Iterative Enforcement by Suppression: Towards Practical Enforcement Theories*

Nataliia Bielova
University of Trento, Italy
bielova@disi.unitn.it +39 0461 883916 (Corresponding author)

Fabio Massacci
University of Trento, Italy
Fabio.Massacci@unitn.it +39 0461 882086

Abstract

Runtime enforcement is a common mechanism for ensuring that program executions adhere to constraints specified by a security policy. It is based on two simple ideas: the enforcement mechanism should leave good executions without changes (transparency) and make sure that the bad ones got amended (soundness). From the theory side, a number of papers (Hamlen et al., Ligatti et al., Talhi et al.) provide the precise characterization of good executions that can be captured by a security policy and thus enforced by mechanisms like security automata or edit automata.

Unfortunately, transparency and soundness do not distinguish what happens when an execution is actually bad (the practical case). They only tell that the outcome of enforcement mechanism should be "good" but not how far the bad execution should be changed. So we cannot

*A preliminary, much shorter version of this paper appears in the proceedings of NordSec'09 [6].

formally distinguish between an enforcement mechanism that makes a small change and one that drops the whole execution.

In this paper we explore a set of policies called iterative properties that revises the notion of good executions in terms of repeated iterations. We propose an enforcement mechanism that can deal with bad executions (and not only the good ones) in a more predictable way by eliminating bad iterations.

Keywords: runtime enforcement, execution monitors, edit automata

1 Introduction

The last few years have seen a renewed interest in the theoretical and practical aspects of the runtime security enforcement mechanisms. These mechanisms dynamically monitor the behavior of applications and immediately take action when the application behaves in a way that violates security policy.

The first formal model of such monitors was defined by Schneider [23]. He presented a *security automata* that recognizes correct (allowed by the policy) run of the application and halts it as soon as its behavior violates the policy. These monitors are provably enforcing a restricting class of security policies called *safety* properties. These properties specify that "nothing bad ever happens".

Later a number of refinements have been proposed, for example Hamlen's work on rewriting [15] and Ligatti et al. works on edit automata [3, 19]. The later work proposes a model of monitors that are not only recognizing the correct behavior of applications, but are also capable of transforming their behavior. This power gives edit automata capability of enforcing more than safety properties, and it was proven that they are able to enforce a richer class of properties, called *renewal* properties.

In our paper [4] and corresponding technical report [5] we analyzed edit au-

tomata and how exactly they can enforce renewal properties. We have proposed a hierarchy of edit automata and found out that these monitors can potentially transform insecure behavior of applications in very different ways, or in other words, enforce the same property differently. For example, we have taken a security property described in the example of the original Ligatti et al. paper [3] and found out that the edit automaton in their running example enforces this property differently from the one constructed by the proof of their theorem. The latter automaton, that we call *Longest-valid-prefix* automaton[1], is a special kind of edit automata that waits until the behavior becomes correct again by itself and only then outputs it. For more details see [4, 5].

We do not think this is due to a mistake in the original papers but rather in a limitation of the notions of soundness and transparency. They are used in most of the papers to characterize good behavior that can be potentially enforced with particular enforcement mechanism. Soundness says that application behavior should always respect the security policy. Transparency says that if the behavior is correct, then it should not be transformed by an enforcement mechanism.

However these notions are not enough to define how an enforcement mechanism actually changes the bad behaviour. In practice the bad behavior of application is usually changed in *some* way such that the result is correct (soundness). This can be done by halting an execution of application (Schneider's security automata [23]), by waiting till it becomes correct again (Longest-valid-prefix automata [3, 4]), by removing wrong subparts (Iterative suppression automata [6]) or by transforming it in some more complex way (Full edit automata [3, 19]). Yet, this part is simply not reflected in the current theories.

[1] In the previous papers [4, 6] we called it *Ligatti* automaton, but changed the naming to be fair to Ligatti's coauthors.

1.1 Contribution of the paper.

In order to close this gap, in this paper we address the following challenge:

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

Our solution is a notion of *"better" enforcement* that makes it possible to overcome the distinction between bad behavior and good behavior and a mechanism that performs concrete enforcement of special kind of security properties.

Informally speaking, our notion of "better" is based on the number of elements from the original execution that should be suppressed in order to get a legal execution. If the resulting execution cannot be obtained from the original one by suppressing elements, we say that the distance is equal to ∞. While still preliminary this notion allows us to show that when transforming bad executions, an iterative suppression automaton is always "better" (i.e. deletes less elements) than a Longest-valid-prefix automaton while both of them are sound and transparent.

We start by presenting our running example that will be used throughout the paper in Section 2. This example of a health-care process of drug dispensation to outpatients has inspired us to do this work. We find it particularly interesting to show what kind of practical enforcement can be done whenever process execution violates the security policy. In Section 3 we present the basic notations for security properties, runtime enforcement, and others. We introduce a new kind of security property called iterative property. Loosely speaking, it captures the practical intuition of repeating executions of the workflow. This property corresponds more accurately than renewal and safety property [19] to the actual behavior of workflow executions that are used in practice. In our practical example it covers all properties of interests with the exception of liveness. We also show the relation between iterative properties and all other

classical security properties. Then we present formal definitions of enforcement mechanism in Section 4 and traditional principles of evaluating the enforcement. Algorithms for constructing a Longest-valid-prefix automaton and a new enforcement mechanism that suppresses the bad iterations are given in Section 5. Section 6 proposes a comparison of the Longest-valid-prefix automaton and our mechanism, it also presents a theorem showing that in practical cases (when the behavior is bad) our mechanism is "better" (producing longer executions) than the Longest-valid-prefix automaton. Finally we describe related work, propose more discussions and conclude in Section 7.

2 Running example

We propose a case study based on a healthcare process of drug dispensation. In Italy hospitals accredited with the Public National Health Service are in charge of administering drugs and providing diagnostic services to patients. Usually in these (public or private) hospitals there is a generic *dispensation process description* that allows hospitals to refund the drugs administered and/or supplied in the hospitals' outpatient departments to the patients that are not hospitalized. In particular, there is a process called *File F* that allows refunding of the drugs for specific critical and crhonic diseases that should be done by the public authority. As another example, if the patient is using a specific drug for the research program purposes (i.e. the patient has been enrolled for the clinical trial for the testing of that drug) then the reimbursement should be done by the clinical trial funds.

Drug dispensation process is a high level business process. It involves human participants as well as IT technologies. Some of contained tasks are completely human activities without any interaction with IT system (e.g. all patients tasks, or delivering drugs from stock to patient (physically) by doctor or nurse). The

drug dispensation process is modeled in such a way that if all the activities correspond to this process, then they also comply with the *dispensation process description* and hence all the dispensed drugs will be refund.

The drug dispensation process starts when the Patient brings his prescription sheet to Doctor or Nurse (we will say Doctor from here on). The Doctor authenticates himself by entering his ID. The system identifies Doctor's operational unit and enables Patient identification. Then the Doctor identifies the Patient and asks him whether he requires anonymization of his health records. The Doctor anonymizes it if required. The system retrieves all necessary data for selecting the drugs to dispensation and offers the option to select drugs to the Doctor. Drug selection is a special subprocess that we will describe in details later. After drug selection Doctor verifies candidate drug list (for dispensation) and continues or restarts the process of drug selection. Then Doctor registers the drug request, takes the drugs physically from the stock, prints dispensation sheet, brings drugs to the Patient and archives copy of dispensation sheet, signed by Patient. More details of the process can be found at [20].

Here we are focusing on the drug selection subprocess. Execution of this subprocess is repeated by the Doctor for every drug in the prescription. First, the Doctor fills in the candidate drug list for his Patient. If the drug is highly sensitive, reviewing therapeutical notes is needed. In this case they will be shown to the Doctor. Then the system checks drug's submission to Research program and in case the drug is registered shows the notification to the Doctor. In case Doctor receives such notification, he should insert the research protocol number, a number of the protocol according to which the drug can be given to the Patient. Then the Doctor inserts all prescription details, the system checks drug availability in stock and eventually notifies the Doctor. If the drug is not available in stock Doctor checks the physical existence in the ward and then

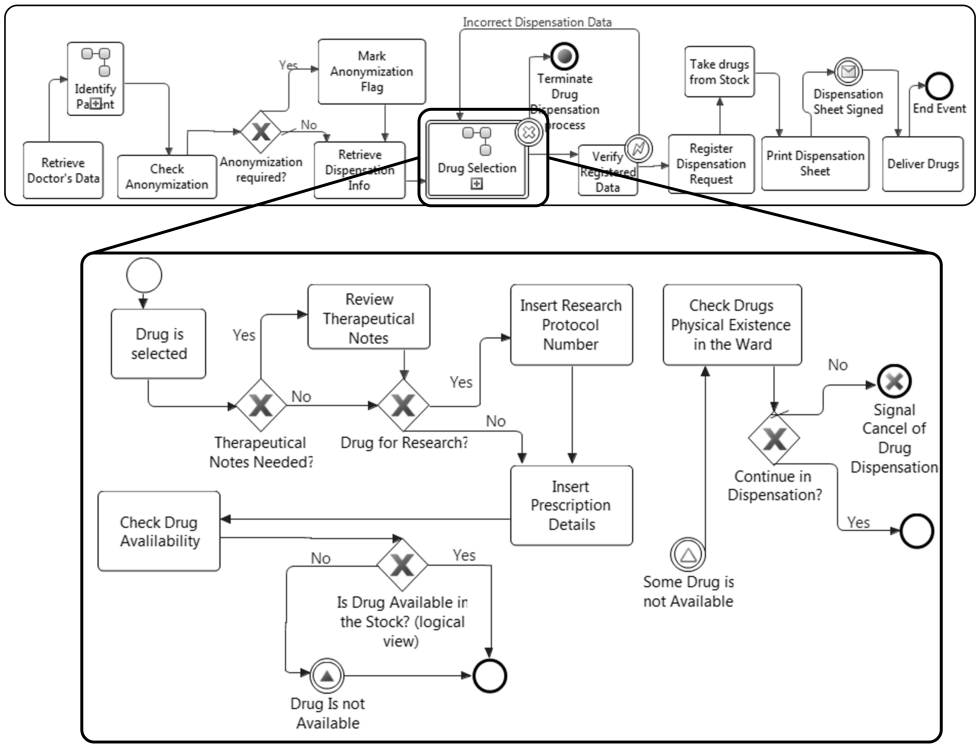


Fig. 1: BPMN diagram drug dispensation process

decides to continue or cancel the dispensation process.

