with the untrusted code either .NET [12] or Java [7] can exploit the mechanism of permissions. Permissions are assigned to enable execution of potentially dangerous functionalities, such as starting various types of connections or accessing sensitive information. The drawback is that after assigning a permission the user has very limited control over its usage. An application with a permission to upload a video can then send hundreds of them invisibly for the user (see the Blogs on UK Channel 4's Video on Demand application). Conditional permissions that allow and forbid use of the functionality depending on such factors as the bandwidth or some previous actions of the application itself are currently out of reach. The consequence is that either applications are sandboxed (and thus can do almost nothing), or the user decided that they are trusted and then they can do almost everything.

To overcome these drawbacks a number of authors have proposed to enforce the compliance of the application to the user's policies by execution monitoring. This is the idea behind security automata [5, 8, 1, 17], safety control of Java programs using temporal logic specs [10] and history based access control [11].

In order to provide enforcement of security policies at run time by monitoring untrusted programs we want to know what kind of policies are enforceable and what sorts of mechanisms can actually enforce them. In a landmark paper [2] Bauer, Ligatti and Walker seemed to provide a definitive answer by presenting a new hierarchy of enforcement mechanisms and classification of security policies that are enforceable by these mechanisms.

Traditional *security automata* were essentially action observers that stopped the execution as soon as an illegal sequence of actions was on the eve of being performed. The new classification of enforcement mechanisms proposed by Ligatti included *truncation, insertion, suppression* and *edit automata* which were considered as execution transformers rather than execution recognizers. The great novelty of these automata was their ability to transform the "bad" program executions in good ones.

These automata were then classified with respect to the properties they can enforce: precisely and effectively enforceable properties. It is stated in [2] that as precise enforcers, edit automata have the same power as truncation, suppression and insertion automata. As for effective enforcement, it is said that edit automata can insert and suppress actions by defining *suppression-rewrite* and *insertion-rewrite* functions and thus can actually enforce more expressive properties than simple safety properties. The proof of Thm. 8 in [2] provides us with a construction of an edit automaton that can effectively enforce any (enforceable) property.

Talhi et al. [19] have further refined the notion by considering bounded version of enforceable properties.

### 1.1 Contribution of the Paper

If everything is settled why we need to write this paper? Everything started when we tried to formally show "as an exercise" that the running example of edit automaton from [2] provably enforces the security policy described in that

paper by applying the effective enforcement theorem from the very same paper. Much to our dismay, we failed.

As a result of this failure we decided to plunge into a deeper investigation and discovered that this was not for lack of will, patience or technique. Rather, the impossibility of reconciling the running example of a paper with the theorem on the very same paper is a consequence of a gap between the edit automata that one can possibly write (e.g. by Ligatti himself in his running example) and the edit automata that are actually constructed according Thm. 8 from [2] and Thm. 8 from [14] and the follow-up papers by Talhi et al. [19]. "Ligatti's automata" are just a particular kind of edit automata. Figure 3 later in the paper shows that we were trying to prove the equivalence of automata belonging to different classes, even though they are the "same" according to [2].

The contribution of this paper is therefore manyfold:

- We show the difference between the running example from [2] and the edit automata that are constructed according Thm. 8 in the very same paper.
- We introduce a more fine grained classification of edit automata introducing the notion of *Delayed Automata* and related security properties and relation between different notion of enforcement.
- We further explain the gap by showing that the particular automata that are actually constructed according Thm. 8 from [2] are a particular form of delayed automata that have an all-or-nothing behavior and that we named Ligatti's automata.

The remainder of the paper is structured as follows. At first we sketch the difference between the edit automaton from the running example and Thm. 8 from [2] (§2). Then we present the basic notions of policies, enforcement and automata in Section 3. We give a more fine grained classification of edit automata introducing the notion of *Delayed Automata* (§4). Section 5 explains relation between different notions of enforcement and types of edit automata. Finally we conclude with a discussion of future and related works (§6).

## 2 The example revised

*Example 1 (Verbatim from [2]).* To make our example more concrete, we will model a simple market system with two main actions, `take(n)` and `pay(n)`, which represent acquisition of $n$ apples and the corresponding payment. We let `a` range over all the actions that might occur in the system (such as `take`, `pay`, `window-shop`, `browse`, etc.) Our policy is that every time an agent takes $n$ apples it must pay for those apples. Payments may come before acquisition or vice versa, and `take(n)`; `pay(n)` is semantically equivalent to `pay(n)`;`take(n)`. The edit automaton enforces the atomicity of this transaction by emitting `take(n)`;`pay(n)` only when the transaction completes. If payment is made first, the automaton allows clients to perform other actions such as `browse` before committing (the `take-pay` transaction appears atomically after all such intermediary actions). On the other hand, if apples are taken and not paid for immediately, we issue

