International Journal of Information Security manuscript No.
(will be inserted by the editor)

Nataliia Bielova · Fabio Massacci

Do you really mean what you actually enforced?

Edit Automata revisited

Abstract In their works 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 et al in their IJIS running example) and the edit automata that are actually constructed according the theorems from Ligatti's IJIS paper or from 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?

Keywords 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 PC users: users demand to install more and more interactive applications from a variety of sources. Unfortunately, the features of those applications are 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 a remote server [19]? 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 [21] or Security-by-Contract [3] which claim that code should come equipped with security claims to be matched against the platform policies could be a solution. However, this will only be a solution for certified code.

To deal with the untrusted code either .NET [14] or Java [10] can exploit the mechanism of permissions. Permissions are assigned to enable execution of potentially dangerous functionalities, such as starting 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 [7]). 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 [1,8,11,20], safety control of Java programs using temporal logics [12] and history based access control [13].

In order to provide enforcement of security policies by runtime monitoring untrusted programs, we want to know what policies are enforceable and what 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

A preliminary, much shorter version of this paper appears in the informal proceedings of FAST'08 [4].

DISI - University of Trento,
via Sommarive 14, 38123 Povo, Trento, Italy
E-mail: surname@disi.unitn.it

mechanisms and a 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 Theorem 8 in [2] provides us with a construction of an edit automaton that can effectively enforce any (enforceable) property.

Talhi et al. [22] have further refined the notion by considering bounded version of enforceable properties.

1.1 Contribution of the Paper

If everything is settled why are we writing 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 following Theorem 8 from [2] or Theorem 3.3 from [15] or Talhi et al. [22]. The edit automata constructed according to those theorems are just a particular kind of edit automata. We named them "Ligatti" automata.

In Figure 1 we show the relation between different classes of automata we are investigating in this paper. *All-Or-Nothing automata* at every step output the whole input sequence or suppress the current action. The notion of effective_enforcement is taken from [2]. *Late automata* are a particular kind of edit automata that always output some prefix of the input.

Figure 8 later in the paper shows the relations among different classes of edit automata, even though they are the "same" according to [2].

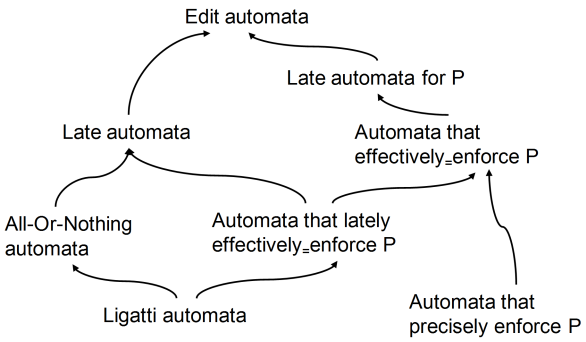


Fig. 1: Relation between classes of edit automata

The contribution of this paper is therefore manifold:

- We introduce a fine grained classification of edit automata and related security properties and relation between different notions of enforcement.
- We show the difference between the running example from [2] and the edit automata that are constructed according to the Theorem 8 in the very same paper.
- We further explain the gap by showing that the particular automata constructed according to Theorem 8 in [2] are a particular form of late automata that have an all-or-nothing behavior (*Ligatti automata*).
- We show that the construction from Talhi et al. [22] only applies to Ligatti automata and provides a more useful construction that is the inverse of Talhi et al. [22] construction: namely from a policy specification expressed as a Büchi automaton, we show how to construct a Ligatti'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 Theorem 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 *Late automata* (§4). Section 5 explains relation between different notions of enforcement and types of edit automata. We provide the construction of Ligatti's automaton that enforces a policy expressed as a Policy automaton(§6). Finally we conclude with a discussion of future and related works (§7).

2 The example revised

We present the example of the market policy verbatim from [2], we will use it throughout this paper.

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 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).

Looking at the example in English we propose several sequences of actions in Table 1: some satisfy the policy and some do not. When the sequence is not allowed by the policy, the enforcement mechanism should change the sequence in such a way that it becomes legal. In this table we partition the sequences in several groups. In the first group, sequences are not yet finished so that we cannot define whether they are legal (which means satisfy the policy from Example 1) or not, because they could become good later on. For example, in sequence 1 the `take(1)` action can be followed by a `pay(1)` action that can make the whole sequence legal; on the other hand it can be followed by `browse` that will make the sequence illegal.

The group "Legal sequences" contains sequences that satisfy the policy: for example `take(1); pay(1)`.

The initially illegal sequences are divided into two groups. One group contains the sequences with an initially bad prefix but the suffix can become legal later. For example, sequence 6 has an illegal prefix `take(1) browse`, however the suffix `pay(2)` can be extended to legal sequence since it can be a beginning of sequence 5: `pay(2); take(2)`. The second group contains sequences that have an illegal prefix followed by a legal continuation, such as sequence 8 which has an illegal prefix `take(1); browse`, but its suffix is a legal sequence 5.

There are some other sequences like `pay(1); browse; pay(2); take(2); take(1)` that we could not add to neither of the groups. This happens because the text leaves open 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.

The enforcement mechanism by means of edit automaton proposed in [2] for Example 1 is shown in Figure 2a. The nodes in the picture represent the automaton states and the arcs represent the transitions. The action that is above the arc defines an input action. Output actions are underlined and placed below the arcs. Arcs with no underlined sequence represent transitions where there is no output. If there is no arc for the current action, then the automaton halts.

Table 1: Sequences of actions for market policy

Temporarily illegal sequences that can become good

No	Sequence of actions	Expected output
1	take(1)	·
2	pay(2)	·
3	pay(2); browse	browse

Legal sequences

No	Sequence of actions	Expected output
4	take(1); pay(1)	take(1); pay(1)
5	pay(2); take(2)	pay(2); take(2)

Initially illegal sequences, but a later suffix can become good

No	Sequence of actions	Expected output
6	take(1); browse; pay(2)	warning
7	take(1); pay(2)	warning

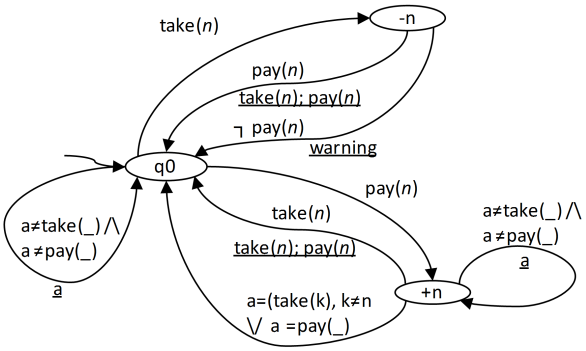
Initially illegal sequences with legal continuation

No	Sequence of actions	Expected output
8	take(1); browse; pay(2); take(2)	warning; browse; pay(2); take(2)
9	take(1); pay(2); take(2)	pay(2); take(2)
10	pay(1); browse; pay(2); take(2)	browse; pay(2); take(2)

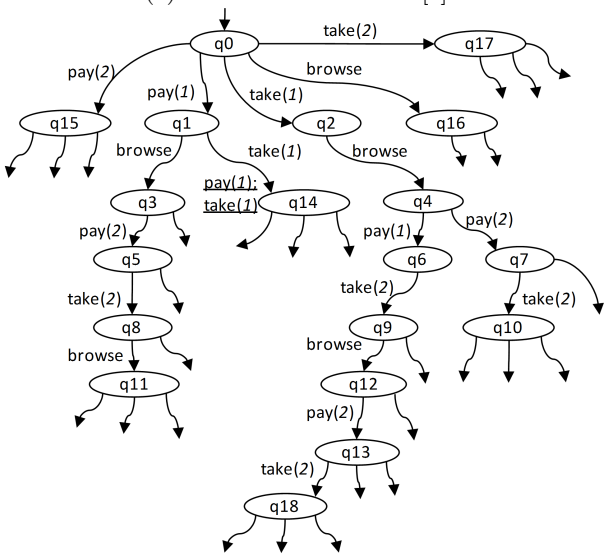
Let us look at sequence 4 from the Table 1: `take(1); pay(1)`. The edit automaton starts in the initial state $q0$. When it reads the first action `take(1)`, it moves to the corresponding state $-n$ and waits for the next input action to arrive. When the action `pay(1)` arrives, the automaton moves back to the state $q0$ while outputting the `take(1); pay(1)` sequence (as indicated at the output of the arc from $-n$ to $q0$).

