DO YOU REALLY MEAN WHAT YOU ACTUALLY ENFORCED?

Edit Automata revised

Nataliia Bielova and Fabio Massacci

July 2008

Technical Report # DISI-08-033

Do you really mean what you actually enforced? [⋆]

Edit Automata revised

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, corporate clients in service oriented architectures have changed the usage models of the average computer: 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 [13]? 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 [16] or Security-by-Contract [3] 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 [10] or Java [6] 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 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 [4, 7, 1, 14], safety control of Java programs using temporal logic specs [8] and history based access control [9].

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 action 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 kinds of automata was that they could transform the *bad* program executions in such executions that satisfy the security policy.

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 Theorem 8 in [2] provides us with a construction of an edit automaton that can effectively enforced any (enforceable) property. Talhi et al. [17] 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 significant 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 Theorem 8 from [2] and Theorem 8 from [12]and the follow-up papers by Talhi et al. [17]. ”Ligatti’s automata” are just a particular kind of edit automata. Figure 4 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 constructued 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.
– We show that the construction from Talhi et al. [17] only applies to Ligatti’s automata and therefore provide a more useful construction that is the inverse of Talhi et al. [17] construction: namely from a policy specification expressed as a Büchi automaton we show how to construct a Ligatii’s automaton that enforces it.

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 enforcement and introduce a more fine grained classification of edit automata introducing the notion of *Delayed Automata* (§3). Section 4 explains relation between different notions of enforcement and types of edit automata. Finally we conclude with a discussion of future and related works (§5).

2 The example revised

In this section we will use the very same example from of the simple market system from [2].

Example 1. Market system [1]

[1] The example is taken from the [2], p.11

"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 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. In addition, in column "Decision is finalized" we mark the sequences in which the decision about the output can be finalized and unmark those, where additional input is required to make a decision; in column "Violation" we mark the sequences, which violate the security policy and the `pay(n)` action does not appear immediately after the `take(n)` action.

According to the example 1 if there is an action different from `pay(n)` after `take(n)` then the `warning` is in the output and we mark the sequence as violation. Hence the sequences 3 and 6-12 are marked in the "Violation" column and violate the policy $\widehat{P}$. Let us explain the expected output from some of these examples. The sequence 7 contains `take(n)` action and `browse` action after it. Hence, the policy is violated and we expect the action `take(n)` be suppressed and output the `warning` action instead. The user cannot take some apples and browse after, but he can simply browse the apples without taking them. Also `pay(2)`;`take(2)` actions do not violate the policy. Therefore the output is `warning`;`browse`;`pay(2)`;`take(2)`. In the same way we obtain the expected output from the sequences 3, 6, 8-12.

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 the decision is made over the sequence 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 for instance in sequence 14 it is not clear whether the decision

Table 1. Sequences of actions for market policy

No.	Sequence of actions σ	Expected output	$\widehat{P}(\sigma)$	Decision is finalized	Violation
1	take(1)	$\cdot$	$\times$		
2	pay(1)	$\cdot$	$\times$		
3	take(1);browse	warning; browse	$\times$	$\surd$	$\surd$
4	pay(1);browse	browse	$\times$		
5	pay(1); take(1)	pay(1);take(1)	$\surd$	$\surd$	
6	take(1);browse;pay(2)	warning;browse	$\times$		$\surd$
7	take(1);browse;pay(2);take(2)	warning;browse; pay(2);take(2)	$\times$	$\surd$	$\surd$
8	take(1);browse;pay(1)	warning;browse	$\times$		$\surd$
9	take(1);browse;pay(1);take(2)	warning;browse	$\times$		$\surd$
10	take(1);browse;pay(1);take(2); browse	warning;browse; warning;browse	$\times$		$\surd$
11	take(1);browse;pay(1);take(2); browse;pay(2)	warning;browse; warning;browse	$\times$		$\surd$
12	take(1); pay(2); take(2)	pay(2);take(2)	$\times$		$\surd$
13	pay(1);browse;pay(2)	browse	$\times$		
14	pay(1);browse;pay(2);take(2)	browse;pay(2); take(2)	$\times$	$\surd$(?)	
15	pay(1);browse;pay(2);take(2); browse	browse;pay(2); take(2);browse	$\times$	$\surd$(?)	

is made because 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 enforces the policy in Table 1.