**Table 1.** Sequences of actions for market policy

| No. | Sequence of actions $\sigma$ | Expected output | $\widehat{P}(\sigma)$ |
|---|---|---|---|
| 1 | take(1) | · | × |
| 2 | pay(1) | · | × |
| 3 | take(1);browse | warning; browse | × |
| 4 | pay(1);browse | browse | × |
| 5 | pay(1); take(1) | pay(1);take(1) | √ |
| 6 | take(1);browse;pay(2) | warning;browse | × |
| 7 | take(1);browse;pay(2);take(2) | warning;browse;pay(2);take(2) | × |
| 8 | take(1);browse;pay(1) | warning;browse | × |
| 9 | take(1);browse;pay(1);take(2) | warning;browse | × |
| 10 | take(1);browse;pay(1);take(2);browse | warning;browse;warning;browse | × |
| 11 | take(1);browse;pay(1);take(2);<br>browse;pay(2) | warning;browse;<br>warning;browse | × |
| 12 | take(1); pay(2); take(2) | pay(2);take(2) | × |
| 13 | pay(1);browse;pay(2) | browse | × |
| 14 | pay(1);browse;pay(2);take(2) | browse;pay(2);take(2) | × |
| 15 | pay(1);browse;pay(2);take(2);browse | browse;pay(2);take(2);browse | × |

a warning and abort the transaction. Consistency is ensured by remembering the number of apples taken or the size of the prepayment in the state of the machine. Once acquisition and payment occur, the sale is final and there are no refunds (durability)."

In order to formally define the allowed and prohibited behavior described in the market policy we present: 1) a predicate $\widehat{P}$ over sequences of executions; 2) the expected output for every input sequence according the original example in Table 1.

Let us explain the expected output of some examples form Table 1, e.g. sequence 7:

- It contains `take(1)` action and `browse` action after it.
- Since there is no `pay(1)` action, the policy is violated. We expect the action `take(1)` be suppressed and output the `warning` action instead.
- The `browse` action does not violate the policy hence we output it.
- Next actions `pay(2)`;`take(2)` do not violate the policy.
- Therefore the output is `warning`;`browse`;`pay(2)`;`take(2)`.

In the sequences 13-15 the text of original example leaves opened a number of interpretations. It is clear that good sequences must have a pair of `take(n)` and `pay(n)` as the text implies, but it is not clear whether we allow interleaving of `pay(n)` and `pay(m)`. The text seems to imply that this is not possible so we mark them as violations.

We say that expected output is defined if either a `take-pay` transaction is completed (after the last `pay(n)` action there is a `take(n)` action) or the transaction is violated (after `take(n)` action there is an action different from `pay(n)`). But it is not clear how the sequence 14 should be transformed because

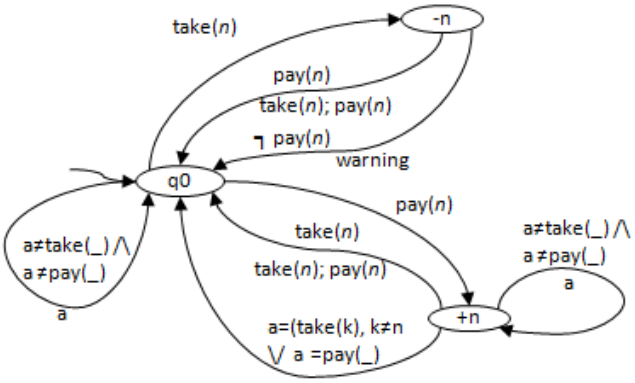


**Fig. 1.** An edit automaton that "effectively" enforces the market policy [2].

**Table 2.** Difference in output for edit automata

| No. | Input | Output | |
|---|---|---|---|
| | | *EA* from Fig. 1 [2] | Constructed *EA* by Thm.8 [2] |
| 1 | pay(1); take(1) | take(1);pay(1) | pay(1);take(1) |
| 2 | take(1); browse; pay(1); take(2); browse; pay(2) | warning;browse | · |
| 3 | take(1); pay(2); take(2) | warning | · |
| 4 | take(1); browse; pay(2); take(2) | warning;take(2); pay(2) | · |
| 5 | pay(1); browse; pay(2); take(2); browse | browse; warning | · |

we don't know if the `pay(1)` action should still be followed by the `take(1)` action or it should be simply suppressed.