The policy given in Example 1 can be enforced by different edit automata in different ways. One of the way to enforce the policy is called *effective enforcement*. This notion guarantees that all legal sequences are not changed by the edit automaton, while all illegal sequences are changed to *some* legal sequences or to nothing. The automaton in Figure 2a provides effective enforcement of the given policy since it does not change the good sequences and always changes illegal sequences to the legal ones. In the following, we will use the notion of *property*, which is simply a predicate that decides whether the sequence is legal or not.

In [2] Theorem 8 says that any property can be effectively enforced by an edit automaton. In Figure 2b we follow the construction from the proof of Theorem 8 in order to build the edit automaton. The automaton is built as follows: for all possible inputs, if the input is legal then automaton outputs it, otherwise it keeps the input until it becomes legal again. Notice that the constructed automaton always outputs the longest valid prefix of the given sequence. In case of invalid prefix, this automaton will output nothing.



(a) Edit automaton from [2].



(b) Edit automaton from the proof of Theorem 8 [2].

Fig. 2: Different edit automata that "effectively" enforce the market policy (Example 1).

For sake of simplicity, we show a constructed edit automaton only partially, from the action set {`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, an action `warning` is considered to be an output action in the edit automaton in [2] (see Figure 2a). As the cardinality of input language is 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.

The edit automaton from the original paper (Figure 2a) and the one constructed by the proof of Theorem 8 of the same paper (Figure 2b) actually produce different output for the same input. In Table 2 we show some cases of input and output of both automata.

Sequence 4 is legal, hence it is not changed by both automata. Sequence 6 has illegal prefix and not yet legal suffix, hence both automata halt (let us ignore the `warning` action since it can be easily added to the construction from Theorem 8). However, for other bad sequences that have different nature, the automata behave differently. The output set of the automaton from Figure 2a [2] is larger than the output of the automaton from Theorem 8 [2].

Table 2: Difference in output for edit automata

No	Input	Output	
		Edit automaton from Figure 2a [2]	Constructed edit automaton by Theorem 8 [2]
4	take(1); pay(1)	take(1); pay(1)	take(1); pay(1)
6	take(1); browse; pay(2)	warning	·
8	take(1); browse; pay(2); take(2)	warning; pay(2); take(2)	·
9	take(1); pay(2); take(2)	warning	·
10	pay(1); browse; pay(2); take(2)	browse	·

The "strange" behavior begins when we take the sequences from the group "Initially illegal sequences with legal continuation", like sequence 8. The edit automaton from Figure 2a [2] can skip the illegal prefix `take(1)`; `browse` and output the legal suffix `pay(2)`; `take(2)`. However, the automaton from Theorem 8 halts as soon as an illegal prefix does not have a legal continuation. In case of sequences 9 and 10 both automata produce a strange output in a sense that the agent has paid twice but never got any apple.

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

1. When the input sequence is legal both edit automata produce the expected output (e.g. sequence 4).
2. When the sequence is illegal the output of both edit automata is unexpected and potentially different for each automaton.
3. The edit automaton constructed following the proof of Theorem 8 [2] is a very particular kind of the edit automaton.

In Figure 3, we show the relation between input and output for edit automaton from Figure 2a [2] and edit automata from Theorem 8 [2] with respect to the "good" and "bad" traces. By 4;8 we mean the concatenation of the sequence 4 with the sequence 8. By 8out we mean an output sequence of Figure 2a automaton when processing sequence 8 as input.

We investigate the problem of sequences with illegal prefix followed by a valid suffix in [5]. The idea is that in sequences such as 8, 9 and 10 the edit automaton should output only the valid suffix ignoring the illegal prefix. We refer the reader to [5] for more details.

In order to explain better this difference, we analyze different classifications of edit automata that explain the behavior of the edit automaton constructed following the

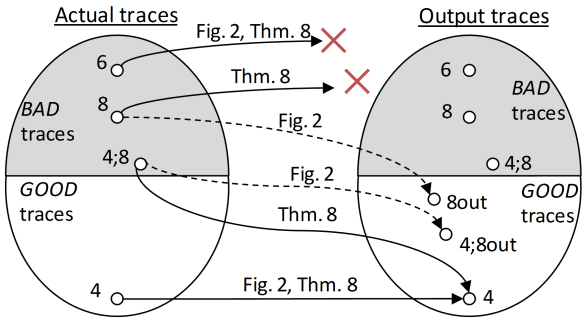


Fig. 3: Relation between input and output for edit automaton from Figure 2a [2] and edit automaton constructed by Theorem 8 [2], or Theorem 3.3 [15]

proof of Theorem 8 and the edit automaton from Figure 2a [2]. For example, all theorems referring to edit automata in [22] are about the particular kind of automata that are constructed following the proof of Theorem 8 [2].

3 Basic notions of policies, enforcement and automata

Similarly to [2,15], 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^*, the set of all infinite sequences is Σ^ω and the set of all sequences (finite and infinite) is Σ^∞. Executions are denoted by σ and actions by a possibly with subscripts or superscripts.

With $\cdot$ we denote an empty execution. The notation $\sigma[i]$ is used to denote the ith action in the sequence. The notation $\sigma[..i]$ denotes the prefix of σ involving the actions $\sigma[1]$ through $\sigma[i]$, and $\sigma[i+1..]$ denotes the suffix of σ involving all other actions beside $\sigma[..i]$. We use the notation $\tau;\sigma$ to denote the concatenation of two sequences.

Definition 1 (Edit automata) 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 the possible states, and $q_0 \in Q$ is the initial state. The total function $\delta : (Q \times \Sigma) \to Q$ specifies the transition function; the total function $\gamma_o : (Q \times \Sigma^* \times \Sigma) \to \Sigma^*$ defines the output of the transition according to the current state, the current input action and the sequence of actions kept so far; the total function $\gamma_k : (Q \times \Sigma^* \times \Sigma) \to \Sigma^*$ 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 $\gamma_o(q, \sigma_k, a)$ and $\gamma_k(q, \sigma_k, a)$ must be defined.

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

The intuition behind the output function γ_o is that at each transition when the automaton proceeds with one more input action, the function defines the output of the automaton at this transition. The intuition of the keep function γ_k is that at each transition it defines the buffer containing the actions that are processed by the automaton but not output yet. Usually, we will use the keep function to add the input action to the buffer or ignore the input action. In the general case, the keep function can perform more actions on the current buffer, for example to add arbitrary actions to it. This means that the state search space of the automaton is $Q \times \Sigma^*$, because each state consists of the control state and the buffer defined by the keep function.

Definition 2 (Run of an Edit automaton) Let $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k\rangle$ be an edit automaton. A *run* of A on an *input* sequence of actions $\sigma = a_1; a_2; \ldots$ is a sequence of pairs $\langle (q_0, \epsilon), (q_1, \sigma_1^k), (q_2, \sigma_2^k), \ldots\rangle$ such that $q_{i+1} = \delta(q_i, a_{i+1})$ and $\sigma_{i+1}^k = \gamma_k(q_i, \sigma_i^k, a_{i+1})$. The *output* of A on input σ is sequence of actions $\sigma^o = \sigma_1^o; \sigma_2^o; \ldots$ such that $\sigma_{i+1}^o = \gamma_o(q_i, \sigma_i^k, a_{i+1})$. We denote a finite run of n steps as

$$(q_0, \sigma) \overset{\sigma_{out}}{\rightsquigarrow}_A (q_n, \sigma[n+1..])$$

where $\sigma_{out} = \sigma_1^o; \ldots \sigma_n^o$ and $\sigma = a_1; \ldots a_n; \sigma[n+1..]$.