According to the Theorem 8 of [2], any property $\widehat{P}$ can be effectively enforced by some edit automaton.

For the sake of simplicity we build the edit automaton only partly (see Fig. 2) and the possible actions are $\Sigma =$ `take(1)`, `take(2)`, `pay(1)`, `pay(2)`, `browse`. Here we use a `browse` action just to present some other actions that the user can do after paying before taking the apples. According to the text of the example, an action `warning` is considered to be an output action in infinite-state edit automaton presented by the authors of [2] (see Fig. 1). On the other hand, the `warning` action is not in the set of possible input actions and hence it cannot appear in the output of automaton constructed following the algorithm given as a proof of Theorem 8 [2].

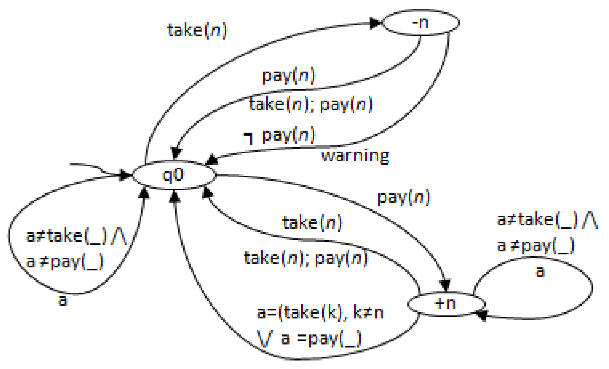

Fig. 1. An edit automaton that "effectively" enforces the market policy [2].

As the cardinality of input language is five ($|\Sigma| = 5$), every state will have five outcoming arcs for all possible actions. We will present here only some of them in order to let the reader see the output sequences for particular input sequences. Constructed automaton is shown in Fig. 2.

As proposed in [2] the state of automaton $q \in \Sigma^* \times \Sigma^* \times \{+, -\}$ (the sequence of actions seen so far, the actions seen but not emitted, and + (-) to indicate that the automaton must not (must) suppress the current action, the notation of Σ^* denotes the set of all finite-length sequences of actions on a system with action set Σ). For example, let us have a look at the the finite input sequence `pay(1);browse;pay(2);take(2);browse`. The correspondent trace is $q0, q1, q3, q5, q8, q11$:

- The initial state $q0 = \langle \cdot, \cdot, + \rangle$.
- When the action `pay(1)` is proceeding in state $q0$, it is suppressed because $\neg\widehat{P}(\texttt{pay(1)})$, so the next state is $q1 = \langle \texttt{pay(1)}, \texttt{pay(1)}, + \rangle$.
- At the next step the action `browse` is proceeding, since $\neg\widehat{P}(\texttt{pay(1)};\texttt{browse})$, this action is also suppressed and next state is $q3 = \langle \texttt{pay(1)};\texttt{browse}, \texttt{pay(1)};\texttt{browse}, + \rangle$.
- At every step the action is suppressed according to the predicate $\widehat{P}$. The next state is $q5 = \langle \texttt{pay(1)};\texttt{browse};\texttt{pay(2)}, \texttt{pay(1)};\ \texttt{browse};\ \texttt{pay(2)}, + \rangle$.
- $q8 = \langle \texttt{pay(1)};\texttt{browse};\texttt{pay(2)};\texttt{take(2)}, \texttt{pay(1)};\texttt{browse};\texttt{pay(2)};\texttt{take(2)}, + \rangle$.
- $q11 = \langle \texttt{pay(1)};\texttt{browse};\texttt{pay(2)};\texttt{take(2)};\texttt{browse}, \texttt{pay(1)};\texttt{browse};\ \texttt{pay(2)};\ \texttt{take(2)};\ \texttt{browse}, + \rangle$

After construction we discovered that for the same input 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. Let us show in Table 2 some cases when the constructed by Theorem 8 edit automaton produces "better" or more expected output and some other cases when the edit automaton from Fig.1 [2] produces "better" output.