The edit automaton that, as authors of [2] say, effectively enforces the market policy is shown in Fig. 1. But the definition of effective enforcement includes the notion of property $\widehat{P}$ : the predicate over the sequences of actions and this predicate is not given explicitly in [2]. That is why following the example in English we are presenting the Table 1 as a market policy. Assuming that our presentation of the policy corresponds to the Example 1, the given edit automaton [2] should effectively enforce the policy in Table 1.

According to the Thm. 8 of [2], any property $\widehat{P}$ can be effectively enforced by some edit automaton. We will construct such automaton according to the proof of this theorem (for more details about construction see technical report [3]).

After construction we discovered that edit automaton that effectively enforces $\widehat{P}$ (Fig. 1) and the one constructed by the proof of Theorem 8 (some edit automaton that effectively enforces $\widehat{P}$) produce different output for the same input. Let us show in Table 2 some cases of input and output of both automata.

Analysing the Table 2 we find out that the transformed sequences of actions are not always the ones expected from the edit automaton. So the question arises:

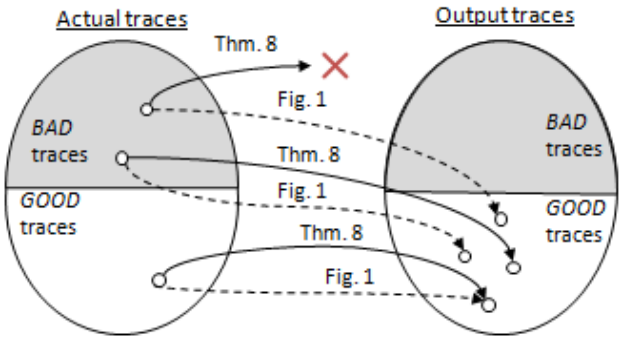


**Fig. 2.** Relation between input and output for edit automaton from Fig. 1 [2] and edit automaton constructed by Thm.8 [2]

*Why the output is predictable in some cases and unpredictable in the others?* The answer to this question is:

1. Both edit automata produce the expected output when the input sequence is legal (sequence 1 in the example)
2. The edit automaton constructed following the proof of Thm. 8 [2] is a very particular kind of the edit automaton.

When the sequence is illegal the output of both edit automata is unexpected. In Fig. 2 we show the relation between input and output for edit automaton from Fig. 1 [2] and edit automata constructed by Thm.8 [2] with respect to the "good" and "bad" traces. In case of "bad" input sequences edit automaton constructed by Thm.8 [2] outputs only the longest valid prefix: so either it outputs some valid sequence (ex.1 in Tab.2) or suppresses all the sequence (ex.2-5 in Tab.2). While edit automaton from Fig. 1 [2] always outputs some "good" sequence of actions even if the longest valid prefix is an empty sequence (ex.2-5 in Tab.2).

In order to explain this difference we analyze different classifications of edit automata that explain the behavior of the edit automaton constructed following the proof of Thm. 8 and the edit automaton from Fig. 1 [2]. For example, all theorems referring to edit automata in [19] are about the particular kind of automaton that is constructed following the proof of Thm. 8 [2].

## 3 Basic notions of policies, enforcement and automata

Similarly to [2] we specify the system at a high level of abstraction, where the set $\Sigma$ is the set of program actions; the set of all finite sequences over $\Sigma$ is denoted by $\Sigma^*$ , similarly the set of all infinite sequences is $\Sigma^\omega$ , and the set $\Sigma^* \cup \Sigma^\omega$ is a set of all finite and infinite executions. Execution $\sigma$ is a finite sequence of actions $a_1, a_2, ..., a_n$. In scope of this paper we assume only finite executions leaving infinite sequences of actions to be considered in future work.

With $\cdot$ we denote an empty execution. The notation $\sigma[i]$ is used to denote the $i$-th action in the sequence (begin counting at 0). The notation $\sigma[..i]$ denotes the subsequence of $\sigma$ involving the actions $\sigma[0]$ through $\sigma[i]$, and $\sigma[i+1..]$ denotes the subsequence of $\sigma$ involving all other actions. We use the notation $\tau;\sigma$ to denote the concatenation of two sequences.