Our definition is slightly different from Ligatti's original definition [2] and the refined definition in [15]. Intuitively, in this paper we have just simplified the original notions from [2] by enucleating the notions of output and memory and always forced the enforcement mechanism to progress in the processing of the input. We later show that our actions are identical to the combinations of atomic actions (read symbol but no output, output symbol but don't read input) from [15] on every non-diverging computation.

We will present the definition of edit automata verbatim from [15]. Every execution of an edit automaton is specified using a labeled operational semantics. The basic single-step has the form

$$(q, \sigma) \xrightarrow{\tau} (q', \sigma')$$

where q is the current state of the automaton, σ is the sequence of actions that is in the input, q' and σ' are the state and sequence of actions after the automaton takes a step, and τ is an output sequence.

> An *edit automaton* E is a triple (Q, q_0, δ_L) defined with respect to some system with action set Σ. As with truncation automata, Q is the possibly countably infinite set of states, and q_0 is the initial state. In contrast to truncation automata, the deterministic and total transition function δ of an edit automaton has the form $\delta_L : (Q \times \Sigma) \to Q \times (\Sigma \cup \{\cdot\})$. The transition function specifies, when given a current state and input action, a new state to enter and either an action to *insert* into the output stream (without consuming the input action) or the empty sequence to indicate

that the input action should be *suppressed* (i.e., consumed from the input without being made observable).

$$\frac{\sigma = a;\sigma' \quad \delta_L(q,a) = (q',a')}{(q,\sigma) \xrightarrow{a'} (q',\sigma)} \qquad \text{(E-Ins)}$$

$$\frac{\sigma = a;\sigma' \quad \delta_L(q,a) = (q',\cdot)}{(q,\sigma) \xrightarrow{\cdot} (q',\sigma')} \qquad \text{(E-Sup)}$$

In [15] Ligatti et al argue that this single-step semantics can easily simulate multi-step semantics. In the rest of this section we will call the edit automaton defined in [15] by single step edit automaton.

This automaton allows diverging computation, a kind of computation where the edit automaton will run forever without reading any input while keeping outputting data. Formally, it means that after some ith action of some finite input, the automaton will not read any more input symbols and will only output symbols. By Q^ω we denote an infinite sequence of states from the set Q.

Definition 3 (Diverging computation) A *diverging computation* starting in the state q_1 of the single step edit automaton (Q, q_0, δ_L) and triggered by $\sigma \in \Sigma^*$ is a sequence $\langle q_1, q_2, \ldots\rangle \in Q^\omega$ such that $(q_i,\sigma) \xrightarrow{a'_i} (q_{i+1},\sigma)$ for some $a'_i \in \Sigma$ for all $i \geq 1$.

Definition 4 (Effectively diverging computation) An single step edit automaton $E = (Q, q_0, \delta_L)$ has an *effectively diverging computation* if there exists a finite sequence $\sigma \in \Sigma^*$ and there exists a finite sequence of states $\langle q_1, q_2, \ldots, q_n\rangle$ such that

1) $\sigma = \sigma_0$, and
2) for all $i \leq n$
   - either $(q_i,\sigma_i) \xrightarrow{a'_i} (q_{i+1},\sigma_i)$
   - or $\sigma_i = a;\sigma_{i+1}$ and $(q_i,\sigma_i) \xrightarrow{\cdot} (q_{i+1},\sigma_i)$ for some $a \in \Sigma$ and $\sigma_i \in \Sigma^*$, and
3) there exists a diverging computation starting in q_n and triggered by σ_n.

While it was theoretically useful in [15], the very idea that a control mechanism could possibly produce output without any input is hardly acceptable by the users[1]. In contrast, the idea that the enforcement mechanism could spend a lot of time in order to process an input and eventually report a long sequence of follow-up actions was considered impractical but understandable.

So our definition is that the only way to produce an infinite output is to get an infinite input. And differently from the original definition, our automaton consumes the input action a at every transition.

[1] We are currently carrying on a large case study on e-Health in the framework of the EU-ICT-IP-MASTER project.

Proposition 1 *For all enforcement mechanisms without diverging computations Definition 1 edit automata and single step edit automata [15] are identical.*

Proof We show how to construct an edit automaton from Definition 1 given an edit automaton from the original definition [15]. For all a, σ', q

1) if $(q, a;\sigma') \xrightarrow{\cdot} (q',\sigma')$ then $\delta(q,a) = q'$ and for all $\sigma_k : \gamma_o(q,\sigma_k,a) = \cdot$
2) if $(q, a;\sigma') \xrightarrow{a'} (q', a;\sigma')$ then one of the two following cases holds:
   (a) let $\langle q_1, \ldots q_n\rangle \in Q^*$ be the longest sequence such that $q = q_1$ and $(q_i, a;\sigma') \xrightarrow{a'_i} (q_{i+1}, a;\sigma')$ for all $0 < i < n$ and $(q_n, a;\sigma') \xrightarrow{\cdot} (q_{n+1},\sigma')$ then $\delta(q,a) = q_{n+1}$ and for all σ_k the output function is $\gamma_o(q,\sigma_k,a) = a'_1; \ldots a'_{n-1}$
   (b) let $\langle q_1, \ldots q_n\rangle \in Q^*$ be the sequence such that $q = q_1$ and $(q_i, a;\sigma') \xrightarrow{a'_i} (q_{i+1}, a;\sigma')$ for all $0 < i < n$ and then the automaton stops proceeding the input and outputting, because there are no successors from the state q_n, then $\delta(q,a) = q_n$ and for all σ_k the output function is $\gamma_o(q,\sigma_k,a) = a'_1; \ldots a'_{n-1}$
3) otherwise let $\delta(q,a) = q_\perp$ and for all σ_k the output function is $\gamma_o(q,\sigma_k,a) = \cdot$.

It is easy to show that both automata have the same I/O relation. Even for the case 2(b) the I/O relation is the same: even though the constructed automaton will proceed with the input action a, since the automaton stops executing the input, the observed behavior (outputting $a'_1; \ldots a'_{n-1}$) is still the same. The only difficult part is to show that we can never reach a state $q_\perp$. Since edit automaton effectively diverging computations this means that either $q_\perp$ is not reachable by a finite prefix (condition (2) of Definition 4) or there cannot be a diverging computation starting in q (condition (3) of Definition 4). So q_n in the construction must exist and thus $q_\perp$ is not reachable. □

In the proof of Proposition 1 we did not define the the keep function γ_k because it is internal function to the automaton: it defines a new value of the suspended sequence σ_k and it is used for a concrete construction of the automaton. And single step edit automaton definition does not have a notion similar to the keep function.

4 A new classification of edit automata

In this section we will detail the classes shown in Figure 1. All omitted proofs can be found in the appendix of this paper.

We start with a wide class of edit automata called *Late automata*. They simply output some prefix of the input. This class will be the container of other less trivial cases when the property $\widehat{P}$ will be taken into account.

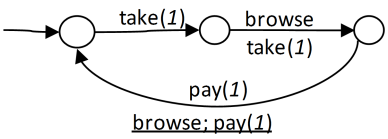


Fig. 4: Example of Late automaton.

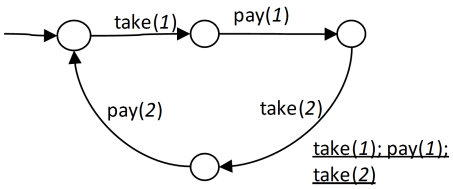


Fig. 5: Example of All-Or-Nothing automaton.