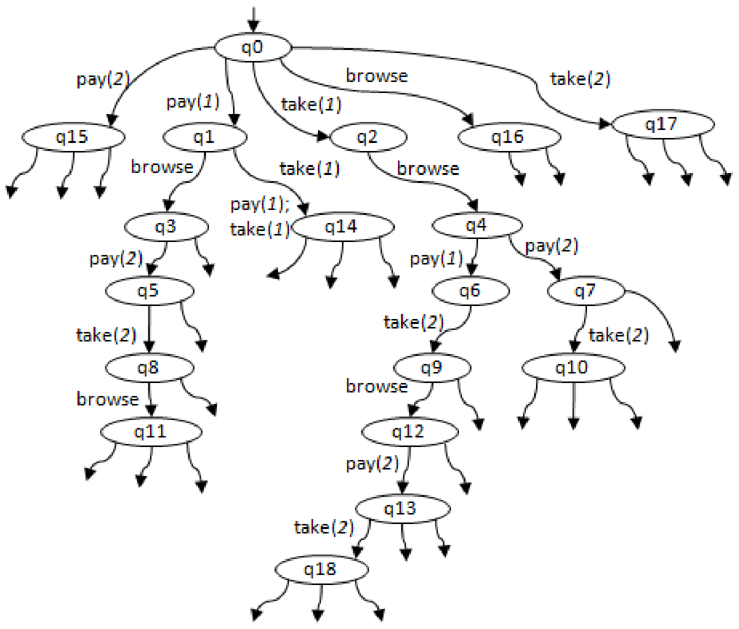

Fig. 2. Constructed edit automaton for the market problem (Ex.1)

In the lines 2-5 of the run, both Edit Automata produce the output that was was not expected by the user according to the Table 1. Only in line 1, when the input sequence is legal, the output is expected.

Let us consider the sequence 2 `take(1)`; `browse`; `pay(1)`; `take(2)`; `browse`; `pay(2)`. The expected output is `warning`; `browse`; `warning`; `browse` because the first two actions `take(1)`; `browse` are not legal and produce `warning` (the `pay` action must always go immediately after the `take` action), while `browse` action is legal by itself, so it can be in output sequence. Then the `pay(1)` action is simply supposed to be suppressed, then `take(2)`; `browse`; `pay(2)` part of the sequence should be transformed into `warning`;`browse` for the same reason. Instead, the sequence is transformed into `warning`;`browse` by the *EA* presented in [2]. This output is not expected by the user. The constructed *EA* outputs the empty sequence which is also not expected.

The sequence 5 `pay(1)`; `browse`; `pay(2)`; `take(2)`; `browse` is supposed to be transformed into `browse`; `pay(2)`; `take(2)`; `browse` because the action `pay(1)` without `take(1)` after it cannot be performed. Instead, the *EA* from [2] outputs only `browse`; `warning` sequence and the constructed *EA* transforms it into the empty sequence. So in this case both automata do not produce the expected output.

The sequence 1 is transformed by both automata as expected (in our market example policy `take(1)`; `pay(1)` is semantically equivalent to `pay(1)`; `take(1)`). After constructing *EA* following the algorithm and the examples presented in

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	·

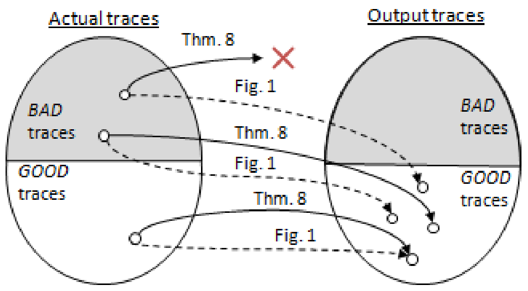

Fig. 3. Relation between input and output for edit automaton from Fig.1 [2] and edit automaton constructed by Thm.8 [2]

Table 2 we found out that the transformed sequences of actions are not always the ones expected from the EA. So the question arises: *Why the output is predictable in some cases and unpredictable in the others?* The answer to this question is:

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

When the sequence is illegal the output of both edit automata is unexpected. Further we will call illegal sequences as "bad" traces or "bad" sequences. In this paper we are going to discover the dependence between "bad" input sequences and expected output, and explain the anomaly of unexpected outputs. We will also analyze different particular types of edit automaton and reveal the type of edit automaton constructed following the proof of Theorem 8 and type of edit automaton from Fig.1 [2]. For example, all the theorems referring to edit automata in [17] are about that particular kind of automaton that is constructed following the proof of Theorem 8 [2].