In Fig. 1 we present a simplified version of the BPMN (Business Process Model Notation) diagram of drug dispensation process emphasizing the drug selection subprocess[2]. We show the whole drug selection process (the top part of the figure) just to give reader a flavor of the process we described above. As a running example of this paper we will use the drug selection subprocess (lower part of the figure).

To ease the comparison with other papers on runtime enforcement [3, 24] we decided to present the BPMN process using finite state automaton. This

[2]We would like to thank ANECT (`http://www.anect.com/en/`) for developing original BPMN diagrams of the full drug dispensation process.

representation is natural, since from a point of view of the Doctor a workflow described above is a sequence of actions that he should perform. To simplify the translation we assume that each choice in BPMN diagram corresponds to the action in resulting process execution that communicates the choice, e.g., if the drug is for research then the corresponding action is "Drug is for research", if the drug is highly sensitive, then an action "Therapeutical notes needed" will be shown. Not only in our example but in many practical cases the policy is given implicitly by describing the workflow corresponding to the legal executions, that can be repeated several times. This is precisely the case when we represent a process description using automaton. Formal definition and a corresponding automaton will be given later in the Section 3.

In practice there are many processes where a user should repeat several operations repeatedly choosing among several options. The Example 1 of drug dispensation is a good demonstration for this kind of process, so proposing a mechanism of enforcing this property covers all the other examples of this kind.

Example 1. *Let us assume the following execution of the process, where 3 different drugs are in the prescription list, hence the execution will consist of 3 parts, that we call an iteration, for each drug:*

1. *The first drug is selected and it is not highly sensitive, so therapeutical notes are not needed; the drug is for research, so the Doctor inserts research protocol number; then he inserts prescription details; and the drug is available in stock. This iteration is correct, in a sense that it corresponds to the process description.*

2. *Then for the second drug: the Doctor selects it; the drug is not higly sensitive so therapeutical notes not needed; the drug is for research but the Doctor inserts prescription details only; the drug is available in stock. In this part of the process execution the drug is for research but the research*

protocol number is not inserted, therefore it is not correct because it violates the process description.

3. *The third part of the execution consists of actions: the Doctor selects a highly sensitive drug, so therapeutical notes needed; the Doctor reviews therapeutical notes; the drug is not for research; Doctor inserts prescription details, and the drug is available in stock. This part is correct.*

♢

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

3 Basic notions of Security properties

The runtime enforcement mechanisms monitor and intercept actions while running the application. In the papers on runtime enforcement theory for untrusted applications [3, 7, 14, 23, 19] the actions get intercepted before they are executed by the target system. Usually the actions are initiated by the application. In this paper we propose a slightly different position: intercepted actions can be initiated either by a user or by an application itself.

And still our theory applies also to application monitors. The user is interacting with an application, by doing so he is executing a process and is issuing sequences of actions. Each action is initiated either by a user of by an application. A runtime monitor is a mechanism that is intercepting the actions of the application before they execute on the system, so that only correct sequences of actions get through the monitor.

3.1 Security properties

Following the standard notation on runtime security policies [3, 14, 23] we denote the set of observable actions by Σ. A *tentative execution*, or a trace, is a finite or infinite sequence of actions; the set of all finite sequences over Σ is denoted by Σ^*, the set of infinite sequences is Σ^ω, and the set of all sequences (finite and infinite) is Σ^∞. Executions are denoted by σ and actions are denoted by a possibly with subscript or superscript. With $\cdot$ we denote an empty execution. 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 write $\tau;\sigma$ to denote concatenation of two sequences and τ must be finite. By $\tau \preceq \sigma$, or $\sigma \succeq \tau$ we denote that τ is a finite prefix of finite sequence σ. Given some σ we write $\forall \tau \preceq \sigma$ as an abbreviation for $\forall \tau \in \Sigma^*.\, \tau \preceq \sigma$ and $\exists \tau \preceq \sigma$ for $\exists \tau \in \Sigma^*.\, \tau \preceq \sigma$. Similarly, for some τ we write $\forall \sigma \succeq \tau$ as an abbreviation for $\forall \sigma \in \Sigma^\infty.\, \sigma \succeq \tau$ and $\exists \sigma \succeq \tau$ for $\exists \sigma \in \Sigma^\infty.\, \sigma \succeq \tau$.

A *security property* is a predicate $\widehat{P}$ over traces or, equivalently, a *security policy* P is a set of traces $P \subseteq \Sigma^\infty$ such that $\sigma \in P \Leftrightarrow \widehat{P}(\sigma)$. Hence, we will use interchangeably the notation $\widehat{P}(\sigma)$ or $\sigma \in P$.

Schneider only considered infinite traces (by extending finite traces repeating the last action) but we prefer to distinguish finite and infinite traces. In the sequel the execution σ that satisfies the property $\widehat{P}$ is called *legal* (or good), and the execution that does not satisfy the property is called *illegal* (or bad).

There are several classes of properties. The property that defines behavior as "nothing bad ever happens" is called *safety* property [1, 17]. No prefix of a legal trace can violate this property, or equivalently, all the extensions of an illegal trace violate the property. More intuitively, safety means that as soon as something bad happens, this is irremediable: if the trace became illegal, it can

never become legal again. Formally,

$$\forall \sigma \in \Sigma^{\infty} : (\neg \widehat{P}(\sigma) \Rightarrow \exists \sigma' \preceq \sigma : \forall \tau \succeq \sigma' : \neg \widehat{P}(\tau)) \tag{1}$$

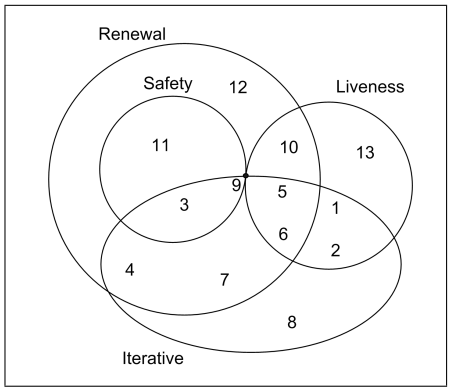
Additional to safety properties, there are *liveness* properties [1] that claim that "something good eventually happens during any execution" or , every illegal trace of finite length is not irremediable. In the other words any finite execution can always be extended to satisfy the property:

$$\forall \sigma \in \Sigma^{*} : \exists \tau \succeq \sigma : \widehat{P}(\tau)) \tag{2}$$

However, except for safety and liveness there are more general properties, which allow executions to alternate between satisfying and violating security property. *Renewal* property presented in [19] is such a property. Its definition states that every infinite-length legal sequence has an infinite number of legal finite prefixes and every illegal sequence has only a finite number of legal prefixes. The original definition of renewal property from [19] is as follows:

$$\forall \sigma \in \Sigma^{\omega} : \widehat{P}(\sigma) \Longleftrightarrow (\forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \widehat{P}(\tau)) \tag{3}$$

It was proved in [19] that every decidable renewal property can be enforced by a kind of edit automaton that outputs the longest legal prefix of the input. The renewal property, similar to the liveness property, implicitly assumes that if an infinite sequence is legal then for every prefix of this sequence the liveness holds, or "nothing irremediably bad happens in any finite prefix". It is obviously implied by the fact that an infinite-length legal execution must have an infinite number of legal prefixes. Therefore if an infinite-length execution has something irremediably bad happened in a finite prefix, then the number of valid prefixes



Legend

1 Nontermination
2 Resource availability
3 Stack inspection
4 Log out and never open files
5 Property 4 on system without file-open actions
6 Eventually audits
7 Transaction property
8 Termination + file access control
9 Trivial
10 Increasingly longer sequences
11 At most 100 SMS messages
12 At most 100 SMS and eventually audit
13 Every k transactions send a report

Fig. 2: Relationships between security properties

is finite, and hence this execution is invalid.

Let us come back to the Example 1 and define which kind of security property it corresponds to. The execution is legal if it consists of iterations that are compliant with the description of the drug selection subprocess. We generalize this class as *iterative properties* (assuming that empty trace is always valid).