Definition 5 (Late automata) A *Late automaton A* is an edit automaton that is described by a 5-tuple of the form $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ with the restriction that it always outputs some prefix of the input:

$$\forall i\ \exists j.\ j \leq i\ \exists q_*.\ (q_0, \sigma) \overset{\sigma[..j]}{\rightsquigarrow}_A (q_*, \sigma[i+1..]) \tag{1}$$

We call property (1) *output latency* since it means that at every step of execution automaton outputs some prefix of the input.

The example of Late automaton is shown in Figure 4. This automaton simply outputs the first action of the input after reading the second and then outputs second and third actions after reading the third action.

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 some prefix of the input (hence it is a particular kind of Late automata). Moreover, on every transition they either output all suspended input actions or suppress the current action.

Definition 6 (All-Or-Nothing automata) An *All-Or-Nothing automaton A* is an edit automaton described by a 5-tuple of the form $A = \langle Q, q_0, \delta,\ \gamma_o, \gamma_k \rangle$ with the following restrictions:

- This automaton outputs a prefix of the input: (1).
- At every step of the transition either it outputs the whole suspended sequence of actions (the input symbols read by the automaton but not in the output yet) or suppresses the current action:

$$\gamma_o(q, \sigma_k, a) = \begin{cases} \sigma_k; a \\ . \end{cases} \tag{2}$$

The example of All-Or-Nothing automaton is given in Figure 5.

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 Late 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 Theorem 8 [2].

Definition 7 (Ligatti automata for property $\widehat{P}$) A *Ligatti automaton E for property* $\widehat{P}$ is an edit automaton described by a 5-tuple of the form $E = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ with the following restrictions:

- The automaton outputs a prefix of the input (1).
- Either it outputs the whole suspended sequence of actions or suppresses the current action (2).
- Output is valid at every transition (here σ' is an already output sequence)

$$\widehat{P}(\sigma'; \gamma_o(q, \sigma_k, a)) \tag{3}$$

- If in the state q the current sequence $\sigma'; \sigma_k; a$ is valid then it outputs the whole sequence:

$$\text{If } \widehat{P}(\sigma'; \sigma_k; a) \text{ then } \gamma_o(q, \sigma_k, a) = \sigma_k; a. \tag{4}$$

At every state a Ligatti automaton for property $\widehat{P}$ keeps the sequence that was read till the current moment (and consists of already output sequence σ' and kept but not yet output sequence σ_k) in order to decide whether $\widehat{P}(\sigma'; \sigma_k; a)$ holds. A possible way of implementing this is $Q = \Sigma^*$. In our definition a 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.

Let us now remind the constructive proof of Theorem 8 [2] and show that the edit automaton constructed following this proof is a Ligatti Automaton for $\widehat{P}$.

The proof of Theorem 8 in [2] constructs an edit automaton as follows:

- States: $q \in \Sigma^* \times \Sigma^* \times \{+, -\}$ [the sequence of actions seen so far, the actions seen but not emitted, and $+(-)$ is used to indicate that the automaton must not (must) suppress the current action]
- The initial state $q0 = \langle \cdot, \cdot, + \rangle$.
- Consider processing the action a in state q.
  - (A) If $q = \langle \sigma, \tau, + \rangle$ and $\neg\widehat{P}(\sigma; a)$ then suppress a and continue in state $\langle \sigma; a, \tau; a, + \rangle$.
  - (B) If $q = \langle \sigma, \tau, + \rangle$ and $\widehat{P}(\sigma; a)$ then insert $\tau; a$ and continue in state $\langle \sigma; a, \cdot, - \rangle$.
  - (C) Otherwise, $q = \langle \sigma, \tau, - \rangle$. Suppress a and continue in state $\langle \sigma; a, \cdot, + \rangle$.

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

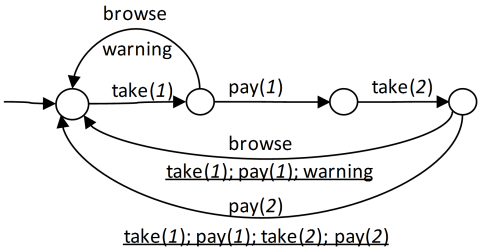


Fig. 6: Example of Late automaton for property $\widehat{P}$.

Proposition 2 also holds for the construction in the proof of Theorem 3.3 in [15].

In a nutshell, the difference between edit automata and Ligatti automata for property $\widehat{P}$ is the following:

- edit automata can suppress arbitrary actions from the input without inserting them later and can insert arbitrary actions in the output.
- 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 (in other words, it outputs the longest valid prefix of the input).

Since the automaton constructed according to the proof of Theorem 8 in [2] is a Ligatti automaton for property $\widehat{P}$ while the automaton given in [2] (Figure 2a) is not a Ligatti automaton, the difference between their behaviors is clear.

Still, the automaton of Figure 2a is not a completely arbitrary edit automaton and we propose a notion of *Late 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 8 (Late automata for property $\widehat{P}$) A *Late automaton A for* $\widehat{P}$ is an edit automaton that is described by a 5-tuple of the form $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ with the following restrictions (σ' is with the following restrictions (σ' is a sequence that is in the output already, σ_k is a sequence of an input symbols read by the automaton but not in the output yet):
If $\widehat{P}(\sigma'; \sigma_k; a)$ then

- Output is a prefix of the input (1), and
- Output is always valid (3).

The example of Late automaton for property $\widehat{P}$ is shown in Figure 6. This automaton is similar to the one given in Figure 2a with the only difference that it delays the output of the first `take-pay` transaction.

Later in Figure 8 we will pictorially describe the relations among different kinds of edit automata. 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 Relationships among the Edit Automata classes

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 principles. The notion of *soundness* requires all the observable output of an 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.

Definition 9 (Precise Enforcement) An automaton A with starting state q_0 *precisely enforces* a property $\widehat{P}$ on the system with action set Σ if and only if $\forall \sigma \in \Sigma^* \; \exists q'$ $\exists \sigma' \in \Sigma^*$ such that

1. $(q_0, \sigma) \overset{\sigma'}{\rightsquigarrow}_A (q', \cdot)$, and
2. $\widehat{P}(\sigma')$, and
3. $\widehat{P}(\sigma) \Rightarrow \forall i \; \exists q''. \, (q_0, \sigma) \overset{\sigma[..i]}{\rightsquigarrow}_A (q'', \sigma[i+1..])$

According to this 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.

There is another notion of enforcement called "effective $_=$ enforcement" [2] (with later refinement in [15]) that obeys the properties of soundness and transparency and output production (for any input there is an output).

Definition 10 (Effective$_=$Enforcement) An automaton A with starting state q_0 *effectively$_=$enforces* a property $\widehat{P}$ on the system with action set Σ if and only if $\forall \sigma \in \Sigma^* \; \exists q' \quad \exists \sigma' \in \Sigma^*$ such that

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

Then we introduce a refinement of effective$_=$enforcement, where an automaton can suppress some actions and later insert them when the sequence turns out to be legal. We name it *Late effective$_=$enforcement.*

Definition 11 (Late Effective$_=$Enforcement) An edit automaton A with starting state q_0 *lately effectively$_=$enforces* a property $\widehat{P}$ on the system with action set Σ if and only if $\forall \sigma \in \Sigma^* \; \exists q' \; \exists \sigma' \in \Sigma^*$ such that

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

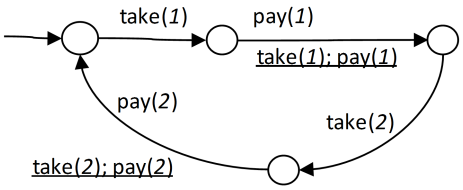


Fig. 7: Edit automaton that lately effectively$_=$enforces property $\widehat{P}$.