As showed in Section 2 the constructed edit automaton following the algorithm in [2] and edit automaton presented in [2] are different. We give the original definition of edit automata from [2]:

An *Edit Automaton* $E$ is described by a 5-tuple of the form $\langle Q, q_0, \delta, \gamma, \omega\rangle$ with respect to some system with actions set $\Sigma$. $Q$ specifies possible states, and $q_0$ is the initial state. The partial function $\delta : (\Sigma \times Q) \rightarrow Q$ specifies the transition function; the partial function $\omega : (\Sigma \times Q) \rightarrow \{-, +\}$ has the same domain as $\delta$ and indicates whether or not the action is to be suppressed (-) or emitted(+); the partial function $\gamma$ is an insertion function, $\gamma : (\Sigma \times Q) \rightarrow \Sigma^* \times Q$. The partial functions $\delta$ and $\gamma$ have disjoint domains.

$$\boxed{(\sigma, q) \xrightarrow{\tau}_E (\sigma', q')} \tag{1}$$

$$(\sigma, q) \xrightarrow{a}_E (\sigma', q') \text{ if } \sigma = a;\sigma' \wedge \delta(a, q) = q' \wedge \omega(a, q) = + \tag{2}$$

$$(\sigma, q) \xrightarrow{\cdot}_E (\sigma', q') \text{ if } \sigma = a;\sigma' \wedge \delta(a, q) = q' \wedge \omega(a, q) = - \tag{3}$$

$$(\sigma, q) \xrightarrow{\tau}_E (\sigma, q') \text{ if } \sigma = a;\sigma' \wedge \gamma(a, q) = \tau; q' \tag{4}$$

$$(\sigma, q) \xrightarrow{\cdot}_E (\cdot, q) \text{ otherwise} \tag{5}$$

Assuming that the function $\gamma$ always inserts all necessary actions that have to appear before the $a$ action, we can rewrite the case of insertion as statement (4) and then statement (2). We consider that after inserting some actions $\tau$ at the next step the automaton will accept the current action $a$ (if inserting $\tau; a$ makes the output illegal then one can simply suppress $a$ without inserting $\tau$). Hence, the equation (4) can be represented as follows:

$$(\sigma, q) \xrightarrow{\tau;a}_E (\sigma', q') \text{ if } \sigma = a;\sigma' \wedge \gamma(a, q) = \tau; q' \tag{6}$$

In this way, the sequences $\sigma$ and $\sigma'$ are not relevant in the definition of transitions. Loosely speaking this was a Mealy-Moore transformation [9]. In order to give a formal definition in our notations, we will use the $\sigma_S$ sequence to define the sequence that was read but is not in the output yet.

**Definition 1 (Edit Automata ($EA$)).** *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* $\Sigma$. *$Q$ specifies possible states, and $q_0 \in Q$ is the initial state. The partial function $\delta : (Q \times \Sigma) \rightarrow Q$ specifies the transition function; the partial function $\gamma_o : (\Sigma^* \times Q) \rightarrow \Sigma^*$ defines the output of the transition according to the current state and the sequence of actions that is read but not in the output yet; the partial function $\gamma_k : (\Sigma^* \times Q) \rightarrow \Sigma^*$ 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* $\gamma_o(q, a, \sigma)$ *and* $\gamma_k(q, a, \sigma)$ *must be defined for all* $\sigma$*.*

$$\boxed{(q, \sigma_S) \overset{\gamma_o(q,\sigma_S;a)}{\longrightarrow}_E (q', \gamma_k(q, \sigma_S; a))} \tag{7}$$

In order for the enforcement mechanism to be effective all functions $\delta$, $\gamma_k$ and $\gamma_o$ should be decidable.

**Proposition 1.** *The Definition 1 of edit automaton has the same expressive power of the original definition [2].*

The proofs of all propositions and theorems are in the technical report [3].

## 4 A new classification of automata

Let us now give a deeper look at the automaton constructed according the proof of Thm. 8 [2]. In this construction at every state the automaton has emitted the sequence $\sigma'$, and $\sigma'$ is the *longest valid prefix* of the input sequence $\sigma$. Indeed, Table 2 shows that this statement holds for edit automaton constructed by Thm. 8 and it doesn't hold for the edit automaton from Fig. 1 [2]. Therefore, in order to understand what kind of edit automaton is in Fig. 1 we need to give a formal definition of this kind of automaton. This automaton outputs some valid prefix only when the sequence can become valid again in the future (e.g. for the sequence `take(1);pay(1);take(2)` after reading `take(1);pay(1)` the automaton will output these actions, and after reading the `take(2)` action it will still output the valid prefix `take(1);pay(1)`). And it outputs some corrected sequence (current valid prefix and some other sequence) if the sequence cannot become valid in the future(in example 2 of Table 2 after reading `take(1);browse` actions the automaton outputs another action `warning`).

This corresponds to the following intuition:

*Remark 1.* The automaton constructed according to the proof of Thm. 8 in [2] just delays the appearance of input actions until the input has built up a correct sequence again.

Formally, we propose a notion of wider class of such automata called *Delayed Automata*. They simply output some prefix of the input. These class will be the container of other less trivial cases when the property $\widehat{P}$ will be called into account.