In Fig.3 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. The main feature of their behavior is that if the sequence is "bad" constructed edit automaton by Thm.8 [2] outputs only the longest valid prefix: so either outputs some valid sequence (ex.1 in Tab.2) or suppresses all the sequence (ex.2-5 in Tab.2). On the other hand, edit automaton from Fig.1 [2] always outputs some "good" sequence of actions even if the longest valid prefix is empty sequence (ex.2-5 in Tab.2).

3 Basic Notions

Similarly to [2] we specify the system at a high level of abstraction, where the set Σ is the set of program actions; the set of all finite sequences over Σ is denoted by Σ^* similarly the set of all infinite sequences is Σ^ω and the set $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ is a set of all finite and infinite executions. Execution σ 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 σ involving the actions $\sigma[0]$ through $\sigma[i]$, and $\sigma[i+1..]$ denotes the subsequence of σ involving all other actions. We use the notation $\tau;\sigma$ to denote the concatenation of two sequences.

Let us present here two different approaches to present the *security policy.*

1. A security policy in [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 not constructive: following the proof of Thm.8 the only infinite states automaton can be constructed.
2. A security policy P specifying the desired behavior of the system can be represented as deterministic Büchi Automaton, in this case the edit automaton with finite number of states can be constructed for the given security policy

3.1 Generic definition of Edit Automata and its' particular types

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 Σ. 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 δ and indicates whether or not the action is to be suppressed (-) or emitted(+); the partial function γ is an insertion function, $\gamma : (\Sigma \times Q) \rightarrow \Sigma^* \times Q$. The partial functions δ and γ have disjoint domains.

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

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

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

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

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

Assuming that the function γ 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 τ 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 τ). 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 σ and σ' are not relevant in the definition of transitions. Loosely speaking this was a Mealy-Moore transformation. In order to give a formal definition in our notations, we will use the σ_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 Σ. Q specifies possible states, and q_0 is the initial state. The partial function $\delta : (\Sigma \times Q) \to Q$ specifies the transition function; the partial function $\gamma_o : (\Sigma^* \times Q) \to \Sigma^*$ 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 $\gamma_k : (\Sigma^* \times Q) \to \Sigma^*$ defined the sequence that will be kept after committing the transition.*

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

Proposition 1. *The definition of edit automaton $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ has the same expressive power of the original definition $A_L = \langle Q, q_0, \delta, \gamma_L, \omega_L \rangle$ [2].*

Proof. We simply define the keep function as $\gamma_k = \cdot$ and the output function as follows:

$$\gamma_o(q, \sigma_S; a) = \begin{cases} a & \text{if } \delta_L(q,a) = q' \wedge \omega_L(q,a) = +, \\ \cdot & \text{if } \delta_L(q,a) = q' \wedge \omega_L(q,a) = -, \\ \tau; a & \text{if } \gamma_L(a;q) = \tau; q', \\ \cdot & \text{otherwise.} \end{cases} \tag{8}$$

Similarly, for the suppression automaton we define $\gamma_k(q, \sigma_S; a) = \cdot$ and

$$\gamma_o(q, \sigma_S; a) = \begin{cases} a & \text{if } \delta_L(q,a) = q' \wedge \omega_L(q,a) = +, \\ \cdot & \text{if } \delta_L(q,a) = q' \wedge \omega_L(q,a) = -, \\ \cdot & \text{otherwise.} \end{cases} \tag{9}$$

For the insertion automaton $\gamma_k(q, \sigma_S; a) = \cdot$ and

$$\gamma_o(q, \sigma_S; a) = \begin{cases} a & \text{if } \delta_L(q, a) = q', \\ \tau; a & \text{if } \gamma_L(a; q) = \tau; q', \\ \cdot & \text{otherwise.} \end{cases} \tag{10}$$