Definition 1. *Property* $\widehat{P}$ *is an* iterative property *iff*

$$\forall \sigma \in \Sigma^* : \forall\, \sigma' \in \Sigma^\infty : \widehat{P}(\sigma) \wedge \widehat{P}(\sigma') \Longrightarrow \widehat{P}(\sigma;\sigma') \tag{4}$$

Fig. 2 (extending Fig. 1 in [19]) represents the relationship between safety, liveness, renewal properties and iterative properties from the point of view of good executions.

Iterative properties. The property 2 originally described in [19] is a liveness and iterative property. Assume that the system opens some resource i with action o_i and closes it with action c_i. The property 2 claims that all the opened resources must be eventually closed. This property is liveness property because any illegal sequence can be made legal by adding all the necessary clos-

ing actions. However, it is not renewal because for valid infinite sequences like $o_1; o_2; c_1; \ldots o_i; c_{i-1}; \cdots$ there is not an infinite number of finite prefixes. This property is however iterative, because for any two sequences (where the first one should be finite) that satisfy this property, their concatenation also satisfies it.

The property of stack inspection (property 3 in the figure) is safety and also iterative property. Renewal property 4 "Log out an never open files" represents the following property. Assume the system has the following actions: a_3 ranges over actions for opening files, a_2 over actions for logging out and a_1 over all other actions. The policy says that the user must eventually log out and never open files. So, this property can be written as $(a_1^*; a_2)^\infty$. It is not a safety property because there exists an illegal sequence of only a_1 actions that can be extended to a legal one by adding a_2. It is not a liveness property, because there is an illegal sequence containing a_3 that can never be extended to a legal one. However, it is a renewal property and also an iterative property because every couple of legal sequences produce a legal sequence. The same property on a system without action a_3 (property 5 in the figure) becomes a liveness property since any sequence of a_1 can become legal by adding a_2. It is also renewal and iterative.

Another property that is liveness, renewal and iterative is property 6. This property $\widehat{P}$ specifies that an execution is good if eventually an audit is performed which corresponds to an action a in the trace. It is obviously a liveness and not a safety property. It is also a renewal property because an infinite-length valid execution must have infinitely many prefixes in which a appears. This property is also iterative, because by concatenating two sequences in which a eventually appears, we get the sequence that satisfies $\widehat{P}$.

The transaction property is also liveness, renewal and iterative. Let τ range over finite sequences of single, legal transactions. A transaction policy is τ^∞

and a legal execution is the one containing any number of valid transactions.

The property 8, which was defined as non-renewal and non-liveness in [19], is an iterative property. If we concatenate two legal executions that are terminated and never access private files then the resulting execution will also be legal. The trivial property that considers all executions legal is iterative as well.

The property of nontermination (number 1 in the figure) is a liveness property since it holds for all infinite-length traces and it is not a safety property because all the finite prefixes of the good traces always have good continuations. However, it is an iterative property because no finite-length traces are valid.

Non-iterative properties. The property "Increasingly longer sequences" states that the sequence is legal iff it is infinite or its length belongs to the following set of numbers $\{F_i\}$: $F_0 = 1$, $F_{i+1} = 2F_i + 1$. Every illegal finite sequence can be prolonged such that its length will belong to the defined set of numbers, so this is a liveness property. It is also renewal because every legal infinite-length sequence has an infinite number of legal prefixes. However, this property is not iterative: by concatenating two legal sequences a new illegal sequence is always obtained.

The property 11 is "at most 100 SMS messages per application run can be sent by a mobile device." This property is useful in practice when the use of communication resources has to be bounded. This property is non-iterative. It is a safety property – if the sequence is illegal (the application sends more than 100 messages) then there is exists a prefix such that any continuation of this prefix is an illegal sequence.

Property 12 is a combination of a safety property (11) and a liveness property (6). It states that at most 100 SMS messages can be sent during an application run and eventually a particular audit action has to be done (for example making a backup of the application state). This property is not a safety property because

of the eventual audit action, it is also not a liveness one since if the application sent more than 100 SMS, the trace can never become valid again. It is also not an iterative property because a concatenation of two runs that sent 100 SMS each does not produce a valid trace. But this property is a renewal one for the same reason why properties 11 and 6 are renewal.

There is also a liveness property 13 that is not iterative. Assume a repeating process (like a transactional one) where after every k transactions a report should be sent. In case the report is not sent on time, a letter with explanations should eventually be sent. This is a liveness property but it is not iterative because the property holds for 1 transaction and for $k-1$ transactions, but their concatenation is not valid because the report for the k transactions is not sent.

These properties above are paradigmatic of the distinction between iterative and non-iterative properties. Intuitively speaking, *iterative properties are properties in which the number of times a globally legal sequence is repeated does not matter*. Of course, within the sequence itself the number of times a particular action is repeated might make a difference between being legal and being illegal, but once a sequence is globally legal it can be repeated as many times as one wishes.

In many practical cases the number does not matter, or it is so large that it is convenient for practical purposes to consider it so. In our scenario, private hospitals have a superior overall limit of drugs that can be dispensed to the public, but this is such a macroscopic and game changing phenomenon that it would not make sense to check it on the individual transactions. Another example is money withdrawal: money can be withdrawn as many times as one wishes provided the account is above zero or is replenished. The practical enforcement mechanism therefore assumes an iterative property. However, also

in this case, the bank has an overall limit (if everybody withdraws...) but this macroscopic event is managed in an entirely ad-hoc way (typically asking the government to step in and convince users to stop withdrawing).

3.2 Property representation

In practice the desired behavior of an application is often described as a workflow. Similarly to the previous work [6], we represent such workflow by a Büchi automaton that can also accept finite sequences like a finite state automaton. The automaton that recognizes sequences satisfying a security property $\widehat{P}$ we call a *Policy automaton* for property $\widehat{P}$. Just to give the notation that will be used in the rest of the paper, we present the definition below.

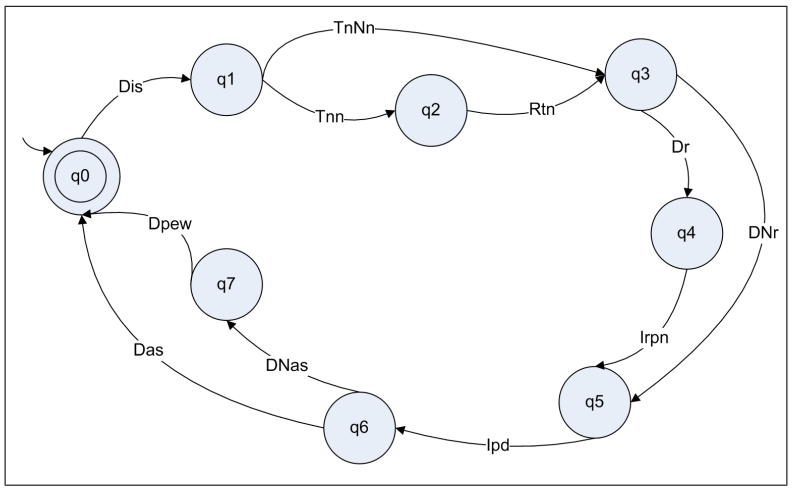
Definition 2. *A* Policy automaton *is a 5-tuple of the form* $\langle \Sigma, Q, q_0, \delta, F \rangle$*, where* Σ *is a finite nonempty set of security-relevant actions,* Q *is a finite set of states,* $q_0 \in Q$ *is the initial state,* $\delta : Q \times \Sigma \to Q$ *is a labeled partial transition function, and* $F \subseteq Q$ *is a set of accepting states.*

In Fig. 3 we present an automaton corresponding to the BPMN description of the drug selection subprocess (Fig. 1).

Definition 3 (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* $\sigma = \langle a_0, a_1, a_2, \ldots \rangle$ *is a sequence of states* $q_{|\sigma|} = \langle q_0, q_1, q_2 \ldots \rangle$ *such that* $q_{i+1} = \delta(q_i, a_i)$*. A* finite run is accepting *if the last state of the run is an accepting state. An* infinite run is accepting *if the automaton goes through some accepting states infinitely often.*

Definition 4 (Property represented as Policy automaton)**.** *Some property* $\widehat{P}_A$ *is represented as a Policy automaton* A *if and only if:*

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



Abbreviations

Dis = Drug is selected
Tnn = Therapeutical notes needed
Rtn = Review therapeutical notes
TnNn = Therapeutical notes Not needed
Dr = Drug is for research
Irpn = Insert research protocol number
DNr = Drug is Not for research
Ipd = Insert prescription details
DNas = Drug is Not available in stock
Dpew = Drug physically exists in the ward
Das = Drug is available in stock

Fig. 3: Policy automaton for a drug selection subprocess.

Notice that the set of infinite traces accepted by a Policy automaton is a renewal property. It can be easily proved because the Büchi acceptance condition is a subset of the definition of renewal properties.

In this paper we are proposing a mechanism that enforces properties represented by the Policy automaton, hence, we will enforce particular renewal properties that are also iterative.

4 Enforcement mechanism

In the following discussion we will present an enforcement mechanism as a sequence transformer $E : \Sigma^\infty \to \Sigma^\infty$. We consider a particular form of enforcement mechanism: *edit automaton* [19]. Edit automata have a power of inserting and suppressing actions from the executions. As an example, they can wait

until the illegal execution becomes legal again by suppressing its actions and then insert all the suppressed actions. They also can behave like a security automata [23] and simply suppress the suffix of the execution that makes it illegal.