The definition above obeys four properties of enforcement: output production (1), soundness (2), transparency (3) and output latency, which is originally presented in equation (1).

Notice that edit automaton that lately effectively$_=$enforces a property $\widehat{P}$ will always output some valid prefix of the input. This conclusion is obvious because soundness ensures that all output is valid and output latency ensures that output is always a prefix of the input.

We show an example of edit automaton that lately effectively$_=$enforces a property $\widehat{P}$ in Figure 7.

It is easy to see from the definitions that edit automata that lately effectively$_=$enforce a property $\widehat{P}$ are a proper subset of edit automata that effectively$_=$enforce $\widehat{P}$. An example is the edit automaton in Figure 2a that effectively$_=$enforces property $\widehat{P}$. From an illegal input sequence `take(1); browse; take(2); pay(2)` it produces `warning; take(2); pay(2)` while automaton in Figure 7 that lately effectively$_=$enforces $\widehat{P}$ outputs nothing.

As it is said in [2] edit automaton from Figure 2a effectively$_=$enforces the market policy (Example 1). But since the market policy is given only in natural language and the predicate $\widehat{P}$ is not given, statements such as "An edit automata effectively enforces the market policy" are a bit stretching the definition.

In Figure 8 we summarize the relations among the different kinds of automata that we have introduced. When drawing two boxes separated by a space we mean that inclusion is probably not proper. In the rest of the paper we prove the correctness of this classification.

Proposition 3 *Late automata and Late automata for property $\widehat{P}$ are not a proper subset of each other.*

For example, for input sequence $\sigma =$ `take(1);browse` the Late automaton from Figure 4 will output `take(1)` action which is not valid, while the Late automaton for property $\widehat{P}$ from Figure 6 will output `warning` action.

From Proposition 3 we can conclude that classes of Late automata and Late automata for $\widehat{P}$ have some common subclass but none of them include the other.

Theorem 1 *Edit automata that effectively$_=$ enforce property $\widehat{P}$ are a proper subset of Late automata for $\widehat{P}$.*

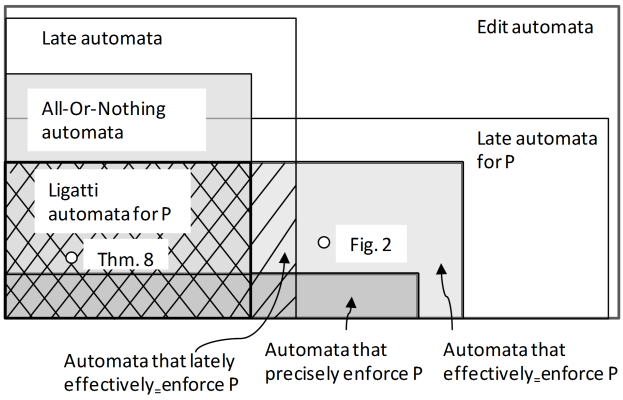


Fig. 8: The classes of edit automata.

Proof First we have to prove that if an edit automaton A effectively$_=$enforces property $\widehat{P}$, then A is a Late automaton for property $\widehat{P}$.

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 properties $\widehat{P}(\sigma')$ and $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$. If σ is valid, it outputs the whole sequence, so automaton A is a Late automaton for $\widehat{P}$ because it outputs a valid prefix (σ is a valid prefix of itself). If σ is invalid, automaton A can output an arbitrary valid sequence while Late automaton for property $\widehat{P}$ can output any arbitrary sequence (valid or invalid).

Next we have to prove that if an edit automaton A is a Late automaton for property $\widehat{P}$ then it is not necessary that A effectively$_=$enforces property $\widehat{P}$.

In case of a valid input the Late 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. □

For example, the Late automaton for $\widehat{P}$ from Figure 6 for a valid input `take(1); pay(1)` will output nothing while the automaton from Figure 2a that effectively$_=$enforces property $\widehat{P}$ will output the whole input `take(1); pay(1)`.

Proposition 4 *Edit automata that effectively$_=$ enforce property $\widehat{P}$ are not a subset of Late automata.*

For example, the automaton from Figure 2a that effectively$_=$enforces property $\widehat{P}$ for an invalid input `take(1); browse` will output the `warning` action which is not possible for a Late automaton that has to output some prefix of the input.

Theorem 2 *Edit automata that lately effectively$_=$ enforce a property $\widehat{P}$ are exactly those Late automata that effectively$_=$ enforce property $\widehat{P}$.*

Proof Similarly to the definitions of late effective$_=$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 lately effectively$_=$enforces property $\widehat{P}$ then it obeys the property of output latency (1) and hence A is a Late automaton. According to definitions 10 and 11, since A lately effectively$_=$enforces $\widehat{P}$ it also effectively$_=$enforces $\widehat{P}$.
(*Only-if direction*). If A is a Late automaton that effectively$_=$enforces property $\widehat{P}$ then it always outputs some valid prefix of the input (since it obeys soundness and output latency) and in case of valid input it outputs the whole input sequence (property of transparency). Hence, first three conditions of late effective$_=$ enforcement hold. The fourth property holds as well because A is a Late automaton. Hence A lately effectively$_=$enforces $\widehat{P}$. □

It is immediate from the definitions the following results:

Proposition 5 *Edit automata that precisely enforce property $\widehat{P}$ are a proper subset of edit automata that effectively$_=$ enforce property $\widehat{P}$.*

Proposition 6 *Edit automata that precisely enforce property $\widehat{P}$ are not a subset of Late Automata.*

Proposition 7 *All-Or-Nothing automata are a proper subset of Late automata.*

For example, the Late automaton from Figure 4 for an input sequence `take(1); browse` will output only `take(1)` action. The given Late automaton can not be an All-Or-Nothing automaton because it can output some non-empty prefix of the input sequence.

Proposition 8 *All-Or-Nothing automata are not a subset of Late automata for property $\widehat{P}$.*

For example, the All-Or-Nothing automaton shown in Figure 5 for the input sequence `take(1); pay(1); take(2)` outputs an invalid prefix of this sequence (in this case the whole sequence) while this is not possible for a Late automaton for property $\widehat{P}$.

Proposition 9 *Edit automata that lately effectively$_=$ enforces property $\widehat{P}$ and All-Or-Nothing automata are not a proper subset of each other.*

For example, the automaton given in Figure 7 that lately effectively$_=$enforces property $\widehat{P}$ for an input sequence `take(1); pay(1); take(2)` outputs the sequence `take(1); pay(1)`. The given automaton cannot be an All-Or-Nothing automaton because after `take(2)` action it outputs non-empty prefix of the suppressed sequence.

On the other hand, for the input sequence `take(1); pay(1); take(2)` the All-Or-Nothing automaton from Figure 5 outputs the whole input sequence, which is invalid. This automaton cannot be considered as automaton that lately effectively$_=$enforces property $\widehat{P}$ because it produces illegal output.

Theorem 3 *All-Or-Nothing automata that lately effectively$_=$ enforce a property $\widehat{P}$ are exactly Ligatti automata for property $\widehat{P}$.*

Proof We have to show that All-Or-Nothing automaton A lately effectively$_=$enforces a property $\widehat{P}$ if and only if A is a Ligatti automaton for property $\widehat{P}$.

(*If Direction*). If automaton A is All-Or-Nothing automaton then equations (1) and (2) hold. Since A lately effectively$_=$enforces property $\widehat{P}$ then it obeys the properties of soundness (condition 2 of Definition 11) and transparency (condition 3 of Definition 11). Notice that soundness means that the output is always valid, which is exactly what equation (3) describes; transparency means that if input is valid then output is equal to the input, which is shown in equation (4). Hence, all the conditions of Ligatti automaton for $\widehat{P}$ are satisfied.
(*Only-if direction*). If A is a Ligatti automaton for property $\widehat{P}$ then

- It obeys properties (1) and (2) hence A is an All-Or-Nothing automaton, and then according to Proposition 7 it is also a Late automaton;
- It obeys properties (3) and (4) and as we noticed, they mean soundness and transparency which correspond to the conditions 2 and 3 of Definition 10. Hence, A effectively$_=$enforces $\widehat{P}$.