□

Let us now give a deeper look at the automaton constructed according the proof of Theorem 8 [2]. In this construction at every state the automaton has emitted the sequence σ', and σ' is the *longest valid prefix* of the input sequence σ. Indeed, Table 2 shows that this statement holds for EA constructed by Theorem 8 and it doesn't hold for EA 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. At every step, this automaton outputs some valid prefix only when the sequences 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 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 transition is defined as follows:*

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

and the restriction that it always outputs a prefix of the input:

$$\sigma_S; a = \gamma_o(q, \sigma_S; a); \gamma_k(q, \sigma_S; a) \tag{11}$$

In order to give a formal definition of the automata from Theorem 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 (11) holds.*
- *At every step of the transition either it outputs all the suspended sequence of actions or suppresses the current action:*

$$\gamma_o(q, \sigma_S; a) = \begin{cases} \sigma_S; a \\ \cdot \end{cases} \tag{12}$$

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 Theroem 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* σ*) and the transition relation is defined in similar way as in equation (7):*

$$\boxed{(\sigma, \sigma_S) \overset{\gamma_o(\sigma,\sigma_S;a)}{\longrightarrow}{}_E (\sigma; \gamma_o(\sigma, \sigma_S; a), \gamma_k(\sigma, \sigma_S; a))} \tag{13}$$

With the following restrictions:

- *The automaton outputs a prefix of the input(the statement (11) holds):*

$$\sigma_S; a = \gamma_o(\sigma, \sigma_S; a); \gamma_k(\sigma, \sigma_S; a)$$

- *Either it outputs all the suspended sequence of actions or suppress the current action (the statement (12) holds):*

$$\gamma_o(\sigma, \sigma_S; a) = \begin{cases} \sigma_S; a \\ \cdot \end{cases}$$

- *Output is a valid prefix of the input:*

$$\widehat{P}(\sigma; \gamma_o(\sigma, \sigma_S; a) \tag{14}$$

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

$$\textit{If } \widehat{P}(\sigma; \sigma_S; a) \textit{ then } \gamma_o(\sigma, \sigma_S; a) = \sigma_S; a. \tag{15}$$

At every state the Ligatti automaton for property $\widehat{P}$ keeps the sequence σ that was read till the current moment in order to decide whether $\widehat{P}(\sigma; \sigma_S; a)$ holds. That explains why $Q = \Sigma^*$. In our definition the Ligatti Automaton for property $\widehat{P}$ is obviously a particular kind of Edit Automaton. 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 Ligatti's own Definition.*

Proof. Indeed, given a property $\widehat{P}$ and input sequence σ let us assume that at the current step $a = \sigma[i]$. Then let us define the rewrite functions ω and γ for Edit Automaton such that Ligatti Automaton for property $\widehat{P}$ is an Edit Automaton. In order to show this we need to keep in the state the sequence of actions σ_A that was already emitted and sequence of actions σ_S that was suppressed in every state: $q = \langle \sigma_A, \sigma_S \rangle$. This is exactly what is done in construction in proof of Theorem 8 [2]. Then we just use the following rewrite functions.

- $\omega(a, q) = +$ if $\widehat{P}(\sigma[..i]) \wedge \sigma_S = \cdot$;
  $\omega(a, q) = -$ if $\neg\widehat{P}(\sigma[..i])$ for suppression, and
- $\gamma(a, q) = \sigma_S, q' \wedge q' = \langle \sigma[..i]; \cdot \rangle$ if $\widehat{P}(\sigma[..i]) \wedge \sigma_S \neq \cdot$ for insertion.

□

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

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

Proof. Consider processing the action a, σ is input so far, σ_S is a suppressed sequence of actions. Let us have a look at two main steps of construction:

- if $\neg\widehat{P}(\sigma; a)$ then suppress a , $\sigma'_S = \sigma_S; a$.
- if $\widehat{P}(\sigma; a)$ then insert $\sigma_S; a$.