We present our own definition of this automaton. Intuitively, we have just simplified the original notions by enucleating the notions of output and memory and always forced the enforcement mechanism to progress in the processing of the input. Our actions can then be shown to be identical to combinations of the atomic actions (read symbol but no output, output symbol but do not read input) from [19] on every non-diverging computation[3].

Definition 5. *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 action 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 sequence of actions kept so far and the current input action; the total function $\gamma_k : (Q \times \Sigma^* \times \Sigma) \to \Sigma^*$ defines the sequence that will be kept after committing the transition.*

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

When the automaton proceeds with one more input action, function γ_o defines the output of the automaton at this transition and function γ_k defines the memory 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 memory or ignore the input action. In general case the keep function

[3] A diverging computation is a computation where the edit automaton will run forever without reading any input while keeping outputting data. While it was theoretically useful in [19] the very idea that an enforcement mechanism could possibly produce output without any input stimulus turned out a difficult sell to our e-health partners. 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.

can perform more actions on the current memory, for example to add arbitrary actions to it.

Definition 6. *Let* $E = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$ *be an edit automaton. A* run *of automaton* E *on an* input *sequence of actions* $\sigma_{in} = a_1; a_2; \ldots$ *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* E *on input* σ_{in} *is sequence of actions* $\sigma_{out} = \sigma_1^o; \sigma_2^o; \ldots$ *such that* $\sigma_{i+1}^o = \gamma_o(q_i, \sigma_i^k, a_{i+1})$.

When an edit automaton E on an input sequence σ_{in} produces an output σ_{out}, we will write $E(\sigma_{in}) = \sigma_{out}$.

We define $\langle \delta, \gamma_o, \gamma_k \rangle$ as a function $Q \times \Sigma \to Q \times (\Sigma \cup \{*\})^* \times (\Sigma \cup \{*\})^*$. We will write $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) = q' | \sigma_o | \sigma_k$ if and only if

$$\begin{cases} \delta(q, a) = q' \\ \gamma_o(q, \sigma_S, a) = \sigma_o \\ \gamma_k(q, \sigma_S, a) = \sigma_k \end{cases}$$

In the figures we will use similar notation: every transition from state q to state q' such that $\langle \delta, \gamma_o, \gamma_k \rangle(q, a) = q' | \sigma_o | \sigma_k$ will be labeled by a triple $expr \,|\, \sigma_o \,|\, \sigma_k$, where $expr$ is an expression like $!a$ (meaning any action from Σ except for a), $a \vee b$ (input action is either a or b) or $\top$ (any action from Σ). To show that we output all the current memory followed by an input action a we will write $*; a$ in place of σ_o; while to show that we add an input action a to the current memory, we will write ADD in place of σ_k.

Basically, we will have four types of transitions in our automata:

a) $a \,|\, *; a \,|\, \cdot$ outputs the memory and empties it afterwards;

b) $a \,|\, \cdot \,|$ ADD adds a current input action a to the memory and does not output anything;

c) $a\,|\cdot|\,a$ empties the memory and then adds to the memory the current input action a;

d) $a\,|\,a\,|\cdot$ empties the memory and then outputs the current input action a;

e) $a\,|\cdot|\cdot$ empties the memory and does not output anything.

The authors of state of the art papers [3, 11, 15, 19] have noted the importance of enforcement mechanisms obeying two abstract principles, called *soundness* and *transparency*. The enforcement mechanism that enforces property $\widehat{P}$ is *sound* if all the outputs are legal according to the property:

$$\forall \sigma \in \Sigma^{\infty} : \widehat{P}(E(\sigma)) \tag{6}$$

An enforcement mechanism is *transparent* if it does not change the executions that already obey $\widehat{P}$:

$$\forall \sigma \in \Sigma^{\infty} : \widehat{P}(\sigma) \Rightarrow E(\sigma) \approx \sigma \tag{7}$$

In the original papers equal ("$\approx$") relation means that the semantics of the valid traces is not changed by an enforcement mechanism. In this paper we will use equivalence relation (which is stronger) in the definition of transparency and prove it for our approach.

However, given a property to be enforced and principles of soundness and transparency, one can construct different enforcement mechanisms that will be sound and transparent but will output different results for the same bad input traces. In the next sections we will present two different constructions of an enforcement mechanism and show that even though both results satisfy soundness and transparency, one enforces a given property "better" than the other. Now let us come back to our running example to see how the currently proposed mechanism can enforce a given property.

4.1 Enforcement of drug selection process

We present Example 1 with more details and formalizations in this section. We will use the same abbreviations of actions as in Fig. 3.

Example 2. *The drug selection process contains a set* Σ *of possible actions:*

$$\Sigma = \{\mathsf{Dis}, \mathsf{Tnn}, \mathsf{Rtn}, \mathsf{TnNn}, \mathsf{Dr}, \mathsf{Irpn}, \mathsf{DNr}, \mathsf{Ipd}, \mathsf{DNas}, \mathsf{Dpew}, \mathsf{Das}\}$$

While running the drug selection process, a tentative execution consists of 3 iterations, one per each drug.

1. *The first iteration is* Dis; TnNn; Dr; Irpn; Ipd; Das, *which is legal.*
2. *The second iteration is* Dis; TnNn; Dr; Ipd; Das, *which is illegal. It means that the drug is submitted to Research program (*Dr *action) but the research protocol number is not inserted (there is no* Irpn *action after* Dr *action).*
3. *The third iteration in* Dis; Tnn; Rtn, DNr; Ipd; Das, *which is a legal iteration.*

The resulting trace is illegal since it has an irremediably bad second part. What kind of behavior is expected from the enforcement mechanism in this case? $\diamondsuit$

Since the property described above is renewal, it can be enforced by Longest-valid-prefix automaton. Then the resulting execution will be a first iteration of the tentative execution. However, the administrators of the e-health system might expect a resulting trace to be longer. Since the drug selection process for the third drug is legal by itself, they would like to have this iteration in the resulting trace. Therefore we think that an expected behavior of the system is following: to delete the second illegal iteration of the execution, and to output the first iteration followed by the third one. However, this trace correction is not provided by existing techniques [19].

5 Construction of enforcement mechanisms

We first describe how to construct a Longest-valid-prefix automaton for a given renewal property. This construction will be used in case the property being enforced is not iterative and will also provide better explanation to the construction of our mechanism.

5.1 Longest-valid-prefix automaton

Longest-valid-prefix automaton is a specific kind of edit automaton whose I/O behavior[4] follows the proof of Theorem 8 of [3]. This automaton always outputs the longest valid prefix of the tentative execution. Since we represent the property as a Policy automaton, we know whether the sequence can ever become good again and hence can be sure whether we should halt the execution or still wait for a valid prefix to arrive from the input. Considering this we show an algorithm of Longest-valid-prefix automaton construction.

Algorithm 1 Longest-valid-prefix automaton construction

Input: Policy automaton $A^P = \langle \Sigma, Q^P, q_0^P, \delta^P, F^P \rangle$;
Output: Longest-valid-prefix automaton $A = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$;
1: $q_0 = q_0^P$;
2: $Q = Q^P \cup \{q_\perp\}$;
3: **for all** $q \in Q$, $a \in \Sigma$ **do**
4: **if** $\exists q' \in Q^P . q' = \delta^P(q, a)$ **then**
5: **if** $q' \in F^P$ **then**
6: $\langle \delta, \gamma_o, \gamma_k \rangle (q, a) = q' \,|\, *; a \,|\, \cdot$;
7: **else**
8: $\langle \delta, \gamma_o, \gamma_k \rangle (q, a) = q' \,|\, \cdot \,|\, *; a$;
9: **else**
10: $\langle \delta, \gamma_o, \gamma_k \rangle (q, a) = q_\perp \,|\, \cdot \,|\, *; a$;

The idea behind this construction is explained below. Suppose the current state of the automaton is q, the next incoming action is a. If there is a transition in the Policy automaton from the state q on an action a to an accepting state,

[4]The original construction in the cited paper yielded an infinite state automaton even if the policy was finite.

then the input read so far is accepted by the Policy Automaton. Therefore, we output all the actions read so far followed by the current action (we output $*; a$ on line 6). If the next state is non-accepting, it means that possibly there is a path to the accepting state, so the sequence read so far can become good. Therefore we simply keep the current action in the memory (we put in the memory $*; a$ and output nothing on line 8).

If there is no transition from state q on action a, it means that there is no path to some accepting state of the Policy automaton, and the sequence can never become valid again. So the next state in the automata has to be an error state: $\delta(q, a) := q_\perp$ (Longest-valid-prefix automaton will output nothing but keep all the input, this corresponds to the line 10 of the algorithm).

The behavior of the constructed Longest-valid-prefix automaton is exactly the same as of one constructed by the proof of Theorem 8 of [3]: it always outputs the longest valid prefix of the input. The only difference is that in the proof Theorem 8 the state of automaton contains all the read actions and if the trace can never become good again there will be as many states as the length of the trace. In our construction, as soon as the trace cannot become good again, the next state will be an error state and all following input actions will be kept. In the sequel an enforcement mechanism that outputs the longest valid prefix will be called E_{LP}, where LP stands for "the longest prefix".

Proposition 1. *A Longest-valid-prefix automaton constructed by Algorithm 1 for a renewal property $\widehat{P}$ represented by Policy automaton A^P is sound and transparent enforcement mechanism according to $\widehat{P}$. This automaton always outputs the longest valid prefix of the input.*

The proofs of the propositions and theorems can be found in the Appendix.