Therefore since A is Late automaton that effectively$_=$enforces $\widehat{P}$ then it lately effectively$_=$enforces $\widehat{P}$ according to Theorem 2 and it is an All-Or-Nothing automaton.□

Now we will clarify which type of edit automaton is constructed following the proof of Theorem 8 in [2] for property $\widehat{P}$ and which type of edit automaton is the one in [2] (Figure 2a).

As the Proposition 2 states, the 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 Figure 2a [2] is an edit automaton that effectively$_=$enforces $\widehat{P}$: it obeys soundness (the automaton always outputs the valid sequence) and transparency (in case of valid input it always outputs all the sequence). The edit automaton given in Figure 2a [2] is not a Late automaton because it does not always output some prefix of the input (see examples 8 and 10 in Table 2).

Therefore, we can conclude that both automata from Theorem 8 [2] and from Figure 2a [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 lately effectively$_=$enforces $\widehat{P}$.

6 From the Policy to the Edit Automata

In previous sections we have shown that edit automaton constructed by the proof of Theorem 8 [2] is actually a Ligatti automaton, a specific kind of edit automaton. Moreover, this automaton has infinite number of states, which is not practical. In this section we will show how given a property $\widehat{P}$ one can build a Ligatti automaton with a finite number of states.

Till now we have presented security property as a predicate $\widehat{P}$ on all possible sequences of executions. Proposition 6.24 of [22] states that for any edit automaton A effectively$_=$enforcing property $\widehat{P}$ there exists a Büchi Automaton specifying $\widehat{P}$. The proof of this proposition assumes that the edit automaton is of a particular kind, i.e., a Ligatti Automaton for $\widehat{P}$. Indeed, the authors assume that each state of given edit automaton contains the longest valid prefix σ_A (i.e. the sequence edited by the automaton while reading) and the suffix of the input σ_k that is suppressed by the automaton after reading. Also the construction is made in such a way that all the states of new Büchi Automaton are the same as in the given edit automaton. Every time the edit automaton suppresses an action, the next state of the Büchi Automaton is considered to be non-accepting, while when the action is inserted the next state of the Büchi Automaton is considered to be accepting. In this construction an edit automaton can insert only all of those actions that were read before. Therefore, this construction can be used only for Ligatti automata for property $\widehat{P}$.

In this paper we reverse the idea of [22] and construct an edit automaton from some automaton that represents our desired security policy. In our model we assume both finite and infinite executions. Since Büchi Automaton accepts only infinite sequences, we need another notion of automaton that can represent our security policy.

Definition 12 (Policy automaton) A *Policy automaton* is a 5-tuple $\langle \Sigma, Q, q_0, \delta, F\rangle$ where Σ is finite nonempty set of security-relevant program actions, Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to Q$ is a labeled partial transition function, and $F \subseteq Q$ is a set of accepting states.

Policy automaton has no output or keep function, it has only accepting states.

Definition 13 (Run of a Policy automaton) Let $A = \langle \Sigma, Q, q_0, \delta, F\rangle$ be a policy automaton. A *run* of A on a finite (respectively infinite) sequence of actions $\sigma = a_0, a_1, a_2, \ldots$ is a sequence of states $q_{|\sigma|} = \langle q_0, q_1, q_2 \ldots\rangle$ such that $q_{i+1} = \delta(q_i, a_i)$. A *finite run is accepting* if the last state of the run is an accepting state. An *infinite run is accepting* if the automaton goes through some accepting states infinitely often.

The Policy automaton combines the acceptance conditions of Büchi Automata and finite state automata.

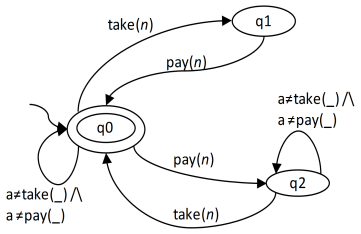


Fig. 9: Policy automaton representation of market policy (Example 1)

Definition 14 (Property represented as Policy Automaton) Some property $\widehat{P}_A$ is represented as a Policy automaton A if and only if

$$\forall \sigma \in \Sigma^\infty : \widehat{P}_A(\sigma) \iff A \text{ accepts } \sigma \tag{5}$$

Let us now define what kind of properties can be represented by a Policy automaton. In [15] the authors define the class of properties called *Renewal* properties.

Definition 15 (Renewal property) Property $\widehat{P}$ is renewal if the following holds:

$$\begin{aligned} &\forall \sigma \in \Sigma^\omega : \widehat{P}(\sigma) \iff \\ &\qquad (\forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \widehat{P}(\tau)) \end{aligned} \tag{6}$$

According to the Theorem 3.3 [15], a property $\widehat{P}$ can be effectively$_=$enforced by some edit automaton if this property is renewal, $\widehat{P}(\cdot)$ and for all finite sequences σ $\widehat{P}(\sigma)$ is decidable. Therefore we will focus on the renewal properties. The proof of Theorem 3.3 [15] is similar to the proof of Theorem 8 [2] and it is non-constructive because the number of states of the resulting edit automaton is infinite.

Theorem 4 *The set of infinite traces accepted by a Policy automaton is a renewal property.*

This is a straightforward proof that the Büchi acceptance condition is a subset of the definition of renewal properties.

In Figure 9 we present a Policy automaton for the market policy from Example 1. According to the Example 1 in English, the automaton should accept all the sequences when `take-pay` transaction is finalized. Otherwise if after `take(n)` action there is some action different from `pay(n)` then the policy is violated and the automaton halts. If after `pay(n)` action there are some other actions different from `pay` and `take` then the automaton simply waits for the `take(n)` action. In case of `take(m)` action when $m \neq n$ the automaton halts. In this way we give our own interpretation to the given example.

For given renewal property $\widehat{P}$ represented as Policy automaton we present a construction of Ligatti automaton that is the inverse of of Talhi et al. [22] construction from Proposition 6.24.

Intuitively, the resulting Ligatti automaton should have the same number of states as the corresponding Policy automaton plus an error state. At every transition that goes to non-accepting state Ligatti automaton should suppress the input action and add it to some suppressed sequence σ. When the next state of Policy automaton is accepting, Ligatti automaton should output the whole suppressed sequence. When there is no transition on some action in a Policy automaton, Ligatti automaton should have corresponding transition going to an error state. Hence, an error state should absorb all the bad input that can never become good.

If given property $\widehat{P}$ accepts at least one infinite sequence, then the policy automaton representing it has a cycle over accepting state. Then when we construct Ligatti Automaton for Policy automaton from Figure 9 according to Theorem 8 [2] instead of state $q2$ we will have infinite number of states, each of them will contain some different sequence of suppressed actions: `pay(n)`, `pay(n)`; a, `pay(n)`; a; a etc.

So far in the paper we have hardly used the keep function γ_k, and one may wonder what is its use. Using the keep function is essential to obtain a finite representation for an enforcement mechanism in presence of finitely representable policy.

In the construction algorithm from the proof of Theorem 8 [2] (see Figure 2b) every state itself includes suppressed sequence, hence there is infinite number of states. When one wants to construct Ligatti automaton, the keep function γ_k should define all the actions that are kept while input sequence is invalid. Then as soon as some next action makes the whole sequence valid (i.e., accepted by Policy automaton), the output function γ_o should output all the suppressed actions and the result of the γ_k function should be an empty sequence.

