RAPHAËL KHOURY

ENFORCING SECURITY POLICIES WITH RUNTIME MONITORS

Thèse présentée
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de doctorat Informatique
pour l'obtention du grade de Philosophiae doctor (Ph.D.)

Département d'informatique
Faculté des sciences et de génie
UNIVERSITÉ LAVAL
QUÉBEC

2011

Résumé

Le monitorage (monitoring) est une approche pour la sécurisation du code qui permet l'exécution d'un code potentiellement malicieux en observant son exécution, et en intervenant au besoin pour éviter une violation d'une politique de sécurité. Cette méthode a plusieurs applications prometteuses, notamment en ce qui a trait à la sécurisation du code mobile.

Les recherches académiques sur le monitorage se sont généralement concentrées sur deux questions. La première est celle de délimiter le champ des politiques de sécurité applicables par des moniteurs opérant sous différentes contraintes. La seconde question est de construire des méthodes permettant d'insérer un moniteur dans un programme, ce qui produit un nouveau programme instrumenté qui respecte la politique de sécurité appliquée par ce moniteur. Mais malgré le fait qu'une vaste gamme de moniteurs a été étudiée dans la littérature, les travaux sur l'insertion des moniteurs dans les programmes se sont limités à une classe particulière de moniteurs, qui sont parmi les plus simples et les plus restreint quant à leur champ de politiques applicables.

Cette thèse étend les deux avenues de recherches mentionnées précédemment et apporte un éclairage nouveau à ces questions. Elle s'attarde en premier lieu à étendre le champ des politiques applicables par monitorage en développabt une nouvelle approche pour l'insertion d'un moniteur dans un programme. En donnant au moniteur accès à un modèle du comportement du programme, l'étude montre que le moniteur acquiert la capacit d'appliquer une plus vaste gamme de politiques de sécurité.

De plus, les recherches ont aussi démontré qu'un moniteur capable de transformer l'exécution qu'il surveille est plus puissant qu'un moniteur qui ne possède pas cette capacité. Naturellement, des contraintes doivent être imposées sur cette capacité pour que l'application de la politique soit cohérente. Autrement, si aucune restriction n'est imposée au moniteur, n'importe quelle politique devient applicable, mais non d'une manière utile ou désirable. Dans cette étude, nous proposons deux nouveaux paradigmes d'application des politiques de sécurité qui permettent d'incorporer des restrictions

raisonnables imposées sur la capacité des moniteurs de transformer les exécutions sous leur contrôle. Nous étudions le champ des politiques applicables avec ces paradigmes et donnons des exemples de politiques réelles qui peuvent être appliquées à l'aide de notre approche.

Abstract

Execution monitoring is an approach that seeks to allow an untrusted code to run safely by observing its execution and reacting if need be to prevent a potential violation of a user-supplied security policy. This method has many promising applications, particularly with respect to the safe execution of mobile code.

Academic research on monitoring has generally focused on two questions. The first, relates to the set of policies that can be enforced by monitors under various constraints and the conditions under which this set can be extended. The second question deals with the way to inline a monitor into an untrusted or potentially malicious program in order to produce a new instrumented program that provably respects the desired security policy.

This study builds on the two strands of research mentioned above and brings new insights to this study. It seeks, in the first place, to increase the scope of monitorable properties by suggesting a new approach of monitor inlining. By drawing on an a priori model of the program's possible behavior, we develop a monitor that can enforce a strictly larger set of security properties.

Furthermore, longstanding research has showed that a monitor that is allowed to transform its input is more powerful than one lacking this ability. Naturally, this ability must be constrained for the enforcement to be meaningful. Otherwise, if the monitor is given too broad a leeway to transform valid and invalid sequences, any property can be enforced, but not in a way that is useful or desirable. In this study, we propose two new enforcement paradigms which capture reasonable restrictions on a monitor's ability to alter its input. We study the set of properties enforceable if these enforcement paradigms are used and give examples of real-life security policies that can be enforced using our approach.

Avant-propos

En premier lieu, je tiens à remercier ma directrice de thèse, Dr. Nadia Tawbi, pour son support et son encadrement tout au long de cette expérience. J'adresse aussi mes sincères remerciements aux membres du jury, les professeurs Josée Desharnais, Mohamed Mejri et Phillip Fong, pour bien avoir accepté d'examiner ma thèse. Je tiens aussi à remercier les nombreuses personnes, amis et collègues, qui m'ont encouragé tout au long de mes études.

Finalement, je tiens à remercier mes parents qui ont sacrifié toutes leurs vies pour m'amener à ce point. Je n'aurais pas pu y arriver sans vous.

à mon père, qui m'a enseigné que
réussite= intelligence + discipline
à ma mère, qui a tout sacrifié pour moi.

It is good to have an end to journey
toward; but it is the journey that
matters, in the end
Ursula K. Le Guin

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ubiquity and growing interconnectivity of modern computer systems impose on users and developers alike that steps be taken to assure the correctness of such systems. Mobile code in particular poses significant risks since the sender is often unknown or untrusted.

One solution that has gained wide acceptance in recent years is runtime monitoring. This approach to code safety allows an untrusted program to run safely by observing its execution and reacting as needed to prevent a violation of the security property. Many security architectures rely upon monitoring, and the wider use of this method holds the promise of greatly mitigating the security risks associated with potentially malicious software and thus of extending the use of mobile and distributed software. Contrasted to other approaches aimed at ensuring software safety, monitoring is characterized by its precision: only executions which violate the security policy are altered or terminated, while valid ones can be allowed to proceed. Furthermore, by being deployed on the client side of a distributed architecture, monitoring can more easily be customized to the specific security needs of each user.

One key element which could lead to a wider use of monitors is to extend the range of security policies that can be enforced by its use. Most monitoring frameworks limit themselves to the enforcement of a specific class of security policies called safety properties. Yet, it has been proved that monitors are capable to enforce a much wider range of security policies, provided they are given the tools to do so.

One option that could be used to extend the range of security policies enforceable by monitors is to rely upon static analysis to produce a model of the program's possible behavior. Prior research has shown that, armed with such a model, a monitor can

enforce security policies that would otherwise be beyond its capabilities. The first contribution of this study is to propose a new method to inline a monitor into an untrusted, and possibly malicious program in order to produce a new version of this program which provably respects the security policy. By drawing on an a priori model of the target program's possible behavior, it becomes possible to enforce properties which lie outside the set of safety properties. We also prove a number of theorems related to the enforcement of securities properties by monitors in this context.

Another element which can be used to extend the range of enforceable policies by monitors is to provide the monitor with the ability to transform its target execution. Prior research has shown that this approach yields the most powerful monitors, but only if the notion of enforcement is sufficiently flexible to allow the monitor to replace a valid sequence with another equivalent sequence. This naturally necessitates the use of an equivalence relation between sequences. But for such an enforcement to be meaningful, the equivalence relation used must be constraining. Otherwise, if the monitor is given too broad latitude to alter the execution, the enforcement may not be useful. In the extreme, if every valid sequence can be thought of as equivalent to each other, any property becomes enforceable simply by always outputting a trivially valid sequence.

The problem is compounded by the fact that the most widely used enforcement paradigm does not place any demand on the monitor when the latter is presented with an invalid sequence, other than it must somehow be corrected. Once the monitor finds that its input sequence is irremediably invalid, it can cast it aside entirely and replace with any valid sequence, no matter how different.

In this study, we develop an alternative policy enforcement framework to study the behavior of monitors capable of transforming their input. In our framework, the monitor's behavior is constrained by a requirement to maintain an equivalence between input and output, regardless of whether this input is valid or not. This intuitively corresponds to an enforcement paradigm, closer to one that would be encountered in practice, in which the actions taken by the monitor are constrained by a limitation that certain behaviors present in the original sequence must be preserved. We study the set of policies enforceable by this paradigm and offer examples of meaningful equivalence relations, which can be used to enforce real-life properties.

In some situations, bounding the monitor to a requirement that the input always be equivalent to the output could be too constraining. Indeed, this sometimes prevents the monitor from taking corrective actions when the input is invalid, as the corrected valid sequence would no longer be equivalent. Even stating the equivalence relation can be problematic. For example, it requires that distinct invalid sequences be thought of as

equivalent if the same valid sequence is a suitable replacement for both. Alternatively, it requires distinct valid sequences to be thought of as equivalent if they are both suitable replacement to the same invalid sequence.

To address these issues, we propose an alternative security enforcement paradigm based on the notion of partial orders rather than equivalence relations. In this framework, the monitor is allowed to replace an invalid sequence with a valid one if the latter is higher on the partial order than the former. This approach enhances our ability to model the corrective behavior of a monitor that replaces invalid sequences with valid ones. We study the enforcement power of monitors operating within this framework, and give examples of several real-life properties that are enforceable. Finally, since several enforcement paradigms are possible for each property, we suggest metrics that allow a user to compare monitors objectively and choose the best enforcement paradigm for a given situation.

The remainder of this study proceeds as follows. In Chapter 2, we review the literature on monitoring, and focus more particularly on studies that address the question of which security policies that be enforced by the approach. In Chapter 3, we present a new approach to in-lining a monitor in an untested program to ensure its compliance with the security policies it captures. In Chapter 4, we propose a new security enforcement framework based on equivalence relations, and in Chapter 5, we suggest an alternative enforcement paradigm based on preorders. Concluding remarks and avenues for future research are given in Chapter 6.

Chapter 2

Review of The Theoretical Models of Monitoring

2.1 Introduction

The first question that arises in the study of monitoring, as in the study of any other security paradigm, is that of identifying precisely which security policies can or cannot be enforced by the enforcement mechanism under consideration. Only then does it become possible to compare the mechanism's expressive power to that of other enforcement mechanisms or develop means to make it more powerful.

To address this issue we begin in Section 2 by giving a rigorous definition of the class of security policies that can be enforced on a target program by monitors. We further give a formal definition of the notions of executions, security policy and monitor that we will manipulate. We then show in Section 3 how these definitions allow us to state the limitations of a rudimentary monitor. From these limitations the set of security properties enforceable by this monitor can be deduced. In Section 4, we examine various ways to enhance the power of monitors, giving in each case a new formal definition of the extended monitor. These definitions will serve to identify the most promising model. We next turn to the enforcement power of the those monitors found to be the most powerful and give a lower bound to the set of properties they can enforce. In Section 5, we revisit the notion of enforcement and propose alternative definitions. Then, in Sections 6 and 7, we examine how computational and memory constraints can affect the set of enforceable properties. The question of how to inline a monitor into a target program is examined in Section 8. Finally, other related work of interest on this topic

are addressed in Section 9.

2.2 Security Policies and Security Properties

The first step in our analysis is to define a theoretical framework that can accommodate the various concepts we elaborate, such as executions, security policies and monitors. To this end we adopt the architecture devised in [11] which is widely used in the field.

An execution σ of a program, or trace, is modeled as a finite or infinite sequence of atomic program actions :

$$\sigma = s_0, s_1, s_2...$$

We let s range over a finite or countably infinite set of atomic actions Σ. The empty sequence is noted ϵ, the set of all finite length sequences is noted Σ^*, that of all infinite length sequences is noted Σ^ω, and the set of all possible sequences is noted $\Sigma^\infty = \Sigma^\omega \cup \Sigma^*$. Likewise, for a set of sequences $\mathcal{S}$, $\mathcal{S}^*$ denote the finite iterations of sequences of $\mathcal{S}$ and $\mathcal{S}^\omega$ that of infinite iterations, and $\mathcal{S}^\infty = \mathcal{S}^\omega \cup \mathcal{S}^*$. Let $\tau \in \Sigma^*$ and $\sigma \in \Sigma^\infty$ be two sequences of actions. We write $\tau;\sigma$ for the concatenation of τ and σ. We say that τ is a prefix of σ noted $\tau \preceq \sigma$,or equivalently $\sigma \succeq \tau$ *iff* there exists a sequence σ' such that $\tau;\sigma' = \sigma$. We write $\tau \prec \sigma$ (resp. $\sigma \succ \tau$) for $\tau \preceq \sigma \wedge \tau \neq \sigma$ (resp. $\sigma \succ \tau \wedge \tau \neq \sigma$). Finally, let $\tau, \sigma \in \Sigma^\infty$, τ is said to be a suffix of σ iff there exists a $\sigma' \in \Sigma^*$ s.t. $\sigma = \sigma';\tau$.

We denote by $pref(\sigma)$ (resp. $suf(\sigma)$) the set of all prefixes (resp. suffixes) of σ. Let $A \subseteq \Sigma^\infty$ be a subset of sequences. Abusing the notation, we let $pref(A)$ (resp. $suf(A)$) stand for $\bigcup_{\sigma\in A} pref(\sigma)$ (resp. $\bigcup_{\sigma\in A} suf(\sigma)$). The i^{th} action in a sequence σ is given as σ_i, $\sigma[i,j]$ denotes the sequence occurring between the i^{th} and j^{th} actions of σ, $\sigma[..i]$ denotes the prefix of sequence σ, up to its i^{th} action and $\sigma[i..]$ denotes the remainder of the sequence, starting from action σ_i. The length of a sequence $\tau \in \Sigma^*$ is given as $|\tau|$.

Let k be an integer. We write $\Sigma_k = \{\sigma \in \Sigma^* : |\sigma| = k\}$ to denote the set of sequences from Σ^* of length k, and $\Sigma_{\leq k} = \{\sigma \in \Sigma^* : |\sigma| =\leq k\}$ to denote the set of sequences of length less or equal to k. Likewise, $pref_k$ and suf_k denote the sets of prefixes and suffixes of length k.

A security policy $P \subseteq \wp\{\Sigma^\infty\}$ is a set of sets of allowed executions. A policy P is a property iff it can be characterized as a set of sequences for which there exists a decidable predicate $\hat{\mathcal{P}}$ over the executions of $\Sigma^\infty : \hat{\mathcal{P}}(\sigma)$ iff σ is in this set. In other words, a property is a policy for which the membership of any sequence can be determined by examining only the sequence itself. Such a sequence is said to be valid or to respect the property.

An example of security policies that are not properties are information flow properties, which limit the way that the execution of a certain trace may influence another. More generally, a policy which forbids that two identical traces occur, or which states that should a certain execution be possible another one must also be allowed, are not security properties, since it is impossible while examining a single execution, to decide whether or not this execution respects the security policy. Since all policies enforceable by monitors are properties, P and $\hat{\mathcal{P}}$ are used interchangeably. Additionally, since the properties of interest represent subsets of Σ^∞, we follow the common usage in the literature and freely use $\hat{\mathcal{P}}$ to refer to these sets. The distinction between a set of valid executions and the predicate over individual sequences that indicates if a given sequence is in this set is only relevant in section 2.6, where we discuss the work of Schneider et al.