The proposed algorithm constructs an enforcement mechanism for a Policy automaton with finite number of states. For the infinite state Policy automaton

one would need to use the set construction instead of explicit algorithmic construction. In particular one can do it by replacing "for all $q \in Q, a \in \Sigma$ do " with the definition of the set of transitions as $T = \{(q' \,|\, *; a \,|\, \cdot) |\, q \in Q^P \cup \{q_\perp\} \wedge a \in \Sigma \wedge q' = \delta^P(q, a) \wedge q' \in F^P\} \cup ...$ (for every case described in the algorithm). We decided to show the algorithmic construction in this paper for the sake of readability. This remark applies to all the algorithms in this section.

5.2 Iterative suppression automaton

For iterative properties that are represented by a Policy automaton we propose a better enforcement than outputting the longest valid prefix. We call it *iterative suppression*. Since the property enforced is iterative, it describes good traces that consist of independent parts, called "iterations". The idea is that this enforcement mechanism is able to recognize when a new good iteration can start, so it can suppress all the bad actions that happen between the good iterations of a tentative execution.

Algorithm 2 is obtained from the previous construction (Algorithm 1) by changing the condition for the traces that cannot become good again, to be more precise statement on line 10 in Algorithm 1 is changed to the statements on lines 10-16 in Algorithm 2.

The states of iterative suppression automaton are composed from two states of the Policy automaton: $Q = \{(q, q_F) |\, q \in Q^P \cup \{q_\perp\}, q_F \in F^P\}$, where $q = q_F$ if $q \in F^P$, or $q \neq q_F$ if there exists a run σ such that q_F is the last accepting state before reaching q. We propose this construction of the state because an edit automaton has to "remember" the last accepting state visited during the run and compare the tentative execution to the new iterations starting only from that last visited accepting state. Condition on line 10 corresponds to the case when the next action a is not an action recognized by the Policy automaton

Algorithm 2 Iterative suppression automaton construction

Input: Policy automaton $A^P = \langle \Sigma, Q^P, q_0^P, \delta^P, F^P \rangle$;
Output: Iterative suppression automaton $E = \langle Q, q_0, \delta, \gamma_o, \gamma_k \rangle$;
1: $Q = \{(q, q_F) | q \in Q^P \cup \{q_\perp\}, q_F \in F^P\}$;
2: $q_0 = (q_0^P, q_0^P)$;
3: **for all** $(q, q_F) \in Q$, $a \in \Sigma$ **do**
4: **if** $\exists q' \in Q^P . q' = \delta^P(q, a)$ **then**
5: **if** $q' \in F^P$ **then**
6: $\langle \delta, \gamma_o, \gamma_k \rangle((q, q_F), a) = (q', q') \,|\, *; a \,|\, \cdot$;
7: **else**
8: $\langle \delta, \gamma_o, \gamma_k \rangle((q, q_F), a) = (q', q_F) \,|\, \cdot \,|\, *; a$;
9: **else**
10: **if** $\exists q'' \in Q^P . q'' = \delta^P(q_F, a)$ **then**
11: **if** $q'' \in F^P$ **then**
12: $\langle \delta, \gamma_o, \gamma_k \rangle((q, q_F), a) = (q'', q'') \,|\, a \,|\, \cdot$;
13: **else**
14: $\langle \delta, \gamma_o, \gamma_k \rangle((q, q_F), a) = (q'', q_F) \,|\, \cdot \,|\, a$;
15: **else**
16: $\langle \delta, \gamma_o, \gamma_k \rangle((q, q_F), a) = (q_\perp, q_F) \,|\, \cdot \,|\, \cdot$;

but this action can start a new iteration from the last visited accepting state q_F. Then, if only transition on (q_F, a) brings to an accepting state q'' of the Policy automaton, a is immediately output and the memory is empty; next state is (q'', q''). If next incoming action a is not accepted, then the memory is cleaned and only a is added to the memory. If a is not starting a new iteration (condition on line 15) then there is a transition to an error state and action a is not kept in the memory (line 16).

The main difference with Longest-valid-prefix automaton is that our automaton is able to recognize new good iterations and suppress only actions that caused violation of the property.

In our previous work [6] we required that all good iterations must have a *unique starting action* – an action that never repeats again in the iteration. Even if there is no unique starting action, the iterative suppression automaton still soundly and transparently enforces iterative properties. If an action a can start a new iteration and also appears in the middle of another iteration, our construction will first build the transitions that repeat the Policy automaton

(lines 5-8) and for the rest of the cases will check whether a can start a new iteration.

Even though it is not necessary to have unique starting actions, we think it is interesting to compare our mechanism for the properties with and without it. In both cases the mechanism is sound and transparent (as we prove below), but the modification of the invalid sequences will be done somewhat differently. Imagine a property where all the good traces match the pattern $(a; b; a; c)^*$ and an execution $a; b; (a; b; a; c)$. On the second input action b our mechanism will make a transition to the error state $q_\perp$ (since it is expecting c to arrive) and exit this state only on the third action a. So the output will be empty. If the pattern did not have the second a inside (for example, $a; b; d; c$) and an execution would be $a; b; (a; b; d; c)$ then our mechanism would recognize that the second a actually starts a new iteration, so the output will be $a; b; d; c$.

However, we did not found in practice the repetition of initial actions. This is somewhat obvious: different execution patterns corresponds to execution of different macro-processes in real life and thus different starting points: starting a HIV drug dispensation process or a transplant process are different and the eventual dispensation of a drug in a transplant process is intrinsically a different action for our stakeholders.

Now we prove that the proposed mechanism is sound and transparent.

Proposition 2. *An iterative suppression automaton constructed by the Algorithm 2 for an iterative property $\widehat{P}$ represented by Policy automaton A^P is sound and transparent enforcement mechanism according to $\widehat{P}$.*

6 Comparison

Let us come back to the running example and corresponding Policy automaton from Fig. 3. Following the Algorithm 1 we have constructed a Longest-valid-

prefix automaton that we now partially show in Fig. 4 (we show all transitions from the states $q_4, q_6, q_\perp$). It is easy to see that this automaton outputs the longest valid prefix: as soon as some wrong action happens (e.g. after defined that drug is for research Dr, no research protocol number is inserted !Irpn in state q_4) the automaton leads to an error state and there are no outcoming transitions from that state.

In Fig. 5 we partially show the result of the iterative suppression automaton construction for the same Policy automaton following the Algorithm 2. For an easier comparison, we also emphasize the outcoming transitions from the states q_4, q_6 and $q_\perp$. This automaton also leads to an error state as soon as something bad happens, however it is able to recognize the beginning of a new good iteration which gives this automaton the power of producing more output for the same bad input.

6.1 Distances

To compare enforcement mechanisms from the point of view of changing bad inputs we should first define how the mechanism can transform them, or in the case of suppression, how many actions it suppresses[5].

Definition 7. *The* suppressing distance *between two finite traces is a total function* $d_S : \Sigma^* \times \Sigma^* \to \mathbb{N} \cup \{\infty\}$*, such that*

$$d_S(a;\sigma,b;\sigma') = \begin{cases} \infty & \text{if } a;\sigma = \cdot \text{ and } b;\sigma' \neq \cdot \\ |a;\sigma| & \text{if } b;\sigma' = \cdot \\ d_S(\sigma,\sigma') & \text{if } a = b \\ 1 + d_S(\sigma,b;\sigma') & \text{if } a \neq b \end{cases}$$

This distance defines how many actions should be suppressed from the first sequence in order to get the second one. If the second sequence cannot be

[5]One could use an edit distance for suppressions, insertions and substitutions [2] but since we are dealing with suppressions only, we propose a notion of suppressing distance.

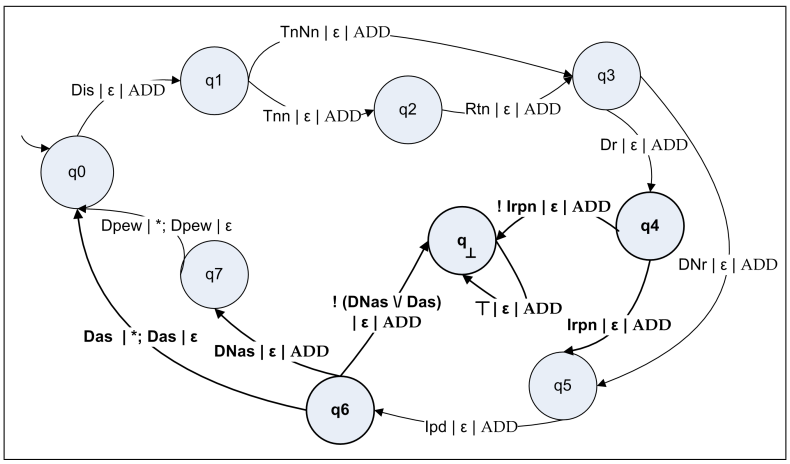


Fig. 4: Resulting Longest-valid-prefix automaton for the Policy automaton from Fig. 3