We define $\langle \delta, \gamma_o, \gamma_k \rangle$ as a function $Q \times \Sigma \to Q \times (\Sigma \cup \{*\})^* \times (\Sigma \cup \{*\})^*$. This is a restriction over the general power that we use here for readability. The semantics of '$*$' in a sequence σ over $\Sigma \cup \{*\}$ is that each occurrence of symbol '$*$' is replaced by the sequence of kept actions σ_k denoted by $[* \mapsto \sigma_k]$, hence we write $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) = q'|\sigma_1|\sigma_2$ if and only if

$$\begin{cases} \delta(q, a) = q' \\ \gamma_o(q, \sigma_k, a) = \sigma_1[* \mapsto \sigma_k] \\ \gamma_k(q, \sigma_k, a) = \sigma_2[* \mapsto \sigma_k] \end{cases}$$

In the sequel we will only use the sequence $*; a$ which means concatenate the kept sequence σ_k with action a.

Following this construction, we build a Ligatti automaton shown in Figure 10, where we use the same notation on the transitions.

Let us present a construction algorithm. For a Policy automaton $A^P = \langle \Sigma, Q^P, q_0^P, \delta^P, F^P \rangle$ the Ligatti automaton $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ is defined as follows.

– $q_0 = q_0^P$

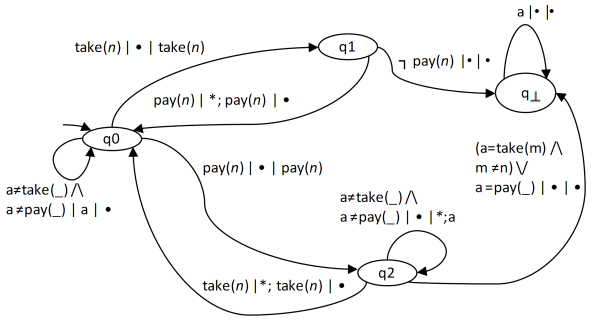


Fig. 10: Finite representation of Ligatti automaton constructed for a Policy automaton (Figure 9)

– $Q = Q^P \cup \{q_\perp\}$
– For all $q \in Q$, $a \in \Sigma$ such that $\exists q' = \delta^P(q, a)$

$$\langle \delta, \gamma_o, \gamma_k \rangle(q, a) = \begin{cases} q'|*; a|\cdot \text{ if } q' \in F^P \\ q'|\cdot|*; a \text{ otherwise} \end{cases} \tag{7}$$

– For all $q \in Q$, $a \in \Sigma$ such that $\nexists q' = \delta^P(q, a)$

$$\langle \delta, \gamma_o, \gamma_k \rangle(q, a) = q_\perp|\cdot|\cdot \tag{8}$$

If we compare Ligatti automaton from Figure 2b and Ligatti automaton from Figure 10, we will see that their output is identical for the same input. The difference is that the automaton built by a proof of Theorem 8 [2] has infinite number of states, while we provide an extended finite representation of Ligatti automaton for renewal property represented as Policy automaton.

Practically the kept sequence can be easily implemented by a queue. The Ligatti automaton has some queue that keeps all the suspended actions (this notion is similar to a very restricted form of *Queue Automaton* [6]). In our particular case the γ_o function outputs all actions in the queue (or not at all) and γ_k function only enqueues elements in the queue (when there is no output) or reset the queue to the empty one.

Theorem 5 *Any security policy represented as a Policy automaton A^P can be effectively$_=$enforced by some Ligatti automaton A.*

Proof We construct a Ligatti automaton A following the construction given above. This automaton has γ_o and γ_k functions that define the output for the transitions and sequence that will be kept after committing the transition. Automaton A effectively$_=$enforces the security policy represented as Policy automaton A^P because all conditions of effective$_=$enforcement are satisfied. Indeed, it obeys soundness because the automaton A outputs some sequence of actions σ only when the reached state is accepting (this statement is equal to $\widehat{P}(\sigma)$). Transparency holds as well because when the input sequence is valid it means that the current state is an accepting state, hence we will output the whole input sequence. □

Corollary 1 *If a Policy automaton A^P representing security policy $\widehat{P}$ is finite then the Ligatti automaton enforcing $\widehat{P}$ is finitely representable as well.*

7 Related work and Conclusions

Schneider [20] was the first to introduce the notion of enforceable security policies. The follow-up work by Hamlen et al. [11] 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 [15].

Fong [9] 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 [16] 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 μ-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 a set of finite executions, we can conclude that both automata from Theorem 8 [2] and from Figure 2a [2] are edit automata that effectively$_=$enforce property $\widehat{P}$. 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 lately effectively$_=$enforces $\widehat{P}$. A problem that is present in the construction of Theorem 8 is that it assumes an oracle that can tell for each sequence σ whether $\widehat{P}(\sigma)$ holds or not. A security policy in Theorem 8 [2] is a predicate $\widehat{P}$ on all possible finite executions, but in this case the edit automaton which effectively enforces this policy is only of theoretical interest: following the proof of Theorem 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 to Theorem 8 in the very same paper is due to a deeper theoretical difference. In order to understand this difference, we have introduced better classification of edit automata introducing the notion of *Late Automata*. The particular automata that are actually constructed according to Theorem 8 from [2] are a particular form of late 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. [22] only applies to Ligatti's automata. Given an (infinite state) 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 Figure 2a? Proposition 6.24 [22] 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 show 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 enforces your policy.

So we fully re-open the most intriguing question that the stream of papers on execution monitors seemed to have closed:

Challenge 1 *You have written your security enforcement mechanism (aka your edit automata); how do you know that it really enforces the security policy you specified?*

Our constructive proof of Theorem 5 is only a first step to address this research challenge. Given a policy specification expressed as a Policy automaton or Büchi automaton (as used in Security-by-Contract [3,17,18]) we constructed an extended finite state automaton that effectively$_=$enforces it.

There is however a much broader issue we would like to raise. Essentially all papers on security monitors cited in this article (and this paper itself) only provide a security judgment over a trace (i.e. a predicate over trace) that considers the trace as a whole. Hence, we are not able to define an incremental notion of security that tells how to fix a bad trace. The Ligatti automaton will output only the longest valid prefix, the only available theoretical fix is truncation. So we open another question:

Challenge 2 *If your enforcement mechanism really enforces your security policy, how exactly it does the enforcement? Does it fix the bad sequences in the way you want?*

Acknowledgment

The authors would like to thank the reviewers for their constructive comments and suggestions that help improve the manuscript. This work has been partly supported by the European Union under the projects EU-ICT-IP-MASTER, EU-FET-IP-SecureChange, EU-FP7-IST-NoE-NESSOS.