A number of classes of properties have been defined in the literature and are of special interest in the study of monitoring. First are *safety* properties [48], which proscribe that certain "bad things" occur during the execution. Let Σ be a set of actions and $\hat{\mathcal{P}}$ be a property, $\hat{\mathcal{P}}$ is a *safety* property iff

$$\forall \sigma \in \Sigma^\infty : \neg\hat{\mathcal{P}}(\sigma) \Rightarrow \exists \sigma' \preceq \sigma : \forall \tau \succeq \sigma' : \neg\hat{\mathcal{P}}(\tau) \qquad \text{(safety)}$$

Informally, this states that any infinite length sequence does not respect the security property if there exists a point in that sequence from which any possible extension does not respect the security policy. This implies that a violation of a safety property is irremediable: once a violation occurs, nothing could be done to correct the situation. This definition also requires that we are able to detect and identify the precise point at which the security property violation occurs.

Alternatively, a *liveness* property [2] is a property prescribing that a certain "good thing" must occur in any valid execution. Formally, for an action set Σ and a property $\hat{\mathcal{P}}$, $\hat{\mathcal{P}}$ is a *liveness* property iff

$$\forall \sigma \in \Sigma^* : \exists \tau \in \Sigma^\infty : \tau \succeq \sigma \wedge \hat{\mathcal{P}}(\tau) \qquad \text{(liveness)}$$

Informally, the definition states that a property is a liveness property if any finite sequence can be extended into a valid sequence. Finally, it is often useful to restrict our analysis to properties for which the empty sequence ϵ is valid. Such properties are said to be *reasonable*. Formally, for an action set Σ and a property $\hat{\mathcal{P}}$, $\hat{\mathcal{P}}$ is *reasonable* iff $\epsilon \in \hat{\mathcal{P}}$.

Some properties are neither safety nor liveness, though it was shown in [2] that any property can be expressed as the conjunction of a safety property and a liveness property. Furthermore, if the set of atomic actions Σ contains more than one element, any property can be stated as the intersection of two liveness properties. We can thus think of any security policy as comprising a liveness component and a safety component. The only property that is both safety and liveness is the trivial property that accepts all executions.

One of the most interesting results in the study of security properties is the ability to represent properties as automata. This representation is widely used, and the automata representation of properties is at the heart of both model checking and monitoring.

A widely used example of the representation is the Büchi [29] automaton. A Büchi automaton is a non-deterministic finite state automaton that accepts infinite length sequences. We represent such automata by a tuple of the form $\langle \Sigma, Q, Q', \delta, F \rangle$ where, Σ is an input alphabet, Q is a set of states, $Q' \subseteq Q$ is a set of initial states, $\delta \subseteq Q \times \Sigma \times Q$ is a transition relation and $F \subseteq Q$ is a set of accepting states. A sequence is accepted if it enters an accepting state infinitely often. Properties expressed in some temporal logics can be translated in Büchi automata. While algorithms exist to translate a non-deterministic Büchi automaton into an equivalent deterministic one, this transformation is not always possible since deterministic Büchi automata are strictly less expressive than their deterministic counterparts. Alternatively, a Büchi automata can be transformed into an equally expressive deterministic formalism, such as the Müller [61] automata or the Rabin [65] automata. We will give a more formal definition of the Büchi automaton in the next section.

Finally, we need to provide a definition of what it means to "enforce" a security property $\hat{\mathcal{P}}$. A number of possible definitions have been suggested. Yet, all have in common that they revolve around imposing that the following two principles be respected.

1. Soundness : All observable behavior of the target program respects the desired property, i.e. every output sequence of the program is present in the set of executions defined by $\hat{\mathcal{P}}$.

2. Transparency : The semantics of valid executions is preserved, i.e. any execution of the unmonitored program that respects the security property must still be present in the monitored program. On this point, we can distinguish between precise enforcement, which does not allow any transformation of the input sequence, and equivalent enforcement, where a valid sequence can be transformed into another equivalent sequence, with respect to some equivalence relation.

Some authors have suggested relaxing these restrictions, for instance by lifting the transparency requirement and allowing some valid executions to be rejected, but the enforcement power of these paradigms have not yet been formally studied. We will revisit these concepts, and give a formal definition of the notion of enforcement in Sections 2.4 and 2.5 and suggest new alternative enforcement paradigms in Chapters 4 and 5.

2.3 Security Automaton Modeling of Monitors

Schneider [69] is the first author to investigate the question of delineating exactly which security policies are enforceable by monitors. He attempted to identify the range of properties enforceable by a monitor that merely observes the execution of a target program, with no knowledge of its possible future behavior, and with no ability to affect it, except by aborting the execution. He designates EM (for Execution Monitoring) as the subset of security policies that could be enforced by these monitors. Although he admits that monitors could be made more powerful if these limitations were relaxed, the class he identifies as EM does serve as a lower bound to the set of security policies enforceable by monitors.

The set of policies in EM is bounded by three limits of these tools. Firstly, such an enforcement mechanism can only accept or reject an execution by considering it alone, without comparing it to prior runs of the same program. As discussed earlier, this is formally expressed as :

$$(\exists \hat{\mathcal{P}} : \forall \sigma \in \Sigma : \sigma \in P \Leftrightarrow \hat{\mathcal{P}}(\sigma)) \tag{2.3.1}$$

where $\hat{\mathcal{P}}$ is a predicate over executions. This implies that only security policies which are also properties are enforceable by the mechanism under consideration.

Secondly, since the monitor does not have access to information about the possible behavior that the program may or may not exhibit, it can never allow an invalid partial

execution, with the confidence that it will necessarily be corrected, on all possible execution paths. The monitor can thus only enforce properties for which a violation of the property is irremediable. This is formalized as follows :

$$(\forall \tau \in \Sigma^* : \neg \widehat{P}(\tau) \Rightarrow (\forall \sigma \in \Sigma^\infty : \neg \widehat{P}(\tau; \sigma))) \tag{2.3.2}$$

If a finite sequence does not respect the security policy, this equation requires that it can never be extended into a finite or infinite sequence that does.

Thirdly, before the execution of each action by the target program, the monitor must decide either to accept it or otherwise abort the execution. It follows that any rejected execution is rejected after a finite amount of time has elapsed. This requirement is formally stated as :

$$(\forall \sigma \in \Sigma^\infty : \neg \widehat{P}(\sigma) \Rightarrow (\exists i : \neg \widehat{P}(\sigma[0..i]))) \tag{2.3.3}$$

According to the classification of security policies presented in the previous section, a policy that satisfies the requirements of equations 2.3.1, 2.3.2 and 2.3.3 is a safety property. Yet, saying that monitors exactly enforce safety properties would be false on two counts. First, the analysis presented above only applies to a subset of monitors, whose behavior is especially constrained. Schneider notes that the range of enforceable security properties could be extended if the definition of a monitor was relaxed, for example by giving the monitor access to information about a program's possible behavior or the ability to alter the target program's behavior in some way. Furthermore, he also points out that other constraints, such as computability constraints, which are unavoidable, would restrict further the behavior of monitors. The set of safety properties can then best be seen as an upper bound to the properties enforceable by the simplest, most constrained monitors.

Finally, Schneider observes that safety properties can be modeled by a specific subclass of Büchi automaton, termed the security automaton.

Definition 2.3.1. *A Büchi automaton is a tuple* $\langle \Sigma, Q, Q_0, \delta, F \rangle$ *such that*

- Σ *is a finite or countably infinite set of symbols;*
- Q *is a finite or countably infinite set of states;*
- $Q_0 \subseteq Q$ *is a subset of initial start states;*

- $\delta : Q \times \Sigma \to 2^Q$ *is a (possibly partial) transition relation;*
- $F \subseteq Q$ *is an acceptance set. An infinite sequence* ρ *of symbols from* Σ *is valid iff at least one state present in the acceptance condition occurs infinitely often in this sequence. Such states are said to be valid or accepting.*

Sequences of symbols model executions of the target program, and valid executions are recognized when they reach a valid state infinitely often. Yet, when restricted to safety properties Schneider shows that a simpler definition can be used, where every state in Q is accepting and the execution is rejected as soon as an attempt is made to take an undefined transition. This alternative form of Büchi automaton, termed the security automaton was first suggested in [3], and can recognize exactly the set of safety properties. These automaton have been widely used in monitoring on account of the close connection between safety properties and monitorable properties, which we outlined above.

Definition 2.3.2. *A security automaton is a deterministic*[1] *automaton* $\langle Q, q_0, \delta \rangle$ *where*

- Σ *is a finite or countably infinite set of symbols;*
- Q *is a finite or countably infinite set of states;*
- $q_0 \subseteq Q$ *is a subset of initial start states;*
- $\delta : Q \times \Sigma \to Q$ *is a (possibly partial) transition function. Rather than using an acceptance set, the execution is aborted if it attempts a transition not present in* δ*.*

2.4 Extending the range of enforceable properties using the Edit Automata

Schneider's research should not be misunderstood as indicating that only safety properties can ever be enforced by monitors. Indeed, his paper [69] only discussed a specific type of monitor; one which was limited in its capacity to react to a security policy violation, and operated with no knowledge of its target's possible behavior. Schneider's study also suggested that the set of properties enforceable by monitors could be

[1] While the original definition in [11] allowed the security automaton to be non-deterministic, subsequent work focused on deterministic automata.

extended under certain conditions. Building on this insight, Ligatti, Bauer and Walker [11, 53] modify Schneider's definition along three axes, namely:

1. according to the means put at the disposal of the monitor to react to a possible security policy violation. In this regard, they distinguish between monitors that can
   - abort the execution of the target program;
   - suppress a disallowed action and continue the execution;
   - insert some action or actions into the control flow;
   - insert or suppress actions in the control flow, effectively combining the power of the previous two models.
2. according to the information made available to the monitor about the possible executions of the program. In this case, the authors distinguish between the uniform context, in which a monitor operates with no knowledge about its target's possible behavior, and the nonuniform context, in which the monitor has knowledge that certain behaviors cannot be exhibited by the target program. Ligatti et al. limit their analysis to finite sequences. Let $\mathcal{S}$ stand for the set of sequences which the monitor considers as possible executions of the target program. A monitor operates in a uniform context if $\mathcal{S} = \Sigma^*$, and in the nonuniform context if $\mathcal{S} \subseteq \Sigma^*$.
3. according to how much latitude the monitor is given to transform its target's execution. The authors distinguish between precise enforcement, which imposes that every action performed by the target program in a valid execution be preserved, and effective enforcement which allows a valid execution to be transformed into a semantically equivalent execution. This naturally necessitates the use of an equivalence relation $\cong$ on executions.

To model the various possible enforcement paradigms, Ligatti et al. introduce four new classes of automata. Unlike the security automata proposed by Schneider, which is designed to *recognize* whether or not an input sequence is valid, the automata proposed by Ligatti et. al. are designed to *modify* the input sequence and output a new one that respects the security policy. The target program's execution is the monitor's input, which is not visible to outside observers. The automata's output represents the "corrected" behavior of the target program, after the monitor has transformed it.

The execution of an automaton is specified using single step judgements of the form $(q, \sigma) \xrightarrow{\tau} (q', \sigma')$ where, q is the current state, σ is the sequence the target program

wishes to execute, τ is the sequence of at most a single action which the monitor outputs on this step, q' denotes the successor state and σ' is the input sequence after this step has been taken. Each such judgement captures a single step of the execution of the monitor. These single-step judgements are generalized to multi-step judgements through reflexivity and transitivity rules as follows.

Definition 2.4.1 (Multi-step semantics, from [11])**.** *The multi-step relation* $(q,\sigma)\stackrel{\tau}{\Longrightarrow}(q',\sigma')$ *is inductively defined as follows.*

For all $q,q',q''\in Q$, $\sigma,\sigma',\sigma''\in\Sigma^{\infty}$ *and* $\tau,\tau'\in\Sigma^{*}$ *we have*

$$(q,\sigma)\stackrel{\varepsilon}{\Longrightarrow}(q,\sigma) \tag{2.4.1}$$

$$\text{if } (q,\sigma)\stackrel{\tau}{\Longrightarrow}(q'',\sigma'') \text{ and } (q'',\sigma'')\stackrel{\tau'}{\longrightarrow}(q',\sigma') \text{ then } (q,\sigma)\stackrel{\tau;\tau'}{\Longrightarrow}(q',\sigma') \tag{2.4.2}$$

The different enforcement paradigms studied in this research, namely: truncation, suppression, insertion and edition, are modeled using different transition relations δ. The simplest model is the truncation automaton, which simulates the behavior of Schneider's security automaton, since this automaton can only either accept each input action, or abort the execution. It is thus defined by a transition function δ in which every transition is restricted to those two options.

Definition 2.4.2. *A truncation automaton* $\mathcal{T}$ *is a tuple* $\langle Q,\Sigma,q_0,\delta\rangle$ *where*

- Q *is a finite or countably infinite set of states;*
- Σ *is a finite or countably infinite set of atomic actions;*
- q_0 *is the initial state, and*
- $\delta : Q\times\Sigma\to Q$ *is a complete and deterministic transition function, which indicates the output of the automaton when a given action is received as input in a given state. In the case of the truncation automaton, the monitor is restricted in that at any step of the execution, the monitor will either output the same action that has been input, or abort. This is captured by the following the operational semantics:*
  $(q,\sigma)\stackrel{a}{\to}(q',\sigma')$ *if* $\sigma=a;\sigma'$ *and* $\delta(q,a)=q'$
  $(q,\sigma)\stackrel{\epsilon}{\to}(q',\epsilon)$ *otherwise*

The suppression automaton possesses an added capacity to insert other actions into the output stream. This is reflected in its transition function.

Definition 2.4.3. *A suppression automaton $\mathcal{D}$ (for deletion) is a tuple $\langle Q, \Sigma, q_0, \delta, \omega \rangle$ where*

- *Q is a finite or countably infinite set of states;*
- *Σ is a finite or countably infinite set of atomic actions;*
- *q_0 is the initial state;*
- *$\delta : Q \times \Sigma \to Q$ is a partial and deterministic transition function and*
- *$\omega : Q \times \Sigma \to \{+, -\}$ is a partial function with the same domain as δ that indicates whether or not the current input action should be suppressed $(-)$ or inserted $(+)$. This is captured by the following the operational semantics:*
  $(q, \sigma) \xrightarrow{a} (q', \sigma')$ *if* $\sigma = a\sigma'$,$\delta(q, a) = q'$ *and* $\omega(q, a) = +$
  $(q, \sigma) \xrightarrow{\epsilon} (q', \sigma')$ *if* $\sigma = a\sigma'$,$\delta(q, a) = q'$ *and* $\omega(q, a) = -$
  $(q, \sigma) \xrightarrow{\epsilon} (q, \epsilon)$ *otherwise*

The insertion automata models a monitor that can insert actions into the control flow. Its transition function thus indicates if a given input action must be output alone or alongside a finite sequence of other actions, or whether the execution must be terminated at that point.