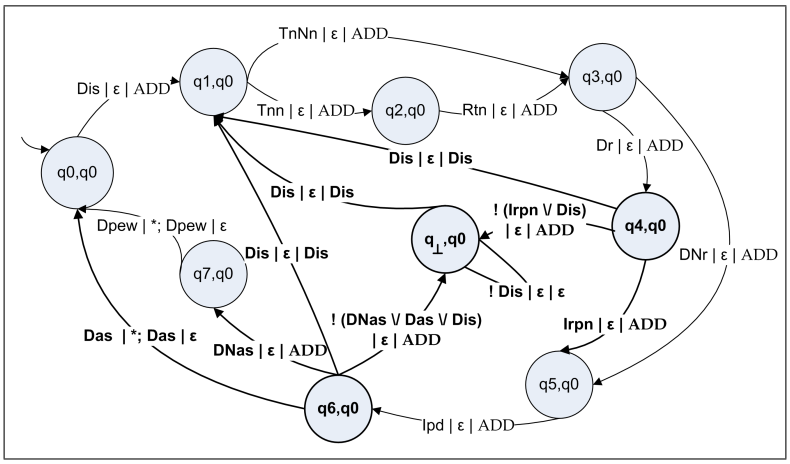


Fig. 5: Resulting Iterative suppression automaton for the Policy automaton from Fig. 3

obtained from the first one by suppressing actions, we say that the distance is equal to ∞. By using this distance we can show that when transforming bad traces, an iterative suppression automaton (Algorithm 2) is deleting less actions than the Longest-valid-prefix automaton (Algorithm 1) while both of them are sound and transparent. So if an enforcement mechanism A is sound

and transparent and produces more outputs than another sound and transparent enforcement mechanism B, then A is better than B.

Before providing a formal definition of "better" enforcement mechanism, we note that we can define that one mechanism transforms the input better than the other mechanism only when the input is *irremediable*. Moreover, even if the sequence σ contains some action a that made it irremediable, and there are no parts of σ after a that are legal, then there is no better enforcement that outputting the longest valid prefix. Hence, for comparison we will consider only irremediable sequences that have a legal substring. More formally,

Definition 8. *A finite sequence σ is* irremediable with legal substring *with respect to the property $\widehat{P}$ if and only if*

$$\neg\widehat{P}(\sigma) \wedge \exists\sigma' \preceq \sigma : (\forall\tau \succeq \sigma' : \neg\widehat{P}(\tau) \wedge$$
$$\exists\sigma_1, \sigma'', \sigma_2 \in \Sigma^* : (\sigma = \sigma_1; \sigma''; \sigma_2 \wedge \sigma_1 \succeq \sigma' \wedge \widehat{P}(\sigma'')))$$

Definition 9. *An enforcement mechanism for property $\widehat{P}$ is a* suppression enforcement mechanism *if it is sound and transparent with respect to $\widehat{P}$ and is able only to suppress actions.*

Definition 10. *Given a security property $\widehat{P}$ a suppression enforcement mechanism A is* better *than a suppression enforcement mechanism B if and only if A is producing more output than B (A suppresses less actions) for all irremediable inputs σ with legal substring:*

$$d_S(\sigma, A(\sigma)) < d_S(\sigma, B(\sigma)) \tag{8}$$

Given an iterative property $\widehat{P}$ represented as a Policy automaton we propose an automatic construction of enforcement mechanism for it in Algorithm 2. If the property is not iterative, then the Longest-valid-prefix automaton can

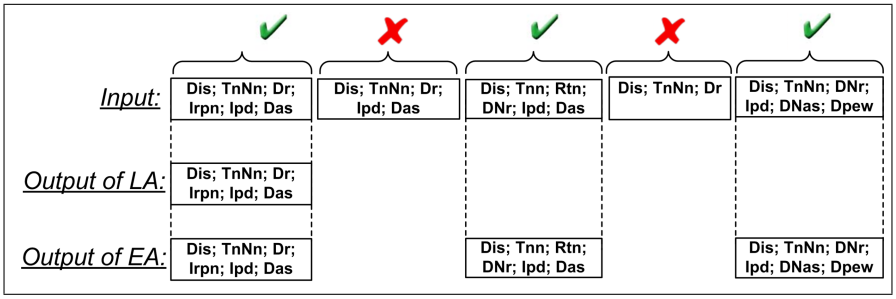


Fig. 6: Output of Longest-valid-prefix automaton (LA) and edit automaton that iteratively enforces by suppression (EA)

enforce it by outputting the longest valid prefix.

If $\widehat{P}$ is iterative, then the enforcement mechanism E_{IS} built by an Algorithm 2 is better than the enforcement mechanism E_{LP} built by an Algorithm 1.

Theorem 1. *For any iterative property $\widehat{P}$ represented by a Policy automaton A^P an edit automaton E_{IS} constructed by Algorithm 2 is* better *than a Longest-valid-prefix automaton E_{LP} constructed*

by Algorithm 1 in a sense of Definition 10.

Let us show in Fig. 6 the output of the Longest-valid-prefix automaton and iterative suppression automaton for the same policy that is represented by a Policy automaton in Fig. 3. The input contains 5 iterations corresponding to the drug selection process. The reader is already acquainted with the first 3 of them - they are the same as in the Example 2. Iterations 1, 3 and 5 are valid, and iterations 2 and 4 are invalid, hence the whole input is not valid and is irremediable. However, for iterations 1, 3 and 5 the Doctor managed to proceed successfully so there are three legal substrings: iterations 1, 3 and 5.

The Longest-valid-prefix automaton shown in Fig. 4 outputs only the first iteration. It means that Doctor will successfully complete selection process only for the first drug. The iterative suppression automaton shown in Fig. 5 will out-

put all three successful iterations. This output is an example that demonstrates the Theorem 1.

7 Related work and Conclusions

7.1 Related Work

Runtime enforcement is a general and powerful technique to enforce the security policies of the system at runtime. The first work that introduced the notion of enforceable security policies and execution monitoring was done by Schneider [23]. He proposed a mechanism based on *security automata* that monitors the execution of the target system and halts it as soon as it violates the property. In this way the proposed mechanism is able to enforce a class of safety properties [17] stating that "nothing bad ever happens".

The follow-up work by Hamlen et al. [15] 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.

Later Ligatti, Bauer, and Walker [3] have introduced *edit automata*, a new mechanism that is capable of enforcing a strictly greater class of security properties. The authors could achieve such result because differently from Schneider, that considers execution monitors as sequence recognizers, they propose to view them as sequence transformers. Edit automata can insert new actions to the execution or suppress them (with a possibility to memorize them for later use). Having this power of modifying program actions at run time, edit automata are provably more powerful than security automata and enforce a class of renewal properties [19] that is strictly bigger than the class of safety properties.

Fong [14] provided a new information-based approach to classify enforceable security properties. He proposed a new mechanism called *Shallow-History automata* that keeps as a history a set of access control events. In order to represent constraints on information available to execution monitors, he used abstraction functions over the history of monitored actions and defined a lattice on the space of all congruence relations over action sequences aimed at comparing classes of enforceable security policies. Still his policies are limited to safety properties over finite executions.

Later Talhi et al. [24] proposed *Bounded history automata* that restricts the security automata and edit automata by adding the limited history. The authors also explored the taxonomy of EM-enforceable properties depending on the size of the execution history saved by an execution monitor.

Another line of work is concerned the synthesis of runtime enforcement monitors. A great part of the work in this direction is done by Martinelli and Matteucci [21]. They have shown how to synthesize such execution monitors. 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. In a later work [22] the authors generalize the approach in the context of real-time systems.

Chabot et al. [8] proposed to synthesize security automata from the security properties expressed as Rabin automata. They provide a construction from safety properties in general case and more than safety in case when additional information about the program is obtained by static analysis. However, the comparison of enforceable properties with other (renewal, liveness) properties is not discussed in the paper.

In [12] Falcone et al. have presented algorithms of constructing edit automa-

ton (that is called "enforcement monitor" in the paper) from the given property expressed as a Street automaton. The properties being enforced are a subset of safety-progress (SP) classification of properties [10, 9]. The authors do not compare these properties with the infinite renewal properties [19], however the construction of the enforcement mechanism in the paper provides an edit automaton that enforces properties like Longest-valid-prefix automaton. Similar to our idea, enforcement mechanism in [12] has finite number of states and a separate memory where a part of an input sequence that is not yet valid is kept. In their next work [13] Falcone et al. propose an upper-bound of the set of enforceable properties. They get this bound by characterizing properties independently from the enforcement mechanisms that only should comply with soundness and transparency. However, the authors define a property to be enforceable only if each incorrect infinite sequence has a finite number of correct prefixes, and this is a definition of renewal property.

An interesting direction has been suggested by Khoury and Tawbi in a recent paper [16]. The authors proposed to define an equivalence relation between the original execution and the execution transformed by the enforcement monitor. They have shown that our approach of iterative suppressions fits into this theory. This work is interesting from a theoretical perspective, what is still missing to have a practical construction of a "better" enforcement mechanism is a precise definition of concrete equivalence relations. Once a concrete relation is defined it would be possible to run a validation with end users that the equivalence is indeed what is expected.

7.2 Discussion

A limitation of the proposed framework is that it assumes that if mechanisms A and B enforce the same policy, A is "better" than B when A suppresses fewer

actions. The new notion of "better" brings us to several threads of discussion.

7.2.1 Action insertions

The notion of "better" is defined according to the number of action suppressed and does not take into account action insertions. One could argue that suppressing many actions can be worse than inserting one, for example in a drug dispensation process suppressing the whole iteration can seem to be worse than inserting a required research protocol number.