References

1. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, pp. 305–314. ACM Press (2005)
2. Bauer, L., Ligatti, J., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security **4**(1-2), 2–16 (2005)
3. Bielova, N., Dragoni, N., Massacci, F., Naliuka, K., Siahaan, I.: Matching in security-by-contract for mobile code. Journal of Logic and Algebraic Programming **78**(5), 340–358 (2009)
4. Bielova, N., Massacci, F.: Do you really mean what you actually enforced? In: Proceedings of the 5th International Workshop on Formal Aspects in Security and Trust, vol. 5491, pp. 287–301. Springer-Verlag Heidelberg (2008)
5. Bielova, N., Massacci, F., Micheletti, A.: Towards practical enforcement theories. In: Proceedings of The 14th Nordic Conference on Secure IT Systems, *Lecture Notes in Computer Science*, vol. 5838, pp. 239–254. Springer-Verlag Heidelberg (2009)
6. Cherubini, A., Citrini, C., Reghizzi, S.C., Mandrioli, D.: QRT FIFO automata, breadth-first grammars and their relations. Theoretical Computer Science **85**(1), 171–203 (1991)
7. CNET Networks: Channel 4's 4od: Tv on demand, at a price. Crave Webzine (2007)
8. Erlingsson, U.: The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University (2003)
9. Fong, P.: Access control by tracking shallow execution history. Proceedings of the 2004 IEEE Symposium on Security and Privacy pp. 43–55 (2004)
10. Gong, L., Ellison, G.: Inside Java(TM) 2 Platform Security: Architecture, API Design, and Implementation. Pearson Education (2003)
11. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Transactions on Programming Languages and Systems **28**(1), 175–205 (2006)
12. Havelund, K., Rosu, G.: Efficient monitoring of safety properties. International Journal on Software Tools for Technol. Transfer (2004)
13. Krukow, K., Nielsen, M., Sassone, V.: 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)
14. LaMacchia, B., Lange, S.: .NET Framework security. Addison Wesley (2002)
15. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Transactions on Information and System Security **12**(3), 1–41 (2009)
16. Martinelli, F., Matteucci, I.: Through modeling to synthesis of security automata. In: Proceedings of the Second International Workshop on Security and Trust Management, *Electronic Notes in Theoretical Computer Science*, vol. 179, pp. 31–46. Elsevier Science Publishers B.V. (2007)
17. Massacci, F., Siahaan., I.: Matching midlet's security claims with a platform security policy using automata modulo theory. In: Proceedings of The 12th Nordic Workshop on Secure IT Systems (NordSec'07) (2007)
18. Massacci, F., Siahaan, I.S.R.: Simulating midlet's security claims with automata modulo theory. In: Proceedings of the 2008 workshop on Programming Language and analysis for security, pp. 1–9. ACM Press (2008)
19. Ray, B.: Symbian signing is no protection from spyware. `http://www.theregister.co.uk/2007/05/23/symbian_signed_spyware/` (2007)
20. Schneider, F.: Enforceable security policies. ACM Transactions on Information and System Security **3**(1), 30–50 (2000)
21. Sekar, R., Venkatakrishnan, V., Basu, S., Bhatkar, S., DuVarney, D.: Model-carrying code: a practical approach for safe execution of untrusted applications. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 15–28. ACM Press (2003)
22. Talhi, C., Tawbi, N., Debbabi, M.: Execution monitoring enforcement under memory-limitation constraints. Information and Computation **206**(2-4), 158–184 (2007)

Appendix

Let us prove the the propositions.

Proposition 2 *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_o is the output so far, σ_k is a suppressed sequence of actions. Let us have a look at two main steps of the construction:

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

Since at every step the output is empty or $\sigma_k; a$ then the automaton obeys the property (2); it always outputs prefix of the input, hence statement (1) holds as well. Constructed automaton outputs the sequence only if it is valid, hence statement (3) holds. It outputs all the suppressed actions if the sequence becomes valid, therefore statement (4) 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 $\widehat{P}$. □

Proposition 3 *Late automata and Late automata for property $\widehat{P}$ are not a proper subset of each other.*

Proof First we have to prove that if an edit automaton A is a Late automaton, then it is not necessary that A is a Late automaton for property $\widehat{P}$.

By σ we denote an input sequence of the automaton and σ' is an output sequence. A Late automaton A obeys only one property: it always outputs some prefix of the input (1). Hence, even if the overall input sequence σ is valid, A can output an invalid prefix of the input ($\neg\widehat{P}(\sigma')$), while Late automaton for property $\widehat{P}$ will always output a valid sequence (3).

Next we have to prove that if edit automaton A is a Late automaton for property $\widehat{P}$ then it is not necessary that A is a Late automaton. In case of invalid input sequence the Late automaton for property $\widehat{P}$ can output another sequence which is not necessarily a prefix of the input, while a Late automaton will always output a prefix of the input (1). □

Proposition 4 *Edit automata that effectively$_=$ enforce property $\widehat{P}$ are not a subset of Late automata.*

Proof We have to prove that if an edit automaton A effectively$_=$enforces property $\widehat{P}$ then it is not necessary that A is a Late automaton.

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 properties: $\widehat{P}(\sigma')$ and $\widehat{P}(\sigma) \Rightarrow \sigma = \sigma'$. In case of invalid input, the automaton A will output some valid sequence (according to soundness), which is not necessary a prefix of the input. Therefore it is not necessarily a Late automaton. □

Proposition 5 *Edit automata that precisely enforce property $\widehat{P}$ are a proper subset of edit automata that effectively$_=$ enforce property $\widehat{P}$.*

Proof If an edit automaton precisely enforce property $\widehat{P}$ then for the valid input sequence it will output in a lock-step mode the whole sequence, hence the property of transparency holds. All the other properties of effective$_=$enforcement hold as well.

If an edit automaton effectively$_=$enforces property $\widehat{P}$ then it is not necessary that it precisely enforces $\widehat{P}$. For an invalid input σ that has a valid prefix $\sigma' \preceq \sigma$, it can output some valid sequence which does not necessarily have a prefix σ'.

Proposition 6 *Edit automata that precisely enforce property $\widehat{P}$ are not a subset of Late Automata.*

Proof The proof is straightforward: for an illegal input edit automaton that precisely enforce property $\widehat{P}$ can output a sequence that is not necessarily a prefix of the input, however a Late Automaton always outputs some prefix of the input.

Proposition 7 *All-Or-Nothing automata are a proper subset of Late automata.*

Proof First we show that if A is All-Or-Nothing automaton then A is a Late automaton. Since (1) holds for All-Or-Nothing automaton A, then A is a Late automaton.

Next we show that if A^* is a Late automaton then it is not necessary that A^* is an All-Or-Nothing automaton. A^* can output some prefix of the input that can be some prefix of all suppressed actions. In this case A^* is not an All-Or-Nothing automaton because 2 does not hold. □

Proposition 8 *All-Or-Nothing automata are not a subset of Late automata for property $\widehat{P}$.*

Proof We show that if A is an All-Or-Nothing automaton, then it is not necessary that A is a Late automaton for property $\widehat{P}$. An All-Or-Nothing automaton A can output some invalid prefix of the valid input which is not possible for a Late automaton for $\widehat{P}$. □

Proposition 9 *Edit automata that lately effectively$_=$ enforces property $\widehat{P}$ and All-Or-Nothing automata are not a proper subset of each other.*

Proof First we have to prove that if an edit automaton A lately effectively$_=$enforces property $\widehat{P}$ then it is not necessary that A is an All-Or-Nothing automaton.

By σ we denote an input sequence of the automaton and σ' is an output sequence. Automaton A can output some valid prefix of the input which is not necessarily all

the suppressed actions, hence A is not All-Or-Nothing automaton.

Next we have to prove that if edit automaton A^* is an All-Or-Nothing automaton then it is not necessary that A^* lately effectively$_=$enforces $\widehat{P}$. At some step A^* can output some invalid prefix of the input while automaton that lately effectively$_=$enforces $\widehat{P}$ always outputs only valid prefix of the input. □

Theorem 4 *The set of infinite traces accepted by a Policy automaton is a renewal property.*

Proof Let us prove the theorem by contradiction. Suppose that there exists a string $\sigma \in \Sigma^\omega$ such that Policy automaton A accepts σ but σ does not satisfy equation (6).

Then there exists a sequence σ', $\sigma' \preceq \sigma$ such that $\forall \tau. \tau \preceq \sigma.\sigma' \preceq \tau.\neg\widehat{P}(\tau)$. In this case there exists a run $s = \langle s_0, s_1, \ldots, s_d, \ldots\rangle$ for a sequence of actions $\sigma = \langle a_1, \ldots, a_d, \ldots\rangle$ such that s_d is *not* an accepting state. Since σ is accepted by A there must be a successor state of s_d that is accepting (otherwise s would have only finitely many accepting states), i.e., a subsequence of $s = \langle s_0, s_1, \ldots, s_d, \ldots, s_l\rangle$ such that at least s_l is accepting then the corresponding sequence of actions $\tau_l = \langle a_1, \ldots, a_d, \ldots, a_l\rangle$ is such that $\sigma' \preceq \tau_l \preceq \sigma \wedge \widehat{P}(\tau_l)$ which is a contradiction. □