Definition 2.4.4. *An insertion automaton $\mathcal{I}$ is a tuple $\langle Q, \Sigma, q_0, \delta, \gamma \rangle$ where*

- *Q is a finite or countably infinite set of states;*
- *Σ is a finite or countably infinite set of atomic actions;*
- *q_0 is the initial state;*
- *$\delta : Q \times \Sigma \to Q$ is a partial and deterministic transition function and*
- *$\gamma : Q \times \Sigma \to \Sigma^* \times Q$ is a partial deterministic function, termed the insertion function, which specifies which actions (if any), the monitor should output at each step of the execution. The domain of the insertion function is disjoin from the domain of the transition function. This is captured by the following the operational semantics:*
  $(q, \sigma) \xrightarrow{a} (q', \sigma')$ *if* $\sigma = a; \sigma'$ *and* $\delta(q, a) = q'$
  $(q, \sigma) \xrightarrow{\tau} (q', \sigma)$ *if* $\sigma = a; \sigma'$ *and* $\gamma(q, a) = \tau, q'$
  $(q, \sigma) \xrightarrow{\epsilon} (q, \epsilon)$ *otherwise*

Finally, the most powerful model, to which we now turn, is the edit automaton, which combines together the expressive power of the two previous automata[2].

Definition 2.4.5. *An edit automaton $\mathcal{E}$ is a tuple $\langle Q, \Sigma, q_0, \delta \rangle$ where*

- *Q is a finite or countably infinite set of states;*
- *Σ is a finite or countably infinite set of atomic actions;*
- *q_0 is the initial state, and*
- *$\delta : (Q \times \Sigma) \to (Q \times \Sigma^\infty)$ is a (possibly partial) deterministic transition function, which indicates the output of the automaton when a given action is received as input in a given state.*

Let $\mathcal{A}$ be any of the automata defined above, we let $\mathcal{A}(\sigma)$ be the output of $\mathcal{A}$ when its input is σ.

By using a formal model of monitors we can define more formally the notion of enforcement. Recall that this revolves around two criteria, correctness, which imposes that the output of the monitor always be valid, and transparency, which requires that the semantics of valid executions be preserved. While the first criterion is straightforward, the second can be stated in a number of different ways, leading to alternative notions of enforcement, which in turn induce different sets of enforceable properties for the same monitor. Two notions of enforcement are particularly interesting in this regard: *precise enforcement* and *effective$_{\cong}$ enforcement*.

A monitor precisely enforces a property if, at any step of the execution, it accepts the input action, provided that it is part of a valid sequence.

Definition 2.4.6. *Let Σ be a set of atomic actions and $\mathcal{S} \subseteq \Sigma^\infty$ be a subset of sequences. An automaton $\mathcal{A}$ precisely enforces a Property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \mathcal{S}$*

1. $(q_0, \sigma) \stackrel{\sigma'}{\Longrightarrow}_{\mathcal{A}} (q', \epsilon)$;
2. $\hat{\mathcal{P}}(\sigma')$; *and*
3. $\hat{\mathcal{P}}(\sigma) \Rightarrow \forall i : \exists q'' : (q_0, \sigma) \stackrel{\sigma'[..i]}{\Longrightarrow}_{\mathcal{A}} (q'', \sigma[i+1..])$

[2]This definition, taken from [76], is equivalent to the original definition of the edit automaton given in [53].

This definition is very restrictive, and goes far beyond imposing syntactic equality between valid input and output. For a monitor to precisely enforce a property, it is necessary that every action present in a valid sequence be output in lockstep with the input. This condition can be relaxed to allow the monitor to transform some valid sequences, as long as the output is equivalent to the input, w.r.t. some equivalence relation $\cong$.

Definition 2.4.7. *Let Σ be a set of atomic actions and $\mathcal{S} \subseteq \Sigma^\infty$ be a subset of sequences. An automaton $\mathcal{A}$ effectively$_\cong$ enforces a Property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \mathcal{S}$*

1. $(q_0, \sigma) \stackrel{\sigma'}{\Longrightarrow}_{\mathcal{A}} (q', \epsilon)$;
2. $\hat{\mathcal{P}}(\sigma')$; *and*
3. $\mathcal{A}(\sigma) \cong \sigma'$

The equivalence relation $\cong$ can capture any semantic property of sequences. Ligatti et al. only impose that it respects an *indistinguishability* requirement.

$$\forall \hat{\mathcal{P}} : \forall \sigma, \sigma' \in \Sigma^\infty : \sigma \cong \sigma' \Rightarrow (\hat{\mathcal{P}}(\sigma) \Leftrightarrow \hat{\mathcal{P}}(\sigma')) \qquad \text{(indistinguishability)}$$

In [21], Chabot shows that syntactic equality is the only equivalence relation which respects this definition. In Chapter 5, we revisit the notion of equivalence and suggest an alternative manner in which they can be used in a monitoring context.

By contrasting various combinations of the different possibilities enumerated above, Ligatti et al. provide a rich taxonomy of classes of security policies, associated with the appropriate model to enforce them. This also illustrates how each of the factors discussed above can influence the set of policies that can be enforced by a monitor.

The simplest model presented in this regard is that of the truncation automaton, which precisely enforces a property on a uniform system. This case is identical to that of the security automaton discussed above, which has been shown to enforce exactly the set of safety properties. In the nonuniform case however, some liveness properties can be enforced. This happens when static analysis or code instrumentation assures the monitor that any partial sequence that violates the property will eventually be corrected. The monitor can thus allow the invalid sequence to proceed, with the confidence that any violation will eventually be corrected. We will examine such nonuniform enforcement by a truncation automaton in greater detail in the next chapter.

The set of policies effectively applicable by truncation automata is also greater than what this mechanism can precisely enforce. The authors give the example of an equivalence relation stating that a certain prefix of an execution is equivalent to each of its valid extensions. The automaton can thus accept, and terminate the execution, when this prefix is input. Yet, this property would not be precisely enforceable if any valid sequence had infinitely many invalid prefixes.

The next model examined is that of the suppression automaton. In a uniform system, the set of policies precisely enforceable by this automaton is the same as that of the truncation automaton. It is perhaps surprising that the ability to suppress an action gives the monitor no added power. This is a result of the narrow definition of precise enforcement: we require that executions which respect the security policy be accepted without modification, and that each action be output by the automaton in lockstep with the program. Since it is impossible to both modify an execution and respect this requirement, the added ability of the suppression automaton cannot extend the range of precisely enforceable policies in a uniform system. In a nonuniform system however, the suppression automaton precisely enforces some properties that can never be enforced by a truncation automaton. The reason is best understood if illustrated by an example. Consider a policy that requires that any acquired resource is released and that it is used no more than n times. Static analysis may be used to ensure compliance with the first part of this requirement, but the question as to how many uses are made of a resource is undecidable in the general case. A truncation automaton cannot enforce the policy since, should an execution attempt to use more than n of the protected resource its only possible reaction would be to allow the invalid sequence or terminate the execution without releasing the resource. Yet, this policy can be precisely enforced by a suppression automaton which simply suppresses the extra uses of the resource while waiting for the release action. In the case of effective enforcement, we also find that the suppression automaton is more powerful than the truncation automaton. Here, the added power is derived from the possibility of suppressing an action that is both superfluous (according to the equivalence relation) and potentially illegal.

Similarly to the two automata discussed above, the insertion automata in a uniform system can only precisely enforce safety properties, because of the constraints imposed upon it by the notion of precise enforcement. On a nonuniform system, the insertion automaton can precisely enforce any property that can be precisely enforced by the suppression automaton. This is because the insertion automaton can mimic the behavior of the suppression automaton in the following way: if the suppression automaton suppresses an action a, it can do so because it knows, from static analysis, that the prefix of the execution that has already elapsed could be extended, possibly by several suffixes. The insertion automaton has only to affix one of this suffixes and terminate the

execution to ensure the respect of the security policy. There are also properties that are precisely enforced by the insertion automaton, but not by the suppression automaton. These cases occur when the insertion automaton uses its ability to insert actions into the control flow and correct an otherwise invalid sequence. From this we can conclude that the set of precisely enforceable policies by the insertion automaton on nonuniform systems is a superset of that which can be enforced by the suppression automaton in the same context. This result, however, does not hold for effective enforcement, where the enforcement power of the insertion and suppression automata are orthogonal. This observation is a bit surprising in light of the previous discussion, and follows from the fact that in certain cases, no actions can be inserted in the sequence and at the same time preserve the semantics of the sequence intact, whereas this can be achieved if one or more actions are suppressed.

The last automaton developed by Ligatti, Bauer and Walker is the edition automaton. Intuitively, it combines the capabilities of all the previous models. Yet, on uniform systems, it remains limited to precisely enforcing safety properties, as are all the other automata that were examined earlier, because of the strict definition of precise enforcement. In the case of nonuniform systems, the edit automaton enforces exactly the same set of properties as the insertion automaton. This result is not surprising since the edit automaton is a combination of the insertion and suppression automaton and, as mentioned earlier, the insertion automaton can enforce in this context a superset of the properties applicable by the suppression automaton. Finally, the set of properties effectively enforced by the edit automaton is a superset of the set of properties enforceable by any other enforcement paradigm we have previously considered. The power of the edit automaton derives from its ability to suppress an input sequence as long as it is potentially invalid, only to reinsert it if it is valid. Since this analysis is limited to finite sequences, and the empty sequence is always assumed to be valid, the edit automaton can effectively enforce any property.

The taxonomy presented above is summarized in figures 2.1 and 2.2, adapted from [11]. Two observations emerge particularly from these figures. The first is that the range of security policies enforceable by most paradigms of monitoring is extended in the non-uniform context, as opposed to the uniform context, with all other factors remaining equal. Secondly, a monitor capable of more varied responses when faced with a violation of the security property may be able to enforce a wider range of security properties, but only if it is given sufficient latitude to alter a valid input sequence by its equivalence relation. Alternatively, if the equivalence relation is too strict and the monitor is unable to transform valid sequences, the added power of the monitor is lost and the set of enforceable security properties is not extended.

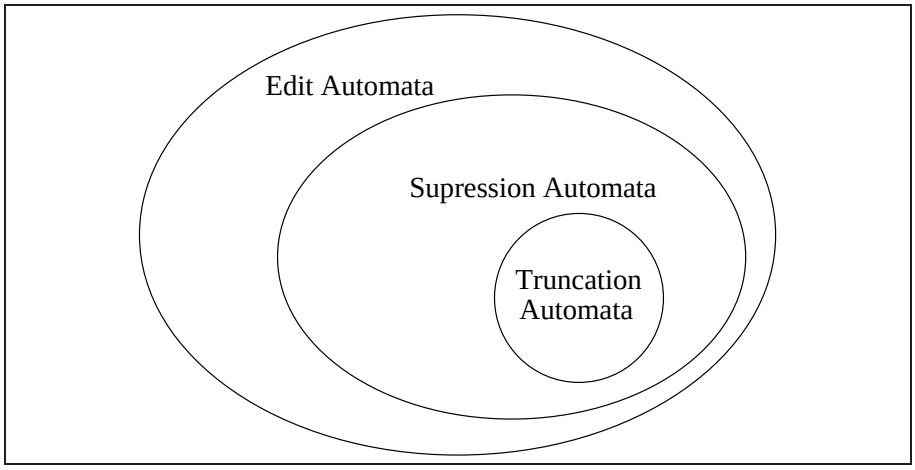


Figure 2.1: Precise application, on non-uniform systems

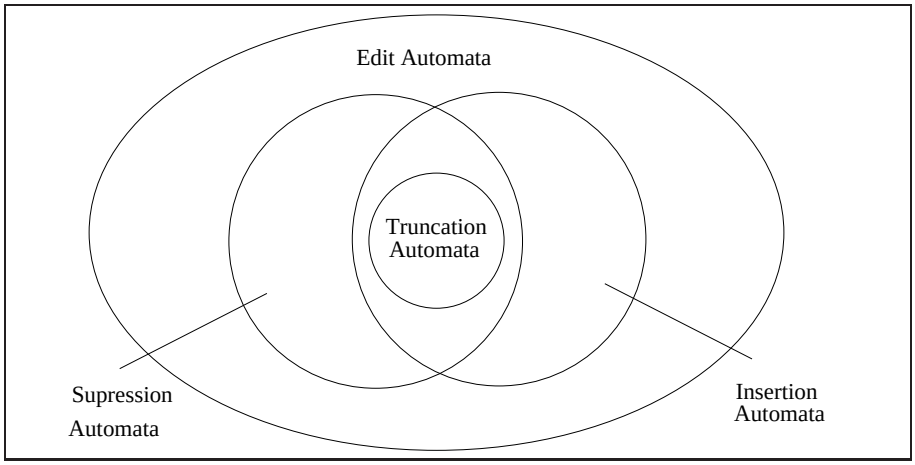


Figure 2.2: Effective$_{\cong}$ application

2.4.1 The edit automaton and Infinite Sequences

The analysis from [11] has focused exclusively on finite sequences. Yet, the behavior of many systems is possibly nonterminating. In [54, 55], Ligatti et al. extend the above analysis to a more general framework containing both finite and infinite sequences. They focus on effective$_{\cong}$ enforcement by the edit automaton, since this is the more powerful framework. Effective enforcement is dependent on an equivalence relation between executions. Since these relations tend to be highly system-specific, Ligatti et. al. focus on syntactic equality (=). It can be argued that any other equivalence relation is more inclusive, the set of policies effectively$_{=}$ enforceable is thus a lower bound on the set of policies that can be effectively enforced by the edit automaton.

Intuitively, the edit automaton works in the following way : as long as the actions input by the target program comply with the security policy, they are immediately output. If, however, the automaton detects that the input actions are part of a potentially malicious sequence, the edit automaton may suppress them (outputting nothing), and then store the input sequence until it is certain that they do indeed respect the security policy, at which point the automaton will re-insert them. On the other hand, if the automaton determines that the input sequence is irrevocably invalid, it terminates the execution, having output only a valid prefix of the input sequence. The monitor is in fact *simulating* the execution of the program until it is certain that the behavior it has so far witnessed is correct. For the time being, it is assumed that the monitor has an unlimited ability to do this for all program actions. In [32], it is observed that a monitor behaving in this manner always outputs the longest valid prefix of its input, if the latter is invalid.