From a general perspective this is a limitation but we think that even suppression alone has some theoretical and foremost practical advantages.

From a theoretical perspective adding insertions will significantly complicate the theory because one would have to answer such questions as "is it better to insert a missing action or wait for the next input action to arrive first?". It is not clear how to define the notion of "better" in this case. Working only with suppressions we can propose a uniquely defined mechanism for a given security policy and can show that this mechanism is better (albeit only in this limited sense) than the existing automatically constructed mechanisms (such as Longest-valid-prefix automata).

There is also a very important practical reason that is strictly linked to the case study: in our domain each action brings a liability. The same drug can be dispensed with different research protocols or within a health care chronic plan. The edit automata would not know which protocol number to insert. The choice of the number implies different costs, different billable institutions, different liability in case of adverse effects on the patient. It is up to the doctor to insert the appropriate protocol number and thus taking the liability of the decision. It is acceptable to ask the doctor to *redo very few steps* but not to have the machine doing any steps by itself. The insertion in the protocol of a machine step would require significant interactions at design stage between the

risk manager, the head of the pharmacy, the clinician involved, etc., so to decide the liability of the machine inserted action would be incorrect.

Pure suppression (of few actions) has therefore the advantage of being amenable to a correct formal treatment and being acceptable by the end-users. Intermediate solutions that keep the human liability would be the subject of future work.

7.2.2 Definition of "better"

Bauer et al. [2] in their work made a proposal for the definition of "better" enforcement mechanism. They counted the average edit distance (i.e., the minimum number of actions inserted, suppressed, or substituted) of a given mechanism and said that it is "more effective" if this measure is lower. This definition seems to be more general than a definition of "better" in our paper but it was not shown how such measure can be used in practice (no constructive algorithms were given) and what does it mean for the end users. Moreover, in this paper we propose an algorithm for constructing "better" enforcement mechanisms.

It may seem that both definitions have a serious drawback because suppressing (or inserting) a single action may be a "bigger" change to an execution than suppressing (or inserting) a large number of other actions. One can think of the following example: suppression of a single action that writes a top secret information into a file can make a "bigger" change to the execution than 100 actions to refresh pixels of the monitor. If there is a "better" mechanism, it should rather suppress 100 pixel-refresh actions than one action of writing a secret info. There can be many similar practical examples where *more* suppression makes less changes to the execution.

This example (together with possible others) points out that there might be many different notions of "better". The example above is just an example in which different actions have different weights and one could simply consider the

suppressions of actions with different weights.

We plan to investigate this kind of cases further in future work but would like to point out that in *all* research papers on run-time enforcement published so far the idea that certain actions have different weights and thus executing one of them could be a "worse" violations than just doing 100 other actions have not been considered. For example this is not even discussed in all Bauer et al. papers [2, 3, 18, 19].

Further, in our case study, the clear feedback from the stake-holders is that, in absence from a clear indication by the risk manager and a clear decision by the hospital direction to hold the risk by deviating for one action instead of another, *all actions from the process are equally important.*

7.3 Conclusions

Runtime enforcement is based on two simple ideas: the enforcement mechanism should leave good traces without changes and make sure that the bad ones got amended. From the theory side, a number of papers [15, 19, 24] provide the precise characterization of good executions that can be captured by a security policy and thus enforced by a specific mechanism. Unfortunately, those theories do not distinguish what happens when an execution is actually bad (the practical case). The theory only says that the outcome of enforcement mechanism should be "good" but not how far should the bad execution be changed.

In this paper we have proposed an enforcement mechanism called *iterative suppression automaton* that makes it possible to overcome the distinction between bad traces and good traces. By revising the notion of good traces in terms of iterations we offered a formal characterization of how enforcement mechanism can deal with the bad traces. The idea of this enforcement is to suppress all the bad actions between the legal iterations in the tentative execution.

Moreover, we have proposed an algorithm that given an iterative property represented as a finite state automaton automatically builds an enforcement mechanism modeled as an edit automaton. To show the advantage of our approach, we presented a theorem specifying how the enforcement mechanisms can be compared and shows that iterative suppression automaton provides a better enforcement than the Longest-valid-prefix automaton in a sense of amount of suppressed actions.

These are the first steps towards closing the gap between the current theoretical works and their practical implementations. As a modest, but still telling example, the running example of Fig. 2 in [3] could not be generated from the policy by any of the formal construction appeared in that paper, nor by the constructions appearing in later papers. In contrast, it can be obtained automatically by our algorithm if, instead of just suppressing every bad iteration, we also emit a warning.

As a future work we consider the case of actions that cannot be fixed. We call these actions *observable* actions. For instance it is important in case of outsourcing services when some requests are transferred to external parties. It is possible to have observable actions also within an organization; for example when a doctor is preparing a set of therapeutical drugs for a specific patient, he takes a wrong drug from a stock, and it is not possible to delete this physical action. It could be modeled as a special kind of iterative suppression automaton that cannot suppress observable actions.

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

Acknowledgments

We would like to thank Andrea Micheletti, Marta Zambelli and Daniela Marino from the e-health unit of Hospital San Rafaele for many helpful suggestions on a previous draft of this paper. Discussion with them were essential to understand in reality what matters and what do not.

We are grateful to the anonymous referees for their constructive suggestions that helped us to improve this paper.

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] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, 1985.

[2] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security*, pages 95–104, Copenhagen, Denmark, 2002. DIKU Technical Report.

[3] L. Bauer, J. Ligatti, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.

[4] N. Bielova and F. Massacci. Do you really mean what you actually enforced? In *Proceedings of the 5th International Workshop on Formal Aspects in Security and Trust*, volume 5491, pages 287–301. Springer-Verlag Heidelberg, 2008.

[5] N. Bielova and F. Massacci. Do you really mean what you actually enforced? Technical Report DISI-08-060, UNITN, 2008.

[6] N. Bielova, F. Massacci, and A. Micheletti. Towards practical enforcement theories. In *Proceedings of The 14th Nordic Conference on Secure IT Systems*, volume 5838 of *Lecture Notes in Computer Science*, pages 239–254. Springer-Verlag Heidelberg, 2009.

[7] A. Brown and M. Ryan. Synthesising monitors from high-level policies for the safe execution of untrusted software. In *Proceedings of the 4th Information Security Practice and Experience Conference*, pages 233–247. Springer-Verlag Heidelberg, 2008.

[8] H. Chabot, R. Khoury, and N. Tawbi. Generating in-line monitors for rabin automata. In *Proceedings of The 14th Nordic Conference on Secure IT Systems*, volume 5838, pages 287–301. Springer-Verlag, 2009.

[9] E. Chang, Z. Manna, and A. Pnueli. Characterization of temporal property classes. In *Proceedings of the 19th International Colloquium on Automata, Languages and Programming (ICALP '92)*, pages 474–486. Springer-Verlag, 1992.

[10] E. Chang, Z. Manna, and A. Pnueli. The safety-progress classification. Technical report, Stanford University, Dept. of Computer Science, London, UK, 1992.

[11] U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.

[12] Y. Falcone, J.-C. Fernandez, and L. Mounier. Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In *Proceedings*

of the Fourth International Conference on Information Systems Security (ICISS'08), pages 41–55. Springer-Verlag Heidelberg, 2008.

[13] Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime verification of safety-progress properties. In *Proceedings of the 9th International Workshop on Runtime Verification (RV'09)*, pages 40–59. Springer-Verlag Heidelberg, 2009.

[14] P.W.L. Fong. Access control by tracking shallow execution history. *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 43–55, 2004.

[15] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, 2006.

[16] R. Khoury and N. Tawbi. Using Equivalence Relations for Corrective Enforcement of Security Policies. In *Proceedings of the 5th International Conference, on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 139–154. Springer-Verlag, 2010.

[17] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[18] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security*, volume 3679 of *Lecture Notes in Computer Science*, pages 355–373. Springer-Verlag Heidelberg, 2005.

[19] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security*, 12(3):1–41, 2009.

[20] D. Marino, J.-J. Portal, M. Hall, C. Bastos Rodriguez, P. Soria Rodriguez, J. Sobota, J. Miksu, and Y. Dwi Wardhana Asnar. Master scenarios. Public Deliverable of EU Research Project D1.2.1, MASTER- Managing Assurance, Security and Trust for Services, Report available at www.master-fp7.eu, 2008.

[21] F. Martinelli and I. Matteucci. Through modeling to synthesis of security automata. In *Proceedings of the Second International Workshop on Security and Trust Management*, volume 179 of *Electronic Notes in Theoretical Computer Science*, pages 31–46, Amsterdam, The Netherlands, 2007. Elsevier Science Publishers B.V.

[22] I. Matteucci. Automated synthesis of enforcing mechanisms for security properties in a timed setting. *Electronic Notes in Theoretical Computer Science*, 186:101–120, 2007.

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

[24] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2-4):158–184, 2007.

Appendix: Proofs of theorems

Proposition 1. *A Longest-valid-prefix automaton constructed by Algorithm 1 for a renewal property $\widehat{P}$ represented by Policy automaton A^P is sound and transparent enforcement mechanism according to $\widehat{P}$. This automaton always outputs the longest valid prefix of the input.*

Proof. Given a Policy automaton A^P we construct a Longest-valid-prefix automaton E following the Algorithm 1. Let us show that automaton E always outputs the longest valid prefix of the input according to the property $\widehat{P}$.