**Definition 2 (Delayed Automata).** Delayed automaton A *is an edit automaton that is described by a 5-tuple of the form* $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$*, where the transition is defined as in equation (7) with the restriction that it always outputs some prefix of the input:*

$$\sigma_S; a = \gamma_o(q, \sigma_S; a); \gamma_k(q, \sigma_S; a) \tag{8}$$

In order to give a formal definition of the automata from Thm. 8 [2] for any property $\widehat{P}$ we present also a wider class of automata called *All-Or-Nothing Automata*. These automata always output a prefix of the input (hence it is a particular kind of the Delayed Automata). Moreover, at every step of the transition either it outputs all suspended inputs or suppresses the current action.

**Definition 3 (All-Or-Nothing Automata).** All-Or-Nothing automaton A *is an edit automaton described by a 5-tuple of the form* $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$*, where the transition relation is defined as in equation (7) with the following restrictions:*

- *This automaton outputs a prefix of the input: the statement (8) holds.*
- *At every step of the transition either it outputs the whole suspended sequence of actions or suppresses the current action:*

$$\gamma_o(q, \sigma_S; a) = \begin{cases} \sigma_S; a \\ . \end{cases} \tag{9}$$

The next step is the refinement of this class towards what we call *Ligatti Automata for* $\widehat{P}$. These automata always output a prefix of the input (hence it is a particular kind of the Delayed Automata) and they are particular kind of All-Or-Nothing automata. Moreover, they output the longest valid prefix. The definition of Ligatti Automaton for property $\widehat{P}$ given below was made according to the construction of edit automaton given in the proof of Thm. 8 [2].

**Definition 4 (Ligatti Automata for property $\widehat{P}$).** Ligatti automaton E for property $\widehat{P}$ *is an edit automaton described by a 5-tuple of the form* $E = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$*, where the set of states* $Q = \Sigma^*$ *(every state contains the already accepted sequence* $\sigma$*) and the transition relation is defined in a similar way as in equation (7):*

$$\boxed{(\sigma, \sigma_S) \xrightarrow{\gamma_o(\sigma, \sigma_S; a)} {}_E(\sigma; \gamma_o(\sigma, \sigma_S; a), \gamma_k(\sigma, \sigma_S; a))} \tag{10}$$

*With the following restrictions:*

- *The automaton outputs a prefix of the input (the statement (8) holds)*
- *Either it outputs the whole suspended sequence of actions or suppress the current action (the statement (9) holds).*
- *Output is a valid prefix of the input*

$$\widehat{P}(\sigma; \gamma_o(\sigma, \sigma_S; a) \tag{11}$$

- *If the current sequence is valid then it outputs the whole sequence:*

$$\textit{If } \widehat{P}(\sigma; \sigma_S; a) \textit{ then } \gamma_o(\sigma, \sigma_S; a) = \sigma_S; a. \tag{12}$$

At every state a Ligatti automaton for property $\widehat{P}$ keeps the sequence $\sigma$ that was read till the current moment in order to decide whether $\widehat{P}(\sigma; \sigma_S; a)$ holds. This explains why $Q = \Sigma^*$. In our definition a Ligatti Automaton for property $\widehat{P}$ is obviously a particular kind of Edit Automatfon. We will show that this statement holds in the original definition as well.

**Proposition 2.** *The Ligatti Automaton for property $\widehat{P}$ is an Edit Automaton according to Ligatti's own Definition.*

Let us now show the inverse of this claim: the edit automaton constructed following the proof of Thm. 8 [2] is a Ligatti Automaton for property $\widehat{P}$.

**Proposition 3.** *The Edit automaton constructed following the proof of Thm. 8 in [2] for property $\widehat{P}$ is a Ligatti Automaton for $\widehat{P}$.*

In a nutshell, the difference between edit automata and Ligatti automata for property $\widehat{P}$ is the following: edit automata suppress and insert arbitrary actions according to the given rewriting functions $\omega$ and $\gamma$ while Ligatti automata for property $\widehat{P}$ can only insert those actions that were read before; suppressed actions either will be inserted when the input sequence becomes valid or all subsequent actions will be suppressed. Thus Ligatti automata for property $\widehat{P}$ outputs the longest valid prefix of the input sequence.

Since the automaton constructed following the proof of Thm. 8 [2] is a Ligatti automaton for property $\widehat{P}$ while the automaton given in [2] (Fig. 1) is an edit automaton, the difference between their behaviors is not clear.

Still, the automaton of Fig. 1 is not a completely arbitrary edit automaton and we propose a notion of *Delayed Automaton for property $\widehat{P}$*. If the sequence is valid it outputs a valid prefix of the input, otherwise it can output some valid sequence (i.e. fixing the input).