The authors define the set of infinite renewal properties (*Renewal*) in order to characterize the set of properties enforceable by an edit automaton in the following way. A property is a member of this set if every infinite valid sequence has infinitely many valid prefixes, while every invalid infinite sequence has only finitely many such prefixes. Formally, for an action set Σ and a property $\hat{\mathcal{P}}$, $\hat{\mathcal{P}}$ is a *Renewal* property iff it meets the following two equivalent conditions

$$\forall \sigma \in \Sigma^{\omega} : \hat{\mathcal{P}}(\sigma) \Leftrightarrow \{\sigma' \preceq \sigma | \hat{\mathcal{P}}(\sigma')\} \textit{is an infinite set} \qquad (\text{renewal}_1)$$

$$\forall \sigma \in \Sigma^{\omega} : \hat{\mathcal{P}}(\sigma) \Leftrightarrow (\forall \sigma' \preceq \sigma : \exists \tau \preceq \sigma : \sigma' \preceq \tau \wedge \hat{\mathcal{P}}(\tau)) \qquad (\text{renewal}_2)$$

Note that the definition of Renewal imposes no restrictions on the finite sequences in $\hat{\mathcal{P}}$. This is consistent with their result in [11] that on systems containing only finite sequences, every property is enforceable by the edit automaton. For infinite sequences, the set of Renewal properties includes all safety properties, some liveness properties

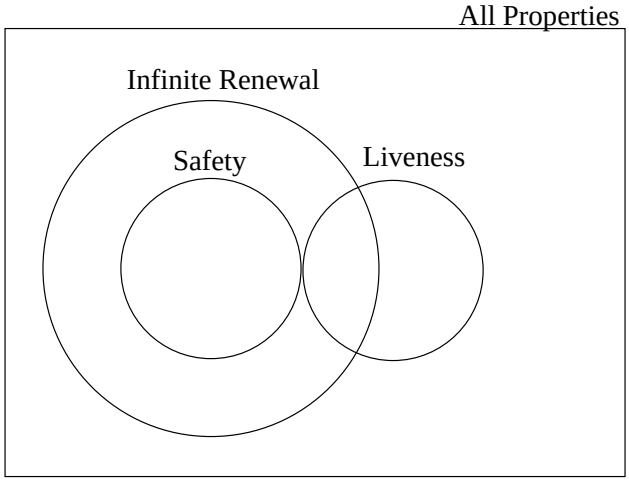


Figure 2.3: The set of Infinite Renewal Properties

and some properties which are neither safety nor liveness. For example, a property that states that a given action must eventually occur in any non-empty execution fits this definition since any valid infinite length execution has infinitely many valid prefixes (prefixes in which that action occurs) while any invalid infinite length execution has only a finite number of valid prefixes (none). The set of infinite renewal properties, contrasted to that of safety and liveness, is given in figure 2.3.

The set of reasonable infinite renewal properties is a lower bound to the set of properties that can be effectively$_{=}$ enforced by the edit automaton. In some cases, this mechanism enforces properties that are not in Renewal. This happens when, at a certain point, the automaton determines that the current prefix of the execution it has been fed can only be extended in a single sequence that respects the security policy. In this case, it can output that sequence and terminate the execution immediately. Ligatti et al. however, regard this case as a marginal corner case.

It is also important to emphasize that this is only a lower bound on the set of properties that can be effectively enforced by the edit automaton. Indeed, the enforcement mechanism does not rely upon the power of the edit automaton to transform an execution in order to make it compatible with the security requirement. Rather than reject an illegal execution, the automaton could either delete some troublesome actions into the control flow and output a modified, valid sequence, or add an action that must inevitably happen for an execution to be valid. For example, in the model described above, the automaton enforces a property stating that every open file is eventually closed by simulating the execution of the target program, from the moment any file is opened until it is eventually closed. We can apply the same property without simula-

tion, by simply inserting a "close file" action at the end of any finite sequence. If the property requires that no two files are opened at the same time, the edit automaton may modify the ordering of actions in the program (if doing so is possible given the control flow). We revisit the enforcement of this property in Chapter 5. Furthermore, the set Renewal is only a lower bound to the set of properties enforceable in a uniform context. As shown in the previous section, the set of properties enforceable by this mechanism is extended when the monitor operates in the nonuniform context, which allows non-renewal properties to be enforced.

2.4.2 Comparing Mechanisms' Enforcement power

Drawing on the work of Ligatti et. al., Chabot [21] suggests the following notation to capture the set of properties enforceable by a given automaton. Let $\{\mathcal{T}, \mathcal{D}, \mathcal{I}, \mathcal{E}, \mathcal{A}\}$ range over the possible enforcement mechanisms, namely truncation ($\mathcal{T}$), suppression ($\mathcal{D}$), insertion ($\mathcal{I}$), edition($\mathcal{E}$), or any enforcement monitor ($\mathcal{A}$), let $\mathcal{S} \subseteq \Sigma^\infty$ stand for a subset of possible execution sequences, and let $\cong$ be an equivalence relation between the sequences of Σ^∞. We write $\mathcal{A}^{\mathcal{S}}_{\cong}$-effectively enforceable to denote the set of properties which are effectively$_{\cong}$ enforceable by a monitor of class $\mathcal{A}$, when the set of possible sequences is $\mathcal{S}$ and $\cong$ is the equivalence relation limiting the monitor's ability to transform it's input. Likewise, we write $\mathcal{A}^{\mathcal{S}}$-precisely enforceable for the set of properties which are precisely enforceable by a monitor of class $\mathcal{A}$ when the set of possible sequences is $\mathcal{S}$. We omit the superscripted set of possible input sequences when it is Σ^∞, and likewise omit the monitor class when it is the edit automaton, the most powerful model. Finally, we sometimes write enforceable$^{\mathcal{S}}_{\cong}$ when the enforcement paradigm is clear from context.

For example, $\mathcal{E}^{\Sigma^\infty}_{=}$-effectively enforceable is the set of properties which are effectively enforceable by an edit automaton in the uniform context, when the equivalence relation is syntactic equality, and $\mathcal{D}^{\mathcal{S}}$-precisely enforceable is the set of properties which are precisely enforceable by the suppression automaton when the set of possible input sequences is $\mathcal{S}$.

The following theorems from [53, 11] can now be given in this notation.

Theorem 2.4.8. *$\mathcal{T}^{\Sigma^*}$-precisely enforceable = $\mathcal{I}^{\Sigma^*}$-precisely enforceable= $\mathcal{D}^{\Sigma^*}$-precisely enforceable = $\mathcal{E}^{\Sigma^*}$-precisely enforceable = Safety*

Theorem 2.4.9. *$\forall \mathcal{M} \in \{\mathcal{T}, \mathcal{D}, \mathcal{I}, \mathcal{E}\} : \exists S \subset \Sigma^* : \mathcal{M}^{\Sigma^*}$-precisely enforceable $\subset \mathcal{M}^{\Sigma^S}$-precisely enforceable.*

Observe that this result does not imply that $\forall S \subset \Sigma^* : \mathcal{M}^{\mathcal{S}}$-precisely enforceable $\subset \mathcal{M}^{\Sigma^*}$-precisely enforceable.

Theorem 2.4.10. $\forall S \subset \Sigma^*$. *Safety* $\subset \mathcal{T}^{\mathcal{S}}$*-precisely enforceable* $\subset \mathcal{D}^{\mathcal{S}}$*-precisely enforceable* $\subset \mathcal{D}^{\mathcal{S}}$*-precisely enforceable* $= \mathcal{E}^{\mathcal{S}}$*-precisely enforceable.*

Theorem 2.4.11. *Safety* $\subset \mathcal{T}^{\Sigma^*}$*-effectively enforceable* $\subset \mathcal{I}^{\Sigma^*}$*-effectively enforceable* $\wedge \mathcal{T}^{\Sigma^*}$*-effectively enforceable* $\subset \mathcal{D}^{\Sigma^*}$*-effectively enforceable.*

Theorem 2.4.12. $\mathcal{I}^{\Sigma^*}$*-effectively enforceable* $\subset \mathcal{E}^{\Sigma^*}$*-effectively enforceable* $\wedge \subset \mathcal{D}^{\Sigma^*}$*-effectively enforceable* $\subset \mathcal{E}^{\Sigma^*}$*-effectively enforceable.*

Theorem 2.4.13. $\forall \hat{\mathcal{P}} \subseteq \Sigma^* : \hat{\mathcal{P}} \in \mathcal{E}^{\Sigma^*}$*-effectively enforceable.*

2.5 Practical Enforcement Concerns

In [16, 17], Bielova et al. revisit an example of a property that is $\mathcal{E}_{=}^{\Sigma^\infty}$-effectively enforceable given by Ligatti et al. in [53]. This is the "market policy" given as follows: Let take(n) and pay(n) be two atomic actions that represent the acquisition and corresponding payment of n apples respectively. The policy states that any apple that is taken must be paid for, either before or after the acquisition takes place (take(n);pay(n) or pay(n);take(n)). In [53] Ligatti et al., produce an automaton that effectively$_=$ enforces this property, as well as a constructive proof that such a property is enforceable. As discussed above, the automaton enforces the property by suppressing the execution as long as its input is invalid, and outputting the current prefix when a valid sequence is reached.

Neither Ligatti et al. nor Bielova et al. give a formal definition of the predicate $\hat{\mathcal{P}}$ defining the market policy, but Bielova et al. observe that any reasonable definition would impose that any consecutive pair of actions of the form take(n);pay(n) or pay(n);take(n) present in the input also be present in the output, since this represents the purchase of n apples by a customer. But consider the behavior of the system when presented with the sequence: pay(1); browse; pay(2);take(2). The automaton suggested by Ligatti et al. outputs nothing, since this automaton is limited to outputting prefixes of its input sequence, which is made irremediably invalid by the presence of a pay(1) action lacking a corresponding take(1) action. Indeed, this meets the definition of effective$_=$ enforcement since ϵ is the longest valid prefix of this sequence. However, the valid transaction pay(2);take(2) is not present in the output.

What Bielova et al. realized is that the definition of effective enforcement is inadequate in that it makes no demands on the behavior of the monitor when it is presented

with an invalid input sequence, other than that it must output a valid sequence. An invalid sequence may be substituted by any valid sequence. In practice, a monitor would most likely be bounded to preserve some aspect of the original input, or limited in its ability to insert new behaviors not present in the input sequence. Bielova et al. concisely state the problem in a subsequent paper : [18]: "*What distinguishes an enforcement mechanism is not what happens when traces are good, because nothing should happen! The interesting part is how precisely bad traces are converted into good ones. To this extent soundness only says they should be made good. The practical systems, being practical, will actually take care of correcting the bad traces. But this part is simply not reflected in the current theories.*"

Bielova et al. suggest several constraints that can be imposed on the behavior of monitors for both valid and invalid inputs. This allows them to define subclasses of the edit automaton, and compare their enforcement power.

The first of these subclasses is the delayed Automaton, thus named because it only delays the appearance of input actions, until a valid prefix has been built up. Such an automaton never outputs an action not present in its input.

Definition 2.5.1. *A delayed automaton $\mathcal{A}$ is an edit automaton, as defined in definition 2.4.5, with the added restriction that*
$\forall \sigma \in \Sigma^* : \mathcal{A}(\sigma) \preceq \sigma$

The automaton defined in [11] is also limited in that at every step, the monitor either outputs all suppressed actions, or suppresses the current action and outputs nothing. In other words, this monitor never outputs only some of the actions it has previously suppressed but not yet output, and it never outputs a previously suppressed action if the current action is being suppressed. Bielova et al. refer to an automaton operating in this manner as an all-or-nothing automaton.

Definition 2.5.2. *An all-or-nothing automaton $\mathcal{A}$ is an edit automaton, as defined in definition 2.4.5, with the added restrictions that*
$\forall \sigma \in \Sigma^* : \mathcal{A}(\sigma) \preceq \sigma$ *and;*
$\forall \sigma : \forall a : \mathcal{A}(\sigma; a) = \sigma; a \vee \mathcal{A}(\sigma; a) = \mathcal{A}(\sigma)$

A third subclass of the edit automaton can be defined by imposing yet another restriction, namely that the automaton always output the longest valid prefix of its input sequence. Such an automaton is termed a *Ligatti's automaton for property* $\hat{\mathcal{P}}$ where $\hat{\mathcal{P}}$ is the property the monitor enforces. The automaton suggested in [11] to enforce the market policy is of this type.

Definition 2.5.3. *Let $\hat{\mathcal{P}}$ be a property. A Ligatti's automaton for property $\hat{\mathcal{P}}$ $\mathcal{A}$ is an edit automaton, as defined in definition 2.4.5, with the added restrictions that*
$\forall \sigma \in \Sigma^* : \mathcal{A}(\sigma) \preceq \sigma$ *and;*
$\forall \sigma \in \Sigma^* : \forall a : \mathcal{A}(\sigma; a) = \sigma; a \vee \mathcal{A}(\sigma; a) = \mathcal{A}(\sigma)$
$\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}(\sigma) \Rightarrow \mathcal{A}(\sigma) = \sigma$

A final subclass of the edit automaton can be defined by limiting the monitor to output a valid prefix of the input when the latter is valid, but leaving the monitor unconstrained otherwise (other than imposing that the output be valid). Such an automaton is called a *delayed automaton for property $\hat{\mathcal{P}}$* .

Definition 2.5.4. *Let $\hat{\mathcal{P}}$ be a property. A delayed automaton for property $\hat{\mathcal{P}}$ $\mathcal{A}$ is an edit automation, as defined in definition 2.4.5, with the added restriction that*
$\forall \sigma \in \Sigma^* : \hat{\mathcal{P}}(\sigma) \Rightarrow \mathcal{A}(\sigma) \preceq \sigma \wedge \hat{\mathcal{P}}(\mathcal{A}(\sigma))$.

In [16], Bielova et al. give an example of a delayed automaton enforcing the market policy, which can output valid transactions of the form take(n);pay(n) or pay(n);take(n) present in the input even if they are preceded by an invalid prefix.

Since Bielova et al. found the definition of effective$_{\cong}$ inadequate to enforce the market policy, they suggest an alternate notion of enforcement called *delayed precise enforcement*. This enforcement paradigm captures a requirement that the monitor must always output a valid sequence, and that if the input sequence is valid, the output must be syntactically equal to it, and furthermore, the output must always be a prefix of the input.