Since at every step the output is $\cdot$ or $\sigma_S; a$ then the automaton obeys the property (12) and also it always outputs prefix of the input, hence statement (11) holds as well. Constructed automaton outputs the sequence only if it is valid, hence statement (14) holds. It outputs all the suppressed actions if the sequence becomes valid, therefore statement (15) holds as well. Since all the conditions of Ligatti Automaton for $\widehat{P}$ are satisfied, we conclude that automaton constructed following the proof of Theorem 8 in [2] for property $\widehat{P}$ is Ligatti Automaton for $\widehat{P}$ □

In a nutshell, the difference between generic Edit Automata and Ligatti Automata for property $\widehat{P}$ is the following: generic Edit Automata suppress and insert arbitrary actions according to the given rewriting functions ω and γ 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 the Ligatti Automata for property $\widehat{P}$ outputs the longest valid prefix of the input sequence.

Since the automaton constructed following the proof of Theorem 8 [2] is a Ligatti Automaton for property $\widehat{P}$ while the automaton given in [2] (Fig.1) is a generic 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. a fixing of 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 (11) holds) and*
- *Output is a valid prefix of the input (the statement (14) holds).*

Later in Figure 4 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.

4 Types of enforcement

The following two main principles to compare different enforcement mechanisms were given in [2]:

Definition 6 (Soundness). *An enforcement mechanism must ensure that all observable outputs obey the property in question.*

Definition 7 (Transparency). *An enforcement mechanism must preserve the semantics of executions that already obey the property in question.*

The notion of *precise* enforcement by [2] obeys both Soundness and Transparency 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 σ 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 such a behavior when the automaton can suppress some actions and then later insert it when it turns out that the sequence is legal, we present the notion of *Delayed precise enforcement.*

Definition 8 (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* Σ *iff* $\forall \sigma \in \Sigma^* \ \exists q' \ \exists \sigma' \in \Sigma^*$.

1. $(\sigma, q_0) \xrightarrow{\sigma'}_A (\cdot, q')$,
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" [12], which also obeys the properties of soundness and transparency.

Definition 9 (Effective$_=$ Enforcement). *An automaton A with starting state* q_0 effectively$_=$ enforces *a property $\widehat{P}$ on the system with action set Σ iff* $\forall\sigma \in \Sigma^* \, \exists q' \, \exists\sigma' \in \Sigma^*$.

1. $(\sigma, q_0) \xrightarrow{\sigma'}_A (\cdot, q')$,
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 the edit automaton A delayed precisely enforces a property $\widehat{P}$ then it effectively$_=$enforces a property $\widehat{P}$.*

Proof. Since the 1st and 2nd conditions are equal for delayed precise enforcement and effective enforcement, we have to prove that if the 3d condition of delayed precise enforcement holds then the 3d condition of effective$_=$enforcement holds as well, hence we have to prove that if (16) then (17):

$$\widehat{P}(\sigma) \Rightarrow \sigma = \sigma' \wedge \forall i \, \exists j. \, j \leq i \, \exists q_*. \, (\sigma, q_0) \xrightarrow{\sigma[..j]}_A (\sigma[i+1..], q_*) \tag{16}$$

$$\widehat{P}(\sigma) \Rightarrow \sigma = \sigma' \tag{17}$$

We can see that formula (16) is a strengthening of (17). □

Let us come back to Example 1. It is said in [2] that the given edit automaton (Fig. 1), effectively enforces the market policy. But since the market policy is given only in natural language, stating that "An edit automata effectively enforces the market policy" is stretching the definition a bit as the market policy is not defined: the predicate $\widehat{P}$ is not given.

Let us show in Fig. 4 all edit automata and its' particular subclasses presented above. The following theorems show the relation between different types of edit automata.

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

Proof. By σ we denote an input sequence of the automaton and σ' is an output sequence. A Delayed automaton A obeys only one property: it always outputs prefix of the input (statement (11) holds). Hence, even if the overall input sequence σ is valid A can output an invalid prefix of the input ($\neg\widehat{P}(\sigma')$), while Delayed Automaton for property $\widehat{P}$ will always output a valid prefix (statement (14)). □

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

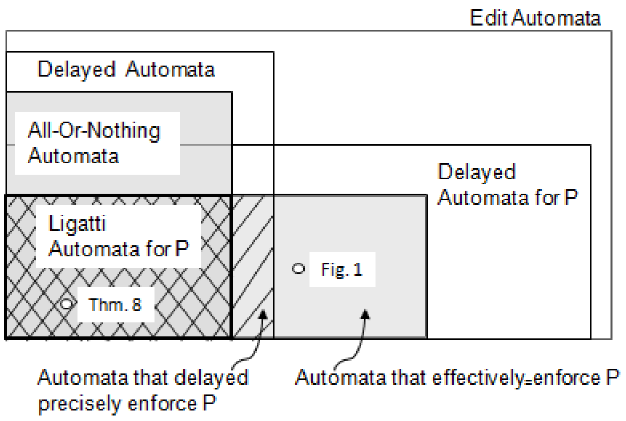

Fig. 4. The classes of Edit Automata.