**Definition 5 (Delayed Automata for property $\widehat{P}$).** Delayed automaton A for $\widehat{P}$ *is an edit automaton that is described by a 5-tuple of the form $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$, where the transition is defined in the same way as in equation (7) with the following restrictions:*

*If $\widehat{P}(\sigma; \sigma_S; a)$ then*

- *Output is a prefix of the input (the statement (8) holds) and*
- *Output is a valid prefix of the input (the statement (11) holds).*

Later in Fig. 3 we will pictorially describe the situation. However, in order to explain more relations present in that picture we need first to define the notion of enforcement in the next section.

## 5 A new classification of enforcement properties

The principles of soundness and transparency were presented in [2] in order to be able to compare different enforcement mechanisms. Let us first see an intuitive description of these mechanisms. The notion of *soundness* requires all the observable output of enforcement mechanism to be valid. The notion of *transparency* means that an enforcement mechanism must preserve the semantics of executions that are already valid. The notion of *precise* enforcement by [2] obeys both of these properties. According to that definition, the automaton in question outputs program actions in lock-step with the target program's action

stream if the action stream $\sigma$ is valid. Suppose that at the current moment the automaton reads $i$-th action in the sequence, and the sequence $\sigma[..i+1]$ is not valid. Then the automaton will not output any other actions.

In order to formalize the behavior where the automaton suppresses some actions and later insert them when the sequence turns out to be legal, we present the notion of *Delayed precise enforcement*.

**Definition 6 (Delayed Precise Enforcement).** *An edit automaton $A$ with starting state $q_0$* delayed precisely enforces *a property $\widehat{P}$ on the system with action set $\Sigma$ iff $\forall \sigma \in \Sigma^* \; \exists q' \; \exists \sigma' \in \Sigma^*$.*

1. $(\sigma, q_0) \xrightarrow{\sigma'}{}_A(\cdot, q')$, *and*
2. $\widehat{P}(\sigma')$, *and*
3. $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma' \wedge \forall i \; \exists j.\; j \leq i \; \exists q_*.\, (\sigma, q_0) \xrightarrow{\sigma[..j]}{}_A(\sigma[i+1..], q_*)$.

There is another notion of enforcement called "effective$_=$enforcement" [14], which also obeys the properties of soundness and transparency.

**Definition 7 (Effective$_=$Enforcement).** *An automaton $A$ with starting state $q_0$* effectively$_=$ enforces *a property $\widehat{P}$ on the system with action set $\Sigma$ iff $\forall \sigma \in \Sigma^* \; \exists q' \; \exists \sigma' \in \Sigma^*$.*

1. $(\sigma, q_0) \xrightarrow{\sigma'}{}_A(\cdot, q')$, *and*
2. $\widehat{P}(\sigma')$, *and*
3. $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$

Let us show the relation between delayed precise enforcement and effective$_=$enforcement.

**Theorem 1.** *If edit automaton $A$ delayed precisely enforces a property $\widehat{P}$ then it effectively$_=$ enforces property $\widehat{P}$.*

Let us come back to Example 1. As it is said in [2] the given edit automaton (Fig. 1) effectively$_=$enforces the market policy. But since the market policy is given only in natural language and the predicate $\widehat{P}$ is not given, statement "An edit automata effectively enforces the market policy" is stretching the definition.

Let us show in Fig. 3 all edit automata and its' particular subclasses presented above. The following theorems show the relation between different types of edit automata.

**Proposition 4.** *If edit automaton $A$ is a Delayed Automaton then it is not necessary that $A$ is a Delayed Automaton for property $\widehat{P}$.*

**Proposition 5.** *If edit automaton $A$ is a Delayed Automaton for property $\widehat{P}$ then it is not necessary that $A$ is a Delayed Automaton.*

From the Thm. 4 and Thm. 5 we can conclude that classes of Delayed Automata and Delayed Automata for $\widehat{P}$ have some common subclass but none of them include the other.

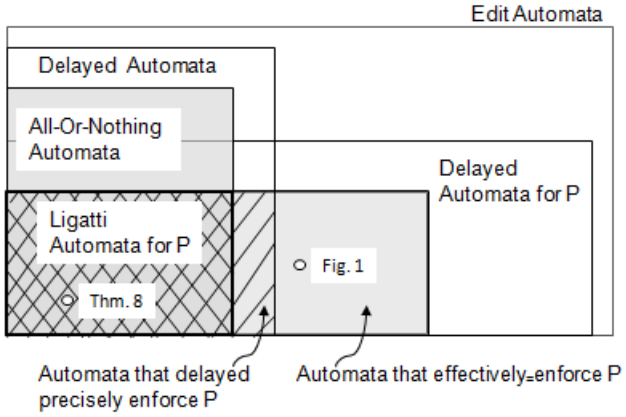


**Fig. 3.** The classes of Edit Automata.

**Theorem 2.** *If edit automaton $A$ effectively$_=$enforces property $\widehat{P}$ then $A$ is a Delayed Automaton for property $\widehat{P}$ and it is not necessary that $A$ is a Delayed Automaton.*