Definition 2.5.5. *Let Σ be a set of atomic actions. An automaton $\mathcal{A}$ delayed precisely enforces a Property $\hat{\mathcal{P}}$ iff* $\forall \sigma \in \Sigma^\infty$

1. $(q_0, \sigma) \stackrel{\sigma'}{\Longrightarrow}_{\mathcal{A}} (q', \epsilon)$;
2. $\hat{\mathcal{P}}(\sigma')$; *and*
3. $\hat{\mathcal{P}}(\sigma) \Rightarrow \sigma = \sigma' \wedge \forall i : \exists j : j \leq i \; \exists q* : (q_0, \sigma) \stackrel{\sigma[..j]}{\Longrightarrow}_{\mathcal{A}} (q*, \sigma[i+1..])$

This definition can be seen as intermediate between precise enforcement and effective$_{=}$ enforcement and captures the manner in which Ligatti et al. propose to enforce security properties in [54]. Any automaton which delayed precisely enforces a Property also effectively$_{=}$ enforces this property.

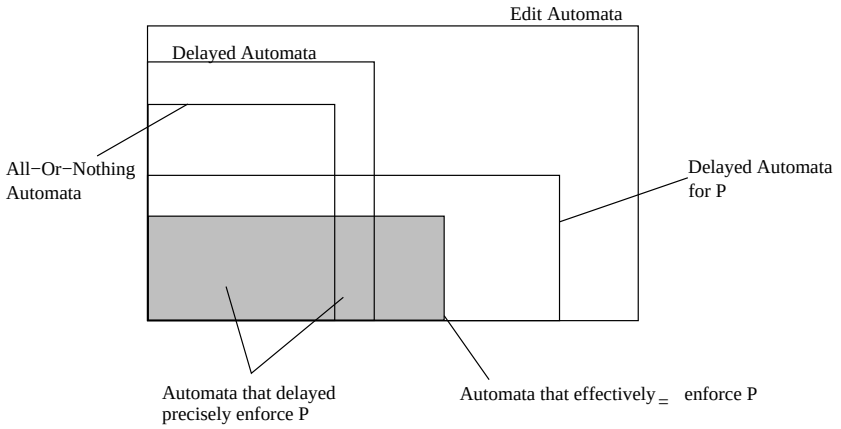


Figure 2.4: Comparing the various subclasses of the edit automata

Comparing the classes of automaton introduced above, we find that the class of Delayed automaton and Delayed automaton for property $\hat{\mathcal{P}}$ intersect, but that neither is a strict subset of the other. An all-or-nothing automaton $\mathcal{A}$ is a Delayed automaton but is not necessarily a delayed automaton for property $\hat{\mathcal{P}}$. Likewise, a delayed automaton, even if it delayed precisely enforces a property $\hat{\mathcal{P}}$ is not necessarily a Ligatti automaton for property $\hat{\mathcal{P}}$. However, the class of edit automata that effectively$_=$ enforce a property $\hat{\mathcal{P}}$ is a subclass of delayed automaton for property $\hat{\mathcal{P}}$. Furthermore, the class of edit automaton that delayed precisely enforces a Property $\hat{\mathcal{P}}$ coincides with the intersection between the class of delayed automaton, delayed automaton for property $\hat{\mathcal{P}}$ and the class of edit automaton that effectively$_=$ enforces the property $\hat{\mathcal{P}}$. Finally, an all-or-nothing automaton enforces a property $\hat{\mathcal{P}}$ iff it is also a Ligatti automaton for property $\hat{\mathcal{P}}$.

Theorem 2.5.6. *If an automaton $\mathcal{A}$ effectively$_=$ enforces a property $\hat{\mathcal{P}}$, then $\mathcal{A}$ is a delayed automaton for property $\hat{\mathcal{P}}$ but is not necessarily a delayed automaton.*

Theorem 2.5.7. *An automaton $\mathcal{A}$ delayed precisely enforces a Property $\hat{\mathcal{P}}$, iff $\mathcal{A}$ is a delayed automaton, $\mathcal{A}$ is a delayed automaton for property $\hat{\mathcal{P}}$ and $\mathcal{A}$ effectively$_=$ enforces the property $\hat{\mathcal{P}}$.*

Theorem 2.5.8. *An all-or-nothing automaton $\mathcal{A}$ precisely enforces the property $\hat{\mathcal{P}}$ iff $\mathcal{A}$ is also a Ligatti automaton for property $\hat{\mathcal{P}}$.*

These results are summarized in Figure 4.

Bielova et al. then turn their attention to monitors capable of producing an output which is not a prefix of its input. They focus on a specific class of properties termed

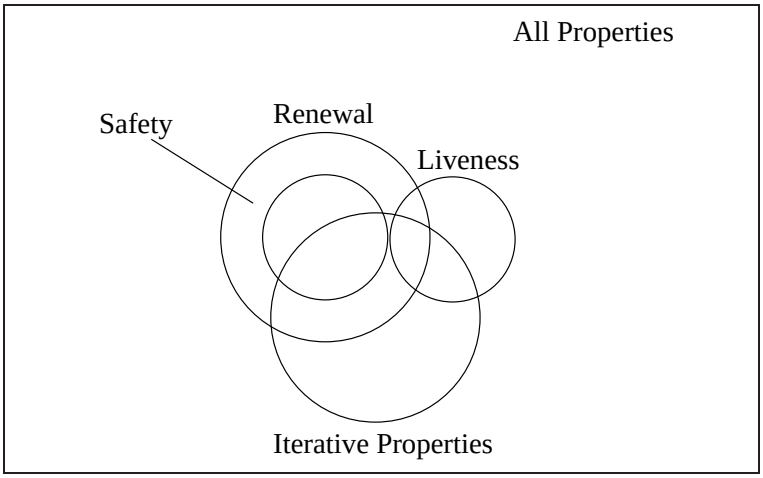


Figure 2.5: Iterative Properties

iterative properties, which model the repeated execution of transactions. We suggest an alternative definition of this idea in Chapter 4.

Definition 2.5.9. *A property* $\hat{\mathcal{P}}$ *is an iterative property iff* $\forall \sigma, \sigma' \in \Sigma^* : \hat{\mathcal{P}}(\sigma) \wedge \hat{\mathcal{P}}(\sigma') \Rightarrow \hat{\mathcal{P}}(\sigma; \sigma')$

Iterative properties serve to model the desired behavior of systems that are intended to repeatedly perform a number of finite atomic transactions. The typical example is the software of an online store or an ATM. Stack inspection is an iterative property, as is termination and access control. More generally, the class of iterative properties includes some but not all safety, liveness and Renewal properties. This is illustrated in figure 2.5.

A new notion of enforcement, tailored to the application of such iterative properties, can be explained as follows. Informally, a monitor *iteratively enforces by suppression* an iterative property $\hat{\mathcal{P}}$ iff every valid transaction is output, and every invalid transaction is suppressed. This idea is defined more formally as follows:

Definition 2.5.10. *Let* Σ *be an action set and let* $\hat{\mathcal{P}}$ *be an iterative property. Automaton* $\mathcal{A}$ *iteratively enforces by suppression property* $\hat{\mathcal{P}}$ *if:*

1. $\forall \sigma \in \Sigma^* : \hat{\mathcal{P}}(\sigma) \Rightarrow \mathcal{A}(\sigma) = \sigma$
2. $\forall \sigma \in \Sigma^* \wedge \neg\hat{\mathcal{P}}(\sigma) : \exists \sigma' \succeq \sigma : \hat{\mathcal{P}}(\sigma') \Rightarrow \mathcal{A}(\sigma) = \sigma_o$ *where* σ_o *is the longest valid prefix of* σ.

3. $\forall \sigma \in \Sigma^* : \neg\hat{\mathcal{P}}(\sigma) \wedge \forall \sigma' \succeq \sigma : \neg\hat{\mathcal{P}}(\sigma') \wedge \exists \sigma_b : \sigma = \sigma_o; \sigma_b \sigma_r \Rightarrow \mathcal{A}(\sigma) = \sigma_o; \mathcal{A}(\sigma_r)$
where σ_o is the longest valid prefix of σ and σ_b is the smallest sequence s.t. after deleting it from σ the resulting sequence is valid.

Bielova et al. show how to construct an edit automaton which iteratively enforce a transaction property $\hat{\mathcal{P}}$. Their work raises a number of interesting points. First, they have identified a shortcoming in the notion of enforcement used in most other studies : effective$_{\cong}$ enforcement does not place any restrictions on the monitor's behavior when it is faced with an invalid sequence. In practice, we may be interested in *how* a monitor deals with such a situation. This raises several new questions, such as: Which valid sequence will be chosen as a replacement? How is this new sequence chosen? Which aspect of the original invalid sequence has to be preserved, and which has to be deleted or replaced?

Answering these questions is central to the conception of practical monitoring software. Indeed, a monitor which effectively$_{\cong}$ enforces a property by always replacing any invalid sequence with the same valid sequence (perhaps ϵ) will not be useful in many critical systems where it is essential that the execution does not terminate.

The research of Bielova et al. also confirms a prior result from [11], namely that the enforcement power of monitors is intricately linked to the definition of enforcement they use. It follows that any restriction imposed on the monitor's ability to alter invalid sequences must be crafted carefully. If it is designed too narrowly, this will unduly restrict the range of enforceable properties. Conversely, if it is designed too leniently, it could result in monitors which enforce the property, but not in a manner that is desirable or useful.

The solution they advance in the case of iterative properties is ingenious and innovative. In fact, it imposes that valid transactions present in the original sequence also be present in the output, thus constraining the monitor's behavior. Yet this solution does suffer from a few drawbacks. First, the enforcement paradigm is specific to the property being enforced (iterative enforcement and iterative properties). Enforcing a different property requires a different enforcement paradigm. A more general enforcement paradigm, which is at best parameterized to fit the desired property, as effective enforcement is parameterized with an equivalence relation $\cong$, may be preferable.

Furthermore, the method of enforcement, namely the suppression of illicit transactions, is also encoded into the definition of enforcement. An automaton enforcing an iterative property by transforming invalid transactions into valid ones would not be iteratively enforcing the property. This makes it harder to compare alternative enforce-

ment paradigms of the same property. In Chapters 4 and 5, we propose two alternative security policy enforcement paradigms for monitors, which address some of the issues mentioned above.

2.6 Imposing Computability Constraints

Both Schneider and Ligatti et al. have stated that the computability constraints make a property unenforceable by a given security mechanism even if it lies inside the set of properties enforceable by this mechanism. For example, a safety property may not be enforceable by a truncation automaton because the monitor is unable to detect the violation of the security property when it occurs. As such, the results presented in the previous sections should be regarded as upper-bounds to the set of properties enforceable by each mechanism.

In [43], Kim et al. give a more precise characterization of this restriction in the case of the truncation automaton. They observe that for a monitor to be able to enforce a property, it needs to be able to identify any invalid sequence upon the inspection of a finite prefix of this sequence. It follows that only properties for which this is possible are enforceable by monitors. Formally :

Theorem 1. *A property $\hat{\mathcal{P}}$ is enforceable by a truncation automaton if and only if $\hat{\mathcal{P}}$ is a safety property and the set $\Sigma^* \backslash pref(\hat{\mathcal{P}})$ is recursively enumerable.*

In other words, a property is enforceable by a truncation automaton if it is a safety property, and the set of executions that do not respect the property is recursively enumerable. As observed in [79], this result can also be given as stating that the set of enforceable properties is the class of co-recursively enumerable properties (coRE).

Proceeding along a similar line of inquiry, Hamlen, Morisette and Schneider [38, 40] compare the set of properties enforceable by three enforcement mechanisms and link them to known classes from computational complexity theory. In order to model the (possibly infinite) execution of the target program, they introduce Program Machines (PM), deterministic state machines, akin to Turing Machines (TM) that can accept an infinite input, and exhibit an output sequence rather than accepting or rejecting this input. In keeping with previous sections of this study, we let σ, τ range over both input and output sequences, and use $M(\sigma)$ to refer to the output of PM M when it is given σ as input. We further let X_M be the set of all possible output sequences of M, and $pref(X_M)$ be that of finite prefixes of sequences in X_M.

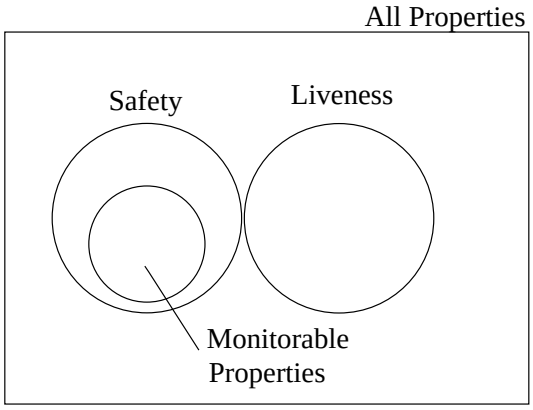


Figure 2.6: Properties Enforceable by Monitors, from [43]

Hamlen et al. first consider the case of static analysis. A property $\hat{\mathcal{P}}$ can be said to be statically enforceable if and only if there exists a TM $M_{\hat{\mathcal{P}}}$, that takes as input a PM M and either accepts or rejects in finite time, according to whether or not $\hat{\mathcal{P}}$ holds on M. This definition is both intuitive and broad enough to encompass any enforcement mechanism based on static analysis. This definition also matches exactly that of the class of recursively decidable properties, (known as the class Π_0 in the arithmetic hierarchy) for which there exists a total computable procedure that decides them. Since every such policy is also in coRE, statically enforceable properties form a subset of monitorable properties. This result is based on the premise that any static analysis could be performed by the monitor, at the onset of a program's execution.

Next, Hamlen et al. examine the class $\mathrm{RW}_{\approx}$ of properties enforceable by program rewriters, defined as a mechanism that, in infinite time, modifies an untrusted program prior to its execution in order to ensure its compliance with a security property. This category includes the more expressive monitors such as the edit automaton. Like Ligatti et al. in [11], they observe that the power of such an enforcement mechanism can only be examined in the context of a specific equivalence relation, which constraints the rewriter's ability to transform a program. The authors suggest defining an equivalence relation $\approx$ between PMs, which in turn is defined in term of an equivalence relation $\cong$ over the possible executions of the PM.