Proof. In case of invalid input sequence Delayed Automaton for property $\widehat{P}$ can output another sequence which is not necessarily a prefix of the input, while Delayed Automaton will always output a prefix of the input (item 11). □

From the theorems 4 and 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.

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.*

Proof. By σ we denote an input sequence of the automaton and σ' is an output sequence. The automaton A that effectively$_=$enforces $\widehat{P}$ obeys the following properties: a) $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$; b) $\widehat{P}(\sigma')$. Hence it always outputs the valid prefix of the input (the whole sequence in this case), so automaton A is a Delayed Automaton for $\widehat{P}$ according to its' definition.

In case of invalid input, automaton A will output some valid sequence (according to the property b) of effective enforcement) which is not necessary prefix of the input. Therefore it is not necessarily 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 $\widehat{P}$.*

Proof. By σ we denote an input sequence of the automaton and σ' is an output sequence. In case of a valid input the Delayed Automaton for $\widehat{P}$ will output some valid prefix of the input and not necessary the whole input, hence the property of effective$_=$enforcement $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$ will not hold. □

From Theorem 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.

Theorem 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:

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

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

These conditions mean that the automaton will produce the 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. *An edit automaton A delayed precisely enforces property $\widehat{P}$ if and only if A is Delayed Automaton, A is Delayed Automaton for $\widehat{P}$ and it effectively$_=$enforces property $\widehat{P}$.*

Proof. Similarly to definitions of delayed precise enforcement and effective$_=$enforcement by σ we denote an input sequence of the automaton and σ' is an output sequence. (*If Direction*). If edit automaton A delayed precisely enforces property $\widehat{P}$ then it always outputs a prefix of the input, therefore statement (11) holds and A is Delayed Automaton. Since the 2nd condition of delayed precise enforcement states that $\widehat{P}(\sigma')$ then the statement (14) of Delayed Automaton for $\widehat{P}$ holds, therefore A is Delayed Automaton for $\widehat{P}$. According to theorem 1, since A delayed precisely enforces $\widehat{P}$ it also effectively$_=$enforces $\widehat{P}$.
(*Only-if direction*). If A is Delayed Automaton, A is Delayed Automaton for $\widehat{P}$ and it effectively$_=$enforces property $\widehat{P}$ then it always outputs some valid prefix of the input (property (11) of Delayed Automaton and property $\widehat{P}(\sigma')$ of effective$_=$enforcement) and in case of valid input it outputs all the input sequence (property $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$ of effective$_=$enforcement). Hence, the 2nd property of delayed precise enforcement holds: $\widehat{P}(\sigma')$. The 3d property holds as well because in this case at every step Delayed Automaton for $\widehat{P}$ outputs some valid prefix and finally it outputs the input sequence because of effective$_=$enforcement. Hence A delayed precisely enforces $\widehat{P}$. □

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

Proof. In case of illegal input sequences a Delayed Automaton will output some prefix and since it delayed precisely enforces property $\widehat{P}$ it will output some (not necessarily the longest one) valid prefix of the input. While Ligatti Automaton for $\widehat{P}$ will always output only the longest valid prefix of input. Therefore, an automaton A may not be a Ligatti Automaton for $\widehat{P}$. □

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

Proof. Since statement (11) holds for All-Or-Nothing automaton A then A is a Delayed Automaton. A is not a Delayed Automaton for $\widehat{P}$ because it can output some prefix of the input such that $\neg\widehat{P}(\sigma')$. $\square$