**Proposition 6.** *If edit automaton $A$ is a Delayed Automaton for property $\widehat{P}$ then it is not necessary that $A$ effectively$_=$enforces property $\widehat{P}$.*

From Thm. 2 and Proposition 6 we conclude that the class of edit automata that effectively$_=$enforces property $\widehat{P}$ is a particular class of Delayed Automata for $\widehat{P}$ and is not a proper subset of Delayed Automata.

Thm. 1 shows that edit automata that delayed precisely enforce property $\widehat{P}$ are a particular type of edit automata that effectively$_=$enforce $\widehat{P}$. The key point is that in the definition of delayed precise enforcement it is left open how illegal input is transformed.

Let us have a look at the 2d and 3d conditions of precise enforcement [2]:

$$\widehat{P}(\sigma')$$

$$\widehat{P}(\sigma) \Rightarrow \forall i\ \exists q''.\ (\sigma, q_0) \xrightarrow{\sigma[..i]}_A (\sigma[i+1..], q'')$$

These conditions mean that the automaton will produce an output *in a step-by-step* fashion with the monitored action stream and will output only a valid prefix. As soon as input sequence becomes illegal, the automaton will stop outputting. Therefore, in case of precise enforcement for illegal input, it will output some valid prefix $\sigma' = \sigma[..k]$ such that $\forall i.\ i \leq k.\ \widehat{P}(\sigma[..i]) \wedge \neg\widehat{P}(\sigma[..k+1])$. In case of delayed precise enforcement for illegal input the output will be some valid prefix $\sigma' = \sigma[..k]$ such that $\forall i.\ i \leq k.\ \widehat{P}(\sigma[..i])$.

**Theorem 3.** *Edit automaton $A$ delayed precisely enforces a property $\widehat{P}$ if and only if $A$ is a Delayed Automaton, $A$ is a Delayed Automaton for $\widehat{P}$ and it effectively$_=$enforces property $\widehat{P}$.*

**Proposition 7.** *Delayed automaton A that delayed precisely enforces property $\widehat{P}$ is not necessarily a Ligatti Automaton for $\widehat{P}$.*

**Proposition 8.** *All-Or-Nothing automaton A is a Delayed Automaton but not necessarily a Delayed Automaton for property $\widehat{P}$.*

**Theorem 4.** *All-Or-Nothing automaton A delayed precisely enforces a property $\widehat{P}$ if and only if A is a Ligatti Automaton for property $\widehat{P}$.*

Now we will define type of edit automaton constructed following the proof of Theorem 8 in [2] for property $\widehat{P}$ and type of edit automaton presented by the authors in [2] (Fig. 1).

As the Proposition 3 states, edit automaton constructed following the proof of Theorem 8 in [2] for property $\widehat{P}$ is a Ligatti Automaton for $\widehat{P}$. The edit automaton given in Fig. 1 [2] is an edit automaton that effectively$_=$enforces $\widehat{P}$: the 2d condition of effective$_=$enforcement is fulfilled (automaton always outputs the valid sequence) and the 3d condition is valid because in case of valid input it always outputs all the sequence. The edit automaton given in Fig. 1 [2] is not a Delayed Automaton because it does not always output some prefix of the input (see examples 2-5 in Tab.2)

Therefore we can conclude that both automata from Theorem 8 [2] and from Fig. 1 [2] are edit automata that effectively$_=$enforce property $\widehat{P}$. But when one wants to construct such an automaton and follows the proof of Theorem 8 [2], he obtains Ligatti Automaton for $\widehat{P}$ that delayed precisely enforces $\widehat{P}$.

## 6 Related work and Conclusions

Schneider [17] was the first to introduce the notion of enforceable security policies. The follow-up work by Hamlen et al. [8] fixed a number of errors and characterized more precisely the notion of policies enforceable by execution monitors as a subset of safety properties. They also analyzed the properties that can be enforced by static analysis and program rewriting. This taxonomy leads to a more accurate characterization of enforceable security policies. Ligatti, Bauer, and Walker [2] have introduced edit automata; a more detailed framework for reasoning about execution monitoring mechanisms. As we already said, in Schneider's view execution monitors are just sequence recognizers while Ligatti et al. view execution monitors as sequence transformers. Having the power of modifying program actions at run time, edit automata are provably more powerful than security automata [13].