Let M_1 and M_2 be two PMs, M_1 is equivalent to M_2, noted $M_1 \approx M_2$ if and only if:

$$\forall \sigma : \sigma \in \Sigma^{\infty} : M_1(\sigma) \cong M_2(\sigma). \tag{2.6.1}$$

The following two constraints on $\cong$ are imposed:

$$M_1(\sigma) \cong M_2(\sigma) \text{ is recursively decidable if } M_1(\sigma) \text{ and } M_2(\sigma) \text{ are finite} \tag{2.6.2}$$

$$\sigma \cong \sigma' \Rightarrow (\forall i \in \mathbb{N} : \exists j \in \mathbb{N} : \sigma[..i] \cong \sigma'[..j]) \tag{2.6.3}$$

The first requirement ensures that a procedure be able to determine whether or not two executions are equivalent. The latter imposes that equivalent executions have equivalent prefixes. Given a certain equivalence relation $\approx$, property $\hat{\mathcal{P}}$ can be enforceable by rewriting if there exists a total computable rewriting function, $R \subseteq \text{PM} \times \text{PM}$ that can transform any PM into a new valid (with respect to $\hat{\mathcal{P}}$) PM, while preserving the semantics of every valid PM. Formally :

$$\hat{\mathcal{P}}(R(M)) \tag{2.6.4}$$

$$\hat{\mathcal{P}}(M) \Rightarrow M \approx R(M) \tag{2.6.5}$$

This definition is equivalent to that of effective$_{\cong}$ enforcement given in section 2.4. Since no restriction is imposed on the PM's output in those cases where the execution is invalid, every satisfiable statically enforceable property is trivially RW$_{\approx}$ enforceable. We can determine statically which programs respect this property, and the rewriting relation R has only to return the program unchanged if it respects the property, or some safe execution otherwise. Only the unsatisfiable property, which rejects all executions, is statically enforceable property but cannot be enforced by program rewriters in this manner, since there is no valid execution which can be substituted for the invalid ones. Policies that are RW$_{\approx}$ enforceable also include some but not all properties in coRE and some properties not in this class.

Turning their attention back to the properties enforceable by monitoring, Hamlen et al. argue that the class coRE actually represents an upper-bound to the set of properties enforceable by this mechanism. Note first that when a monitor detects a violation of the security property, it must react to prevent this violation. In their model, this intervention is modeled by a sequence of actions (possibly the end of program token) that the monitor appends to the current input sequence. Let I be the set of all possible interventions on the monitor's part. The property $\hat{\mathcal{P}}_I$, which forbids all such interventions cannot be enforced by a monitor. Yet this property is in coRE if I is a computable set. For example, the non termination property $\hat{\mathcal{P}}_{end}$, which demands that the target program's execution does not terminate, is not enforceable by a truncation automaton, since the only remedial course at the disposal of such a monitor is explicitly forbidden.

Even more remarkably, the authors observe that the same property can become enforceable or not by monitors depending on how the predicate $\hat{\mathcal{P}}$ that characterizes

this property is stated. For example, consider the property P_{RU} which states that a resource cannot be used (by the action e_{use}) after it has been released (by the action e_{rel}). This property can be stated as :

$$\hat{\mathcal{P}}_{RU1}(\sigma) = (\forall i : i \geq 1| : e_{rel} \notin \sigma[..i] \vee e_{use} \notin \sigma[i+1..]).$$

This predicate states that no e_{use} can occur after a e_{rel} action. This property can be enforced by a suppression automaton, by suppressing the occurrence of e_{use}, or by a truncation automaton, by aborting the execution.

Alternatively, this same property can be given as :

$$\hat{\mathcal{P}}_{RU2}(\sigma) = (e_{rel} \notin \sigma \vee (\forall \sigma'.\sigma; \sigma' \in \Sigma^{\infty}| : e_{use} \notin \sigma')).$$

This predicate states that any execution containing a e_{rel} action cannot be extended to include a e_{use} action. Both are the same property, in the sense that both define the same set of execution. While this is a safety property in coRE, the question of deciding whether or not a finite execution will be extended to respect $\hat{\mathcal{P}}_{RU2}$ is undecidable. This is because, while both predicates rule out the same executions, $\hat{\mathcal{P}}_{RU2}$ actually defines a violation of the property in any execution by an event that occurs earlier than $\hat{\mathcal{P}}_{RU1}$. A truncation mechanism which relies upon the definition of $\hat{\mathcal{P}}_{RU1}$ is thus unable to prevent some violations of the policy from occurring.

A better characterization of the set of properties enforceable by monitors actually is the intersection of the class coRE and the set $\mathrm{RW}_{\approx}$. Properties in this intersection exhibit a particular behavior termed benevolence. A property is benevolent if there exists a decision procedure $M_{\hat{\mathcal{P}}}$ that rejects any invalid prefix of an invalid execution, but accepts any valid prefix of a valid execution, for all possible PM M. This allows the monitor to reject valid prefixes if it determines that they will be extended to invalid executions. Observe that the monitor is not required to accept the valid prefixes of invalid executions. Formally, a property $\hat{\mathcal{P}}$ is benevolent if there exists a decision procedure $M_{\hat{\mathcal{P}}}$ such that for all PM M :

$$\neg(\forall \sigma \in X_M : \hat{\mathcal{P}}(\sigma)) \Rightarrow (\forall \sigma \in \mathit{pref}(X_M) : (\neg\hat{\mathcal{P}}(\sigma) \Rightarrow M_{\hat{\mathcal{P}}}(\sigma)\text{rejects})) \tag{2.6.6}$$

$$(\forall \sigma \in X_M : \hat{\mathcal{P}}(\sigma)) \Rightarrow (\forall \sigma \in \mathit{pref}(X_M) : (M_{\hat{\mathcal{P}}}(\sigma)\text{accepts})) \tag{2.6.7}$$

$\hat{\mathcal{P}}_{RU1}$ is an example of a benevolent property.

Properties that do not lie in the intersection of coRE and RW do not have the benevolence property, and thus cannot be meaningfully enforced by monitors. In the absence of a decision procedure $M_{\hat{\mathcal{P}}}$ that fits the description above, the monitor is required to either reject some valid executions, or to only reject a violation of the security property after it has occurred. This result squares with our intuition that monitors can be inlined in the program they aim to protect, so that any property enforceable by monitor can also be enforced by program rewriting.

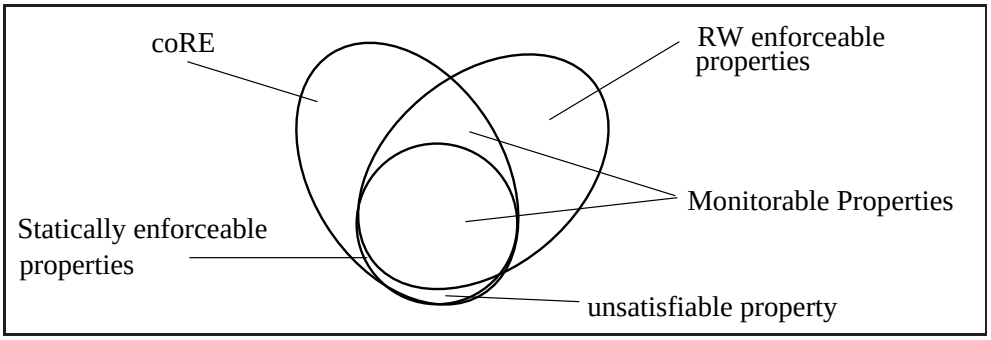


Figure 2.7: Properties enforceable by various mechanisms, from[40]

A summary of the findings of Schneider et al. is given in Figure 2.7. As discussed earlier, any statically enforceable property is seen as being also enforceable by monitoring and program rewriting, under the assumption that a monitor can perform a static analysis of its target before its execution begins. Note however that this presupposes that both the static and the dynamic analyzer have access to the same information about the program's behavior. Static analyzers often have access to source code unavailable to monitors, which could allow them to enforce a greater range of properties. The set of properties enforceable by monitors is defined as the intersection of coRE properties and $\text{RW}_{\approx}$, properties. Properties in the intersection of these classes exhibit an interesting behavior, termed benevolence, which allows the monitor to reject all invalid behaviors before they occur. This result indicates that the class of monitorable properties is somewhat smaller than previous studies have shown, an inconstancy that is explained by the fact that such previous studies considered that a monitor can enforce some properties even though the monitor only discovered a violation of the property after it had occured and thus outputs an invalid sequence. Finally the set of properties enforceable by program rewriting is shown to be a superset of that of properties enforceable by monitoring, owing to the rewriters greater ability to transform the target program. This is consistent with the conclusion of Ligatti et al. [11] who argue that the edit automaton, which can alter a program's semantics in the manner of a rewriter, is strictly more powerful that an truncation or insertion automaton, which lacks this ability.

2.7 Monitoring with Memory Constraints

Among the assumptions made previously when studying the enforcement power of monitors is the assumption that they are unconstrained by any memory limitations. In practice however, it is reasonable to assume that monitors only have a finite amount of computational resources at their disposal. Several authors have thus examined the set of properties enforceable by a monitor whose finite memory allows it to keep only a partial record of its target ongoing execution. Despite this limitation, memory-bounded automata where shown to be capable of enforcing several interesting real-life properties.

2.7.1 Shallow History Automata

The study of memory-bounded automata and their use in monitoring was pioneered by Fong [36]. He introduces the shallow history automaton (SHA), which only records the shallow history i.e. the unordered set of security relevant events performed by the target program. Formally, a shallow automaton is a tuple $\langle \Sigma, F(\Sigma), H_0, \delta \rangle$ where :

- Σ is a finite or countably infinite set of events;
- $F(\Sigma)$ is the set of all possible shallow histories;
- $H_0 \in F(\Sigma)$ is an initial access history, usually $\emptyset$;
- $\delta : F(\Sigma) \times \Sigma \to F(\Sigma)$ is a transition function, which is defined as $\delta(H, a) = H \cup \{a\}$ if δ is defined for $\langle H, a \rangle$.

The set of properties enforceable by SHA, known as $\mathrm{EM}_{\mathrm{SHA}}$, is a strict subset of the set EM of properties enforceable by Schneider's security automaton. Yet, several interesting real-life properties can be enforced under this paradigm. Fong gives the following 4 examples :

The Chinese Wall Policy The Chinese Wall Policy [57] seeks to prevent conflicts of interests. It specifies, for example, that a consultant cannot advise a client if he also serves as an advisor to competitors of this client. The Chinese Wall property is defined in terms of sets of data objects O, and subjects S, and a set of conflicts of interests which maps each data objects with a list of objects that conflict with it. Once a subject has accessed a given object, he is restricted from accessing any

object that conflicts with it. In order to enforce this policy, a monitor only needs to keep track of the set of objects that each subject has previously accessed, with no regard for the sequencing in which these accesses have occurred. This allows the property to be enforced by a SHA, which aborts the execution if a conflicting data access is attempted. In chapter 5, we will suggest alternative enforcement paradigms for this same property, that allows more valid data accesses to be output.

Low-Water-Mark Policy The low-water-mark policy is a data access policy formulated by Biba [15]. The model defines a set S of subjects, a set O of objects, and a set L of trustworthiness levels, partially ordered by the binary relation $\leq$. The trustworthiness level of an object is a permanent classification. That of users is monotonously decreased each time an object is read, so that the trustworthiness level of the user becomes the greatest lower bound between its previous level and that of the object it reads. A subject is only allowed to write to an object or execute another subject if its trustworthiness level is greater or equal to that of its target. Once again for this policy to be enforceable, the monitor needs only to keep track of which access events have occurred, with no regard for their sequencing. A subjects's label can be straightforwardly computed from an unordered list of such events and an access event aborted by the monitor if its trustworthiness level is deemed inadequate.

One-out-of-k authorization One-out-of-k authorization [28] is a safety paradigm whereas applications are classified into equivalence classes based on the tasks they perform, and are then granted access rights inherent to each class. For example, an application that attempts to open a socket will be classified as a browser and allowed network actions, while a program that attempts to access user files will be classified as an editor and given the right to access user and temporary files. Applications are permanently assigned to a class according to their runtime behavior. This property can be easily enforced by a SHA, in the following manner : let Σ be the set of all possible accesses, and let each application class i be defined by a set $C_i \subseteq \Sigma$. A SHA whose transition function is defined so that $\langle H, a\rangle$ is defined iff $H \cup \{a\} \subseteq C_i$ for some i.

Assured Pipelines The Assured Pipelines policy [19, 81] seeks to ensure the integrity of data objects that are processed by several procedures. Let O be a set of data objects and S a set of procedures. $\Sigma = S \times O$ thus defines the set of access events and each event $\langle s, o\rangle$ corresponds to the application of procedure s to data object o. An instance of an assured pipeline policy is specified by an enabling relation $e \subseteq S \times S$, with the following restriction that e must define an acyclic graph. The presence of a pair $\langle s, s'\rangle$ in e indicates that any action of the form $\langle s', o\rangle$ is only allowed if s was the last process that accessed o. Because of the restriction

that e must be an acyclic graph, each action $\langle s', o\rangle$ can occur at most once. It follows that a monitor SHA can enforce this property simply by examining the unordered access events that have already occurred during the execution and contrasting them with e.

Fong further suggests that other meaningful subclasses of EM could be formed by abstraction, i.e. by merging several states of a SA into a single abstract state, making them indistinguishable from the perspective of policy enforcement. Let A be a set of abstract states and $\alpha : \Sigma^* \to A$ be an abstraction function. An automaton whose states are built from applying the abstraction function α to the states of a security automaton SA is noted SA_α. A property is enforceable by a SA_α if it can be enforced using only information left behind after the abstraction process. We write EM_α for the set of properties enforceable by SA_α. The SHA, as well as the set of properties it can enforce, can be seen as an instance of this abstraction process, generated by a particular abstraction function. Every abstraction α induces an equivalence relation $\equiv_\alpha$ such that $\forall \sigma, \sigma' \in \Sigma^* : \sigma \equiv_\alpha \sigma' \Leftrightarrow \alpha(\sigma) = \alpha(\sigma')$.

The use of equivalence relations provides an intuitive comparison point between the subsets of EM. Let $\mathrm{EM}_\equiv$ be the set of properties enforceable by $\equiv$. Fong shows that more properties are enforceable by a monitor bounded by an equivalence relation $\equiv_1$ than one bounded by $\equiv_2$ if $\equiv_1$ is more differentiating than $\equiv_2$.