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

Proof. (*If Direction*). If automaton A is All-Or-Nothing automaton then the statement (12) holds. Since A delayed precisely enforces property $\widehat{P}$ then according to the theorem 3: a) A is a Delayed Automaton hence the statement (11) holds; b) A is Delayed Automaton for $\widehat{P}$ therefore statement (14) holds; c) A is All-Or-Nothing automaton that effectively$_=$enforces $\widehat{P}$ hence at every step it outputs all suspended sequence or suppress the action (statement (11))and and it always outputs the input sequence σ if $\widehat{P}(\sigma)$. Therefore in case of valid input ($\widehat{P}(\sigma)$) the output is equal to the input, so statement (15) holds as well. We have proved that A obeys all the conditions in the definition of Ligatti Automaton for $\widehat{P}$.
(*Only-if direction*). If A is Ligatti Automaton for $\widehat{P}$ then

– It obeys the property (12) hence A is All-Or-Nothing Automaton;
– It obeys the property (11) hence A is Delayed Automaton;
– It obeys the property (14) hence A is Delayed Automaton for $\widehat{P}$;
– It obeys the property (15) hence A always outputs input sequence in case of valid input and the longest valid prefix in case of illegal input, hence A effectively$_=$enforces $\widehat{P}$.

Therefore since A is Delayed Automaton, A is Delayed Automaton for $\widehat{P}$ and it effectively$_=$enforces $\widehat{P}$ then it delayed precisely enforces $\widehat{P}$ according to the theorem 3. $\square$

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 $\widehat{P}$.

5 Related work and Conclusions

Schneider [15] was the first to introduce the notion of enforceable security policies. The follow-up work by Hamlen et al. [7] fixed a number of errors and characterized more precisely the notion of policies enforceable by execution monitoring by security automata, and identifying enforceable policies 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. Jay Ligatti, Lujo Bauer, and David 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 [11].

Fong [5] 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.

In case when security policy is represented by predicate $\widehat{P}$ over set of finite executions 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 a Ligatti Automaton for $\widehat{P}$ that delayed precisely enforces $\widehat{P}$. If the security policy is represented as Büchi Automaton A_{EM}, the edit automaton that effectively$_=$enforces this policy can be constructed and in this case A_{EM} is also a Ligatti automaton for $\widehat{P}$ that delayed precisely enforces $\widehat{P}$.

In summary we have shown that the difference between the running example from [2] and the edit automata that are constructued 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 in honor of their inventor.

Hence, the construction from Talhi et al. [17] 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 for Fig.1? Proposition 6.24 [17] simply does not apply.

What remains to be done? Our results shows that the edit automaton that you can actually write (eg by using Polymer) does not necessarily correspond to the theoretical construction that provably guarantees that your automaton enforce your policy.

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 *Proc. of the ACM SIGPLAN 2005 Conf. on Prog. Lang. 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. *Int. J. of Inform. Sec.*, 4(1-2):2–16, 2005.
3. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *Proc. of EuroPKI'07*. Springer-Verlag, 2007.
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, May 2004.
6. L. Gong and G. Ellison. *Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation*. Pearson Education, 2003.
7. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *TOPLAS*, 28(1):175–205, 2006.
8. K. Havelund and G. Rosu. Efficient monitoring of safety properties. *Int. J. on Software Tools for Technol. Transfer*, 2004.
9. K. Krukow, M. Nielsen, and V. Sassone. A framework for concrete reputation-systems with applications to history-based access control. In *Proc. of CCS'05*, 2005.
10. B. LaMacchia and S. Lange. *.NET Framework security*. Addison Wesley, 2002.
11. J. Ligatti, L. Bauer, , and D. Walker. Enforcing non-safety security policies with program monitors. In *Proc. of the 10th ESORICS*, pages 355–373, 2005.
12. Jarred Adam Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, June 2006.
13. Bill Ray. Symbian signing is no protection from spyware. `http://www.theregister.co.uk/2007/05/23/symbian_signed_spyware/`, May 2007.
14. F.B. Schneider. Enforceable security policies. *J. of the ACM*, 3(1):30–50, 2000.
15. F.B. Schneider. Enforceable security policies. *TISSEC*, 3(1):30–50, 2000.
16. 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 *Proc. of the 19th ACM Symp. on Operating Sys. Princ.*, pages 15–28. ACM Press, 2003.
17. C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Inform. and Comp.*, 206(2-4):158–184, 2007.