Fong [6] provided a fine-grained, information-based characterization of enforceable policies. In order to represent constraints on information available to execution monitors, he used abstraction functions over sequences of monitored programs and defined a lattice on the space of all congruence relations over action sequences aimed at comparing classes of EM-enforceable security policies. Still his policies are limited to safety properties over finite executions.

Martinelli and Matteucci [15] have shown how to synthesize program controllers that monitor behavior of the untrusted components of the system. Given the system and a security policy represented as a $\mu$-calculus formula the user can choose the controller operator (truncation, suppression, insertion or edit automata). Then he can generate a program controller that will restrict the behavior of the system to those specified by the formula.

When a security policy is represented by a predicate $\widehat{P}$ over set of finite executions we can conclude that both automata from Thm. 8 [2] and from Fig. 1 [2] are edit automata that effectively$_{=}$enforce property $\widehat{P}$. If one wants to construct such an automaton and follows the proof of Thm. 8 [2], he obtains a Ligatti automaton for $\widehat{P}$ that delayed precisely enforces $\widehat{P}$. A problem that is present in the construction of Thm. 8 is that it assumes an oracle that can tell for each sequence $\sigma$ whether $\widehat{P}(\sigma)$ holds or not.

A security policy in Thm. 8 [2] is a predicate $\widehat{P}$ on all possible finite sequences of executions, but in this case the edit automaton which effectively enforces this policy is only of theoretical interest: following the proof of Thm. 8 only infinite states automata can be constructed.

In summary, we have shown that the difference between the running example from [2] and the edit automata that are constructed according Thm. 8 in the very same paper is due to a deeper theoretical difference. In order to understand this difference we have introduced a more fine grained classification of edit automata introducing the notion of *Delayed Automata*. The particular automata that are actually constructed according Thm. 8 from [2] are a particular form of delayed automata that have an all-or-nothing behavior and that we named Ligatti's automata after their inventor.

Hence, the construction from Talhi et al. [19] only applies to Ligatti's automata. Given a Ligatti automaton they can extract the Büchi automaton that represent the policy effectively enforced by the Ligatti automaton. What happens if the automaton is not a Ligatti automaton? For example the automaton from Fig. 1? Proposition 6.24 [19] simply does not apply. It needs to be shown whether given a general edit automaton one can construct a Büchi automaton so that the latter represents the policy that is effectively enforced by the former. We leave this question open for future investigation.

What remains to be done? Our results shows that the edit automaton that you can actually write (e.g. by using Polymer) does not necessarily correspond to the theoretical construction that provably guarantees that your automaton enforce your policy. A first step would be to find a construction that given a security policy represented as a Büchi automaton gives the Ligatti automaton that effectively enforces it.

So we fully re-open the most intriguing question that the stream of papers on execution monitors seemed to have closed: *you have written your security enforcement mechanism (aka your edit automata); how do you know that it really enforces the security policy you specified?*

## References


1. L. Bauer, J. Ligatti, and D. Walker. Composing security policies with polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 305–314. ACM Press, 2005.
2. L. Bauer, J. Ligatti, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.
3. N. Bielova and F. Massacci. Do you really mean what you actually enforced? Technical Report DISI-08-033, UNITN, 2008.
4. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *Proceedings of the 4st European PKI Workshop: Theory and Practice*. Springer-Verlag, 2007.
5. U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. Technical report 2003-1916, Department of Computer Science, Cornell University, 2003.
6. P.W.L. Fong. Access control by tracking shallow execution history. *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 43–55, May 2004.
7. L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.
8. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.
9. J. Hartmanis. *Algebraic structure theory of sequential machines*. Prentice-Hall, Englewood Cliffs, New Jersey, 1966.
10. K. Havelund and G. Rosu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technol. Transfer*, 2004.
11. K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems with applications to history-based access control. In *Proceedings of the 12th ACM Conference on Communications and Computer Security*, 2005.
12. B. LaMacchia and S. Lange. *.NET Framework security*. Addison Wesley, 2002.
13. J. Ligatti, L. Bauer, , and D. Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security*, pages 355–373, 2005.
14. Jarred Adam Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, June 2006.
15. F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. In *Proceedings of the Second International Workshop on Security and Trust Management*. Lecture Notes in Computer Science, 2006.
16. Bill Ray. Symbian signing is no protection from spyware. `http://www.theregister.co.uk/2007/05/23/symbian_signed_spyware/`, May 2007.
17. F.B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.
18. R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D.C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted applications. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 15–28. ACM Press, 2003.
19. C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2-4):158–184, 2007.