Theorem 2.7.1. *Let $\equiv_1$, $\equiv_2$ be equivalence relations. $\equiv_1 \subseteq \equiv_2 \Rightarrow EM_{\equiv_1} \subseteq EM_{\equiv_2}$ Furthermore, if the former inclusion is strict, so is the latter.*

This comparison also argues for a lattice based classification of the subclasses of EM defined by abstraction. Let the join operator be the intersection of two equivalence relations ($\equiv_1 \sqcup \equiv_2 = \equiv_1 \cap \equiv_2$), and meet the transitive closure of the union of two equivalence relations ($\equiv_1 \sqcap \equiv_2 = (\equiv_1 \cup \equiv_2)^+$). These two operators define a lattice over the set of all equivalence relations over a given set of actions Σ^*. The top element of this lattice, ($\equiv_\top$) corresponds to the class EM, and the bottom element ($\equiv_\bot$), corresponds to the class of memoryless properties enforceable by an automaton with a single state. The classes of properties enforceable by a SHA and by any other subclass of EM are ordered on a poset induced by the above lattice classification.

The following are corollaries of theorem 2.7.1.

Corollary 2. $EM_\bot \subset EM_{SHA} \subset EM_\top$

Let $\mathcal{SHA}^{\Sigma^*}$-enforceable stand for the set of properties enforceable by a Shallow history automaton. Fong shows that this set is a strict subset of the set of properties

enforceable by an unbounded truncation automaton.

Corollary 3. *$\mathcal{SHA}^{\Sigma^*}$-enforceable $\subset$ $\mathcal{T}^{\Sigma^*}$-precisely enforceable.*

2.7.2 Bounded History Automata

Fong was the first to examine the enforcement power of monitors operating with information constraints. These constraints were modeled by abstracting the states of a security automaton enforcing safety properties. Following this line of enquiry, and linking it to Ligatti's research on the edit automata, Talhi, Tawbi and Debbabi [76, 74, 75] devised the Bounded History Automaton (BHA) which has only a limited space with which to store the program's execution history and showed how this new automata class can still enforce a relevant set of properties despite this limitation. They further showed how bounded-memory monitors can be modeled both by Bounded Security Automata (BSA) or by Bounded Edit Automata (BEA), analogous to the security and edit automata introduced by Schneider [69] and Ligatti[11] respectively.

To better reason about bounded histories, Talhi et al. introduce the following notation : Let Σ be a finite or countably infinite alphabet of atomic actions. Σ_k is the subset of sequences of length k while $\Sigma_{\leq k}$ is the subset of finite sequences of Σ of length less than k, ($\Sigma_k = \{\sigma \in \Sigma^* : |\sigma| = k\}$ and $\Sigma_{\leq k} = \{\sigma \in \Sigma^* : |\sigma| \leq k\}$). $Pref(\sigma)$ is the set of all prefixes of a sequence σ, $Suf(\sigma)$ that of the set of all suffixes. $Pref_k(\sigma)$ $Suf_k(\sigma)$ denote the sets of prefixes and suffixes of lengths k respectively and the set of k-length factors of σ is given as $Fact_k(\sigma) = \{\sigma' \in \Sigma_k | \exists \sigma'' \in \Sigma^*. \exists \sigma''' \in \Sigma^\infty : \sigma = \sigma''; \sigma'; \sigma'''\}$.

Bounded history automaton (BHA) is a class of automata used to model security properties enforceable by monitors equipped with only a bounded memory with which to store the past history of the target execution. Each state of the automaton thus represents a finite-size abstraction of the input sequence that has occurred so far. Talhi et al. define two subclasses of BHA: The Bounded Security Automaton, which is analogous to a truncation automaton, and can only allow an action or abort the execution, and the Bounded edit Automaton, analogously to an edit automaton, which can also suppress and insert actions into the input stream.

Definition 2.7.2. *A Bounded Security Automaton of bound k (noted k-BSA), is a tuple $\langle \Sigma, Q = \Sigma_{\leq k}, q_0, \delta \rangle$ where :*

- *Σ is a finite or countably infinite set of input actions.*

- *Q is a finite or countably infinite set of states, each of which represent a bounded history of size smaller or equal to k.*
- *k is the maximal size of the history.*
- *$q_0 \in Q$ is the initial state. This is usually the empty history ϵ.*
- *$\delta : (Q \times \Sigma) \to Q$ is the (possibly partial) transition function. If, for a given automaton A, $\delta(h, a) = h'$, this signifies that the bounded history h' is an abstraction of $h; a$, and that h' contains all the information necessary for the enforcement of the property captured by A.*

Analogously to an edit automaton, a BHA can be enriched with the ability to insert or suppress actions in the programs they monitor. The history stored by a bounded edit automata consists of two sequences; the first has been output by the automaton, while the second is suppressed in order to reinsert it if a valid prefix is recognized.

Definition 2.7.3. *A Bounded edit automaton of bound k (noted k-BEA), is a tuple of the form $\langle \Sigma, Q = (\Sigma_{\leq k}, \Sigma_{\leq k}) \leq k, q_0, \delta \rangle$ where :*

- *Σ is a finite or countably infinite set of input actions.*
- *Q is a finite or countably infinite set of states, each of which is a pair $\langle \sigma_{Acc}, \sigma_{Sup} \rangle$ such that $\sigma_{Acc}, \sigma_{Sup} \in \Sigma_{\leq k}$.*
- *k is the maximal size of the history.*
- *$q_0 \in Q$ is the initial state. This is usually the empty history. $\langle \epsilon, \epsilon \rangle$.*
- *$\delta : (Q \times \Sigma) \to Q$ is the (possibly partial) transition function. As with the BSA, if $\delta(h, a) = h'$ then h' is an abstraction of $h; a$ that contains all the relevant information necessary for the enforcement of the property accepted by the BEA.*

Talhi et al. prove a number of interesting theorems related to the expressivity of this enforcement mechanism. Let EM_{kSA} be the set of properties enforceable by BSA, and EM_{kEA} be the set of properties enforceable by BEA. EM_{kSA} is naturally a subset of safety properties, enforceable by unbounded SAs, and EM_{kEA} a subset of reasonable infinite renewal properties, enforceable by unbounded EAs. The set of properties enforceable by bounded automata increase monotonously as a larger memory is made available to to the monitor. Formally :

Theorem 2.7.4. *For any two integers k and k' such that $k < k'$, we have $EM_{kSA} \subset EM_{k'SA}$ and $EM_{kEA} \subset EM_{k'EA}$.*

BSA are equally expressive as the SA_α defined by Fong. For any BSA, there exists a SA_α, and conversely, for any SA_α, there exists a k-bound BEA with some maximal memory size k that enforces the same property. Furthermore, any SHA can be translated into a k-bound BEA where k is the cardinality of the set Σ of input actions, provided that Σ is finished.

Theorem 2.7.5. *For any BSA enforcing a Property $\hat{\mathcal{P}}$ there exists a SA_α that also enforces $\hat{\mathcal{P}}$.*

Theorem 2.7.6. *For any SA_α, enforcing a Property $\hat{\mathcal{P}}$ there exists a k-bound BSA $\mathcal{A}$ enforcing the same property, where k is the maximal size of the histories of $\mathcal{A}$.*

Theorem 2.7.7. *Let $\mathcal{A} = \langle \Sigma, F(\Sigma), H_0, \delta \rangle$ be a SHA enforcing a Property $\hat{\mathcal{P}}$, if Σ is a finite set and $|\Sigma| = k$, then there exists a k-bound BSA that enforces the same property.*

There exists a close connection between the set of properties enforceable by BHA and a class of properties termed locally testable properties (or local properties)[13]. Locally testable properties are properties that are recognizable by a class of automata, termed scanners, which are equipped with a finite memory and a sliding window of size k. This window is slid along the input sequence starting with the initial action, with only the actions in the window being visible to the scanner at any given time. Properties are enforceable by scanners if they can be recognized using only the prefixes and suffixes of length less than k and the factors (subsequences) of length k. In [46], it is observed that such properties are particularly well suited to be enforced by monitoring because a monitor enforcing such a property needs only to keep a record of the last k computational cycles. Furthermore, if a monitor enforces several such properties, this record can be shared, so that the memory overhead of the monitor is independent of the number of locally testable properties being enforced.

Let P, S, X and F be 4 sets, such that $P, S \subseteq \Sigma_{k-1}$, $X \subseteq \Sigma_{\leq k-1}$ and $F \subseteq \Sigma_k$.

Definition 2.7.8. *A property L is k-locally testable if and only if it respects the following two rules[76]:*

- $\forall \sigma \in \Sigma^*.\sigma \in L \backslash \{\epsilon\}| : (\sigma \in X) \vee ((Pref_{k-1}(\sigma) \cap P \neq \emptyset) \wedge (Suf_{k-1}(\sigma) \cap S \neq \emptyset) \wedge (Fact_k(\sigma) \subseteq F))$.
- $\forall \sigma \in \Sigma^\omega.\sigma \in L \backslash \{\epsilon\}| : \forall \sigma' \in Pref(\sigma) : \exists \sigma'' \in Pref(\sigma).\sigma' \in Pref(\sigma'') \wedge \sigma'' \in L$.

Informally, a property $\hat{\mathcal{P}}$ is k-locally testable if any finite non-empty sequence $\in \hat{\mathcal{P}}$ is either of size less than k or if it has both a prefix and a suffix in the sets P and S

respectively, and that all their factors of size k are in the set F. Taken together, these requirements mean that a scanner examining this property through a sliding window of size k will always be able to recognize that the sequence under consideration is in the $\hat{\mathcal{P}}$.

A property L is locally testable if it is k-locally testable for some integer k. There exists several algorithms for deciding if a language is locally testable, and finding the minimal k value for which it is k-locally testable [78, 58, 45, 44]. These algorithms are polynomial in time with respect to the size of the language's alphabet and that of the automaton used to represent the language.

The set of locally testable properties intersects with that of prefix testable, suffix testable, prefix-suffix testable, testable properties, and contains a subset of strongly locally testable properties [65]. Prefix testable properties are properties that can be recognized by examining only prefixes of the input sequences, and conversely suffix testable properties are recognizable by examining only suffices of the input sequences. Prefix-suffix testable properties are recognizable by examining both prefixes and suffixes of the input sequences. Each of these classes contains a subset of locally testable properties, the k-prefix, k-suffix and k-prefix-suffix testable properties, which can be recognized by examining only a bounded prefix, suffix or a prefix and a suffix of size k. Finally, strongly locally testable properties are a subset of locally testable properties which are recognizable by inspecting factors of a bounded size.

The connection between locally testable properties and properties enforceable by BHA is as follows. Since the BSA is a subclass of the SA, properties enforceable by the BSA will necessarily be a subset of those enforceable by SAs, and thus will be prefix closed. Prefix-closed k-prefix are enforceable by k-bounded BSA, so are k-strongly enforceable properties and locally testable properties in general if they are prefix-closed. Conversely, suffix testable and prefix-suffix testable properties are generally not enforceable by BSA.

Theorem 2.7.9. *Any prefix-closed k-prefix testable property $\hat{\mathcal{P}}$ is enforceable by some k-BSA.*

Theorem 2.7.10. *Any k-strongly locally testable property $\hat{\mathcal{P}}$ is enforceable by some k-BSA.*

Theorem 2.7.11. *Any prefix-closed k-locally testable property $\hat{\mathcal{P}}$ is enforceable by some k-BSA.*

Theorem 2.7.12. *Any suffix testable and prefix-suffix testable properties are not enforceable by BSA.*

The more powerful BEA naturally enforces strictly more properties. A k-bounded BEA can enforce any k-prefix testable property, any k-strongly enforceable properties and any k-locally testable properties. Suffix testable and prefix-suffix testable properties are generally not enforceable by BEA.

Theorem 2.7.13. *Any prefix-closed k-prefix testable property $\hat{\mathcal{P}}$ is enforceable by some k-BEA.*

Theorem 2.7.14. *Any k-strongly locally testable property $\hat{\mathcal{P}}$ is enforceable by some k-BEA.*

Theorem 2.7.15. *Any k locally testable property $\hat{\mathcal{P}}$ is enforceable by some k-BEA.*

Theorem 2.7.16. *Any suffix testable and prefix-suffix testable properties are not enforceable by BSA.*

Talhi, Tawbi and Debbabi's contribution in this context is three fold. First, they devise a new abstract model of monitors, the Bounded History Automaton, and explore its enforcement power. Second, they devise a new taxonomy of enforceable properties, based on the amount of memory needed for the enforcement. Thirdly, they show a connection between locally testable properties and those enforceable by BHA. Their research allows us to better characterize the properties enforceable by monitors constrained by memory limitations. Since real-life monitors will certainly be constrained in this way, their study offers relevant insights on the enforcement power of monitors.

2.7.3 Finite Automaton

A final enquiry in the set properties enforceable by an edit automaton with memory constraints was done by Beauquier et. al. In [12], they examine the set of properties enforceable by an edit automaton with a finite, but unbounded, set of states. They focus on effective enforcement and on uniform systems containing both finite and infinite sequences, with $=$ as the equivalence relation, and define a new class of properties, termed memory bounded properties, which coincides with the set of properties that are effectively$_=$ enforceable by a finite edit automaton.

Before presenting the result of this study, we introduce the specific notation that was used by the authors in this regard. Let Σ be a set of atomic actions and $\hat{\mathcal{P}} \subseteq \Sigma^\infty$ be a property. We write $\hat{\mathcal{P}}_{fin}$ for $\hat{\mathcal{P}} \cap \Sigma^*$ and $\hat{\mathcal{P}}_{inf}$ for $\hat{\mathcal{P}} \cap \Sigma^\omega$. $\overrightarrow{\hat{\mathcal{P}}_{fin}}$ designates the set of infinite sequences which have infinitely many prefixes in $\hat{\mathcal{P}}_{fin}$.

Definition 2.7.17. *A property $\hat{\mathcal{P}}$ is memory bounded if $\hat{\mathcal{P}}$ is of the form $\hat{\mathcal{P}}_{fin} \cup \overrightarrow{\hat{\mathcal{P}}_{fin}} \bigcup_{j \in J} R_j; \beta_j$ where*

- $\epsilon \in \hat{\mathcal{P}}$;
- J *is finite;*
- $\hat{\mathcal{P}}_{fin}$ *is regular;*
- $\forall j \in J : R_j$ *is regular;*
- $\forall j \in J : \beta_j$ *is an ultimately periodic sequence in* $Sigma^{\omega}$;
- $\forall j \in J : R_j \cap Pref(\hat{\mathcal{P}}_{fin}) = \emptyset$;
- $\forall i, j \in J : i \neq j = R_j \cap Pref(R_i) = \emptyset$
- *there exists a constant K such that for every $u \prec v$ s.t. $u \notin Pref(\hat{\mathcal{P}}_{fin})$ and $v \in R_j |v| - |u| \preceq K$*

We can now state the main result from [12].