Let us denote the state of E after executing sequence σ with q, and a is the next incoming action. Let us show that this automaton maintains the invariant $\textsc{Inv}_L(\sigma)$ that σ is the input seen so far and σ_o has been output, where σ_o is the longest valid prefix of σ according to $\widehat{P}$. Initially $\textsc{Inv}_L(\cdot)$ holds because $\widehat{P}(\cdot)$ and automaton has not output anything so far. Let us assume that $\textsc{Inv}_L(\sigma)$ holds and prove that $\textsc{Inv}_L(\sigma; a)$ holds as well in any action a.

1) If $\widehat{P}(\sigma; a)$, then there is an accepting run of A^P on $\sigma; a$. It corresponds to the line 5 of the algorithm. In this case the output of E is $*; a$. The memory was obtained in a following way. Every time when the non-accepting state is reached, the action is kept in the memory (line 8) and when finally the sequence becomes valid, a corresponding state of A^P is accepting and the memory of E contains all the read actions. Hence, the output in this case will be $\sigma; a$ and $\textsc{Inv}_L(\sigma; a)$ holds.

2) If $\sigma; a$ is not irremediable then while executing σ the path in E will correspond to the path in A^P, to be more precise, for every next action $\sigma[i]$ there will be a transition in A^P. Therefore, only statements on lines 6 and 8 will be used. Hence E outputs the longest valid prefix and the memory contains all the not yet output actions that make a trace bad and $\textsc{Inv}_L(\sigma; a)$ holds.

3) If $\sigma; a$ is irremediable then the sequence can never become legal again. It means that there was an action $\sigma[j]$ (or a) in the input such that there was no transition in A^P from the state where the run $\sigma[..j-1]$ (or σ) arrived on the input action $\sigma[j]$. At that point condition for statement on line 10 was satisfied, after which no output can be produced. Therefore, if there exists a valid prefix, then it was output when the corresponding accepting state of

A^P was reached before the sequence became irremediable. It means that E outputs the longest valid prefix and $\textsc{Inv}_L(\sigma; a)$ holds.

Therefore, in all the cases the invariant $\textsc{Inv}_L(\sigma; a)$ is maintained and hence the Longest-valid-prefix automaton E constructed by Algorithm 1 always outputs the longest valid prefix. Therefore, in case of valid input the automaton will not change the input (transparency maintained) and in case of invalid input it will output a valid trace (soundness maintained). □

Proposition 2. *An iterative suppression automaton constructed by the Algorithm 2 for an iterative property $\widehat{P}$ represented by Policy automaton A^P is sound and transparent enforcement mechanism according to $\widehat{P}$.*

Proof. Given a Policy automaton A^P let us construct an edit automaton E following the Algorithm 2. Now let us show that automaton E satisfies soundness and transparency according to property $\widehat{P}$.

Let us assume that state q is a state of the automaton A^P after executing sequence σ, and a is the next incoming action. Let us assume that invariant $\textsc{Inv}_e(\sigma)$ stating that $\widehat{P}(E(\sigma))$ and if $\widehat{P}(\sigma)$ then $E(\sigma) = \sigma$ holds and prove that $\textsc{Inv}_e(\sigma; a)$ holds for any action a.

Initially $\textsc{Inv}_e(\cdot)$ holds because automaton has not output anything so far and $\cdot$ is valid. Let us assume that $\textsc{Inv}_e(\sigma)$ holds and prove that $\textsc{Inv}_e(\sigma; a)$ holds as well in any possible case.

1) If $\widehat{P}(\sigma; a)$, then there is an accepting run of A^P on $\sigma; a$. In this case the output of E is $*; a$ (line 6). The memory $*$ was obtained in a following way. Every time when the non-accepting state is reached, the action is kept in the memory (line 8), and when finally the sequence becomes valid, a corresponding state of A^P is accepting and is reached thus the memory of E

contains all the read actions. Hence, the output in this case will be $\sigma; a$ and $\textsc{Inv}_e(\sigma; a)$ holds.

2) If $\sigma; a$ is invalid but not irremediable then while executing σ the path in E will correspond to the path in A^P, to be more precise, for every next action $\sigma[i]$ there will be a transition in A^P. Therefore, only statements on lines 6 and 8 will be used. By doing so E will output the longest valid prefix and memory of E will contain all the not yet output actions that make a trace bad. Hence $\textsc{Inv}_e(\sigma; a)$ holds.

3) If $\sigma; a$ is irremediable then it can never become legal again. According to Algorithm 2, E outputs a a sequence of actions only when a corresponding state of the Policy automaton is reached. Since the property being enforced is an iterative property, then the output of E is always valid. Hence, $\textsc{Inv}_e(\sigma; a)$ holds.

Therefore, in all the cases the invariant $\textsc{Inv}_e(\sigma; a)$ is maintained and hence the edit automaton E constructed by Algorithm 2 satisfies soundness and transparency. □

Theorem 1. *For any iterative property $\widehat{P}$ represented by a Policy automaton A^P an edit automaton E_{IS} constructed by Algorithm 2 is* better *than a Longest-valid-prefix automaton E_{LP} constructed by Algorithm 1 in a sense of Definition 10.*

Proof. According to Definition 10, E_{IS} is better than E_{LP} means that for all irremediable inputs σ with legal substring:

$$d_S(\sigma, E_{IS}(\sigma)) < d_S(\sigma, E_{LP}(\sigma))$$

We have to prove that for some arbitrary iterative property $\widehat{P}$ represented

by the Policy automaton, the equation above always holds. Let us prove first that the distance between σ and $E_{LP}(\sigma)$ and between σ and $E_{IS}(\sigma)$ is never equal to ∞.

E_{LP} always produces the longest valid prefix of the input according to the Proposition 1, hence $d_S(\sigma, E_{LP}(\sigma)) \neq \infty$. According to the Algorithm 2, E_{IS} does not add new actions to the input sequence because it only keeps the input actions in the memory on lines 8 and 14 and output this memory and input action on lines 6 and 12. Hence, $d_S(\sigma, E_{IS}(\sigma)) \neq \infty$.

Let us take an invariant $\textsc{Inv}_d(\sigma)$ stating that for all irremediable inputs σ with legal substring $d_S(\sigma, E_{IS}(\sigma)) < d_S(\sigma, E_{LP}(\sigma))$ holds and prove the theorem by induction using this invariant. Since $\sigma = \cdot$ is a valid sequence, we take as a basis of induction a sequence of two actions $a; b$ such that a is a forbidden action that makes the sequence irremediable and $b \in P$. Then, E_{LP} outputs the longest valid prefix of $a; b$, which is $E_{LP}(a; b) = \cdot$. The run of E_{IS} on input $a; b$ is: an action a is not allowed by the policy P, hence there is no transition from the initial state q_0 of the corresponding Policy automaton on action a. Moreover, action a does not start any new iteration (condition on line 10 does not hold). Hence, according to Algorithm 2: $\langle \delta, \gamma_o, \gamma_k \rangle((q_0, q_0), a) = (q_\perp, q_0) \,|\cdot|\, \cdot$. For the next action b the condition on line 12 holds because b is accepted by the Policy automaton. Hence, on this transition b is output. Therefore, $E_{IS}(a; b) = b$, which means that $d_S(a; b, E_{IS}(a; b)) = d_S(a; b, b) = 1 < d_S(a; b, E_{LP}(a; b)) = d_S(a; b, \cdot) = 2$.

Now, assuming that $\textsc{Inv}_d(\sigma)$ holds for σ let us prove it for $\sigma; a$ where a is an arbitrary action. Let us consider two cases:

1) If σ is a valid trace or σ is invalid trace, but it is not irremediable, or σ is irremediable but has no legal substring in a sense of Definition 10 then $\sigma; a$ does not correspond to the Definition 10.

2) If σ is irremediable with the legal substring, after proceeding over σ the current state of E_{LP} is $q_\perp$ and $E_{LP}(\sigma) = \sigma_o$, where σ_o is the longest valid prefix of σ. The current state of E_{IS} is (q, q_F) and $E_{IS}(\sigma) = \sigma_F$. According to the $\textsc{Inv}_d(\sigma)$, $d_S(\sigma, \sigma_o) = d_S(\sigma, E_{LP}(\sigma)) > d_S(\sigma, E_{IS}(\sigma)) = d_S(\sigma, \sigma_F)$. Then, since E_{LP} is in the error state, the output can never become longer, so $E_{LP}(\sigma; a) = \sigma_o$. However, E_{IS} has a possibility to recover good iterations when recognizing a beginning of a new iteration. Hence, if $\exists q'' \in Q^P . q'' = \delta^P(q_F, a)$ then in case $q'' \in F^P$ we have $E_{IS}(\sigma; a) = \sigma_F; a$ and $E_{IS}(\sigma; a) = \sigma_F$ otherwise. In any case, since $E_{LP}(\sigma; a) = \sigma_o$, we have $d_S(\sigma; a, E_{LP}(\sigma; a)) = d_S(\sigma; a, \sigma_o) = d_S(\sigma, \sigma_o) + 1 > d_S(\sigma, \sigma_F) + 1 = d_S(\sigma; a, \sigma_F) \geq d_S(\sigma; a, E_{IS}(\sigma; a))$ and $\textsc{Inv}_d(\sigma; a)$ holds.

□