Theorem 2.7.18. *Let $\mathcal{F}_{=}^{\Sigma^{\infty}}$-effectively enforceable stand for the set of properties that are effectively enforceable by a finite edit automaton in a uniform context, with $=$ as the equivalence relation. A Property $\hat{\mathcal{P}}$ is in $\mathcal{F}_{=}^{\Sigma^{\infty}}$-effectively enforceable iff $\hat{\mathcal{P}}$ is memory bounded.*

2.8 In-lining a monitor into a Program

The final question we examine in this chapter is how to inline a monitor into a possibly untrusted program, in order to ensure its compliance with the property that it captures. This process also raises other interesting issues of optimizing the resulting code and proving the correctness of the in-lining process. A wide variety of implementations of monitors has been suggested in the literature. We focus here on automata-based approach, since this is the framework we have used in all other sections.

2.8.1 SASI

This issue was first addressed by Erlingsson and Schneider in [31], where they introduce the SASI monitoring framework. The idea behind this framework is to inject a monitor, modeled by a security automaton, into an untrusted object code.

To inline the security automaton into the target program, a special segment of code that simulates the automaton's behavior is added before each instruction. The current state of the automaton is captured in state variables that can only be modified by the added code. Before any instruction can be executed, this code updates the state variables to reflect the state that the automaton reaches after taking the appropriate transition from its current state. If no transition is defined from the current state for the next instruction, the execution is aborted.

By using this method, we can be sure that the target code respects the security policy since we do not perform a single program instruction without verifying that doing so will not violate the security property. Yet the runtime cost incurred is substantial since several text instructions must be performed for each program instruction. To reduce this cost, the inlining of the target program and the security automaton is done in four steps, in a manner designed to reduce overhead. These steps are :

Insert security automata The security automaton is inserted before each instruction.

Evaluate transitions Static analysis is used to identify which states are reachable from each program point, and thus identify for each program point which transitions will necessarily be taken, (labeled as true) and which cannot be taken (labeled as false).

Simplify Automaton Any transition labeled as false is deleted.

Compile Automaton The remaining security automaton is translated into a code which is then added at the appropriate program point to simulate the behavior of the automaton.

Two prototypes of SASI were developed for x86 assembly language and Java. The x86 prototype is comparable in performance to Misfit [71], a specialized tool for code rewriting. Likewise, the JAVA version of SASI can simulate the functionalities of the Java Security Manager (SM) with no statistically significant difference in overhead. Yet in both these cases the set of policies enforceable by SASI is a superset of those

enforceable by those of Misfit or SM. This shows SASI to be both as efficient as some specialized tools and highly polyvalent.

2.8.2 Colcombet and Fradet's Approach

While Erlingsson and Schneider propose to inline a security automaton into object code, Colcombet and Fradet apply the same operation on source code [26]. Although similar to the preceding approach, the most salient aspect of their approach is the manner in which the runtime overhead is systematically minimized by a combination of static analysis and optimizations. The result is a code that provably respects the desired security property with a minimal amount of runtime checks.

Their approach consists in seven steps.

Property Encoding The first step is the choice of the desired security policy. As was the case with [31], the user is limited to safety properties, since the chosen policy must be translated into a security automaton.

Program Annotation The second step is to define a function E that associates relevant program instructions with events. A program instruction is considered relevant if it can affect the validity of the security property. Each instruction must be associated with exactly one event. All actions judged irrelevant are associated with the dummy event $\star$.

Program Abstraction The next step consists in constructing a conservative abstraction of the target program. The authors settle on a control-flow graph whose nodes represent program points and whose edges represent instructions (abstracted into events) but stress that a more precise abstraction could be used.

Direct Instrumentation The fourth step consists in adding to the graph the runtime checks that will make it impossible to accept a trace that violates the security property. This is done by constructing a new structure called an I-graph. The I-graph is built from the control flow graph by adding a set of states, some of which are marked as accepting, and a transition function. These states correspond to the states of the security automaton with which we want to constrain the program's behavior, and the transition function mimics the automaton's behavior. If the transition function dictates that the execution must enter a non-accepting state, the execution is halted. This step results in a structure that can be translated into a program which always respects the desired property, but with a very high

added runtime overhead incurred by systematically updating the state after each relevant program action and then verifying if that update places the automaton in a non-accepting state. The purpose of the next two steps is thus to reduce this added overhead.

Minimization This step seeks to identify and remove the states that are unreachable for each program point. This operation is analogous to the one made in [31] to reduce the added runtime cost incurred by instrumenting the program. An iterative algorithm is used to compute the set of reachable states. A state v is reachable at a given node if there exists a sequence of events ω which, when fed to the transition function from the initial node r_0 reaches v.

The I-graph can then be further simplified by merging several states together at certain nodes. This can be done for any two states r_1 and r_2 if the same set of traces can be generated from a node v whether it is in state r_1 or in r_2 . This is similar to standard automaton minimization. After this step is performed, the transition function links equivalent sets of states in each node to their successors in another node, but the semantics of the I-graph remains unchanged.

Erasing The I-graph's transition function is translated into dynamic tests, which in turn are the source of the added runtime. It is thus essential to reduce the number of tests. This is done by erasing those transitions for which the current state remains unchanged after the event occurs. As was the case in the preceding step the semantics of the target program are preserved thoughtout this step.

Concretization The last step in the instrumentation process is to convert the optimized I-graph into code. While this step is dependent on the target language we wish to use, the task is made easier by the fact that the I-graph is still very similar to a control flow graph. Its nodes are program points, its edges are instructions. All that remains is to add a program variable in order to store the current state and a series of assignations and tests on that variable in order to model the evolution of the current state during the execution.

2.8.3 Automata Injection

In [63, 62], Ould-Slimane et al. give a formalization of the in-lining process. Their method is based on the idea of automata composition, using a new operator, $\odot_g^f$ which embeds an automata representing a property into another representing a program (modeled as a labeled transition system). Their approach distinguishes itself from other monitoring frameworks presented in this chapter in that the transitions contain pairs or predicates and actions, rather than simply actions. Each predicate is a condition

that must be respected by the action for the transition to be taken. Otherwise, the execution is aborted. These predicates allow the insertion into the target program of runtime tests which simulate the monitor's behavior.

The $\odot$ operator is parameterized with two functions, f, g that help define the monitor's behavior in certain instances. f inserts the runtime test on the automaton's transitions and g determines the accepting states of the automaton. In [63] and [62], Ould-Slimane et al. give two possible examples of such functions, both of which apply to a truncation automaton. The enforcement using an edit automaton is discussed in [64].

Definition 2.8.1. *Let* $\mathcal{A}_\infty = (Q_1, \Sigma, \delta_2, q1_0, F_1)$ *and* $\mathcal{A}_\in = (Q_2, \Sigma, \delta_2, q2_0, F_2)$ *be two automata as defined in definition 2.3.1. The injection modulo functions (f,g) of* $\mathcal{A}_\infty$ *into* $\mathcal{A}_\in$*, denoted* $\mathcal{A}_\infty \odot_g^f \mathcal{A}_\in$ *is the automaton* $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ *where*

- Q *is the smallest set s.t.*
  - $(q1_0, q2_0) \in Q$;
  - *if* $(s_1, s_2) \in Q$ *and* $s'_1 = \delta_1(s_1, a_1)$ *and* $s'_2 = \delta_2(s_2, a_2)$ *then* $(s'_1, s'_2) \in Q$
- δ *is defined by way of function* f *as follows :*
  - $\delta : Q \times \Sigma \to Q$
  - $\delta((s_i, s_{i'}), f(a_i, a_{i'})) = (s_j, s_{j'})$ *where* $s_j = \delta_1(s_i, a_i)$ *and* $s'_{j'} = \delta_2(s_{i'}, a_{i'})$
- $q_0 = (q1_0, q2_0)$
- F *is defined as by way of function* $g : Q \times \Sigma \times \delta(q_0) \to F$*, which is given as input to the composition operator.*

This method is proved to be sound (the resulting program always respect the desired property) and complete (every valid execution present in the original program is preserved). Contrasting the three implementations presented in this section with the theoretical studies of monitors presented in the preceding sections, we come to the following conclusions. First, all such implementations of monitors (except that of [64]) are limited to the enforcement of safety properties, despite the fact that it was shown in [11] and other papers that monitors can, in many cases, enforce a greater range of security policies. For instance, none of the approaches presented here makes use of static analysis to extend the range of enforceable properties even though the needed abstraction is available, and is used to reduce the number of runtime tests added by the in-lining process. In this thesis, we propose a new inlining procedure which draws upon

a static model of the target program to enforce some nonsafety properties. Finally, Ould-Slimane et al. state that the instrumented program produced by the in-lining process is equivalent to the original program, but do not specify the precise notion of equivalence they use. Since it was shown in [11] that the set of enforceable properties is greatly dependent on the choice of the equivalence relation, a more formal characterization could have been beneficial. We explore some possible equivalence relations and their use in monitoring in Chapter 4, and an alternative to equivalence relations, preorders, in Chapter 5.

2.9 Other Work in Monitoring

In parallel to the main issues discussed so far that are central to our thesis, the question of how to construct a monitor from an automaton representing a security property, has been addressed several times in the literature. A constructive proof that it is possible for all properties over finite sequences is given is [11]. An alternative algorithm is given in [18].

In [35, 34, 32, 33], Falcone et al. show that the class of Renewal properties proposed by Ligatti et al. is equivalent to the union of five of the six classes of the safety-progress classification of properties [24], namely safety, guarantee, obligation, response and persistence, with the class of reactivity properties containing properties which are not effectively$_=$enforceable by the edit automaton. This classification is an alternative to the safety-progress dichotomy, in which properties are arranged in a hierarchy of six classes. They further give algorithms to construct a monitor from a property for each of the 5 classes for which this is possible.

While the automata based model of monitors presented throughout this chapter is the most widely used in the literature, other models have been proposed. In [82], Zhu et al. suggest modeling monitors by way of a *stream automata* while another alternative model, the *Mandatory results Automata* is suggested by Ligatti et al. in [56]. Both of these models differ from the edit automaton in that they distinguish between the action set of the target and that of the system it interacts with. These models made it easier to study the interaction between the target program, the monitor and the system. In this study, while we continue to focus on the truncation and edit automaton, we believe the results we present can easily be applied to the more refined automata models introduced in these papers. In [39], Jun models monitors as Mealy Machines [59] and study their expressive power.

In [7, 10] Bauer et al. study the set of monitorable properties using a slightly different definition of monitoring, in which a monitor is tasked only with detecting and reporting the occurrence of a violation of the security property. They are thus sequence recognizers, as defined by Schneider, rather than sequence transformers, as defined by Ligatti. The monitor thus incrementally examines each prefix τ of an ongoing execution w.r.t. a three-valued variant of LTL [66] or TLTL [68] and returns one of the three following possible results :

$\top$ If every possible extension of τ satisfies the desired property. In [47], such a prefix is termed a *good prefix* for this property.

$\bot$ If every possible extension of τ violates the property. In this case, τ is a *bad prefix* for this property [47].

? otherwise.

The properties enforceable in this context are characterized by the absence of a prefix υ, for which there is no $\nu \in \Sigma^*$ s.t. $\upsilon;\nu$ is either a good prefix or a bad prefix for the property being monitored. In this case, υ is called an *ugly prefix* for this property. The set of monitorable properties thus includes all safety properties, for which every invalid execution has a bad prefix, that of co-safety properties [47], for which every valid execution has a good prefix, and some other properties which are neither safety nor co-safety. The authors further show how to construct a monitor that is both minimal (w.r.t. its size) and optimal, in that it detects a good or bad prefix as early as possible. This result, and its implications, are expounded in more detail in [4].

In [8, 9], the authors extend this framework into a four valued semantics, which distinguishes between a possibly true and possibly false values instead of just "?". This allows the monitor to distinguish between a sequence which will not respect the property unless some future event occurs, from one which would respect the property if it ends on the next step.

In [27], an algorithm is given to extract a monitor which detects the minimal bad prefix of a property stated as a Büchi automaton. This method does not monitor the liveness component of a property.

Other monitoring frameworks are more specific with regard to the systems or the properties they can be used to enforce. The monitoring of security protocols is discussed in [5, 6]. That of information flow policies is discussed in numerous papers including [25] and [37]. In [70] Sen at al. propose a decentralized monitor which monitors safety

properties in distributed programs. The optimization of monitors is further discussed in [80]. The in-lining of monitors in concurrent programs is discussed in [51]. An algebraic method to inline a safety property into a program is given in [49, 52]. In this approach, both the property and the program are stated using process algebra. The instrumented program is shown to be equivalent to the original one using a notion of equivalence based on bisimulation. The monitoring of networks is discussed in [60].

A more exhaustive list of all implementations of formal monitor is given in [54].

2.10 Conclusion

This chapter has presented an overview of the main results of studies on runtime software monitoring, which seek to delimitate the set of policies applicable by such monitors.

The analysis shows that monitors can be used to enforce a wide range of interesting security policies. The most basic monitors can be shown to enforce only a subset of safety policies. However, we have identified three situations where this range can be extended, namely : (1) if the monitor knows which executions paths are possible and which are not (or has an approximation of this information), (2) if the monitor has more means at its disposal to react to a potential security policy violation (rather than simply aborting the execution), and (3) if the monitor is not tightly bound to respect the semantics of a valid execution, but can instead transform its input w.r.t. some equivalence relations between sequences. We explore this idea in Chapters 4 and 5.

However, all the implementations of monitors which we have surveyed are limited to a narrow class of security properties, namely safety properties, and on a simple enforcement mechanism based on truncation. A greater range of security properties can be enforced if the monitor could rely on a static analysis of the target program. In the next chapter, we propose a new method to inline a truncation monitor into a target program in order to enforce non-safety properties.

Chapter 3

Generating In-Line Monitors Based on Static Analysis

3.1 Introduction

As was discussed in the previous chapter, studies on security policy enforcement mechanisms show that an a priori knowledge of the target program's behavior increases the power of these mechanisms [11, 40]. However, most practical implementations of monitors do not take advantage of this possibility and restrict themselves to enforcing safety properties. Furthermore the needed abstraction is often already available, and is used to minimize the runtime overhead incurred by the property monitoring process.

In this chapter, we present an approach to generate a safe instrumented program, from a security policy and an untrusted program in which the monitor draws on an a priori knowledge of the program's possible behavior. The policy is stated as a deterministic Rabin automaton, a model which can recognize the same class of languages as non deterministic Büchi automata [65].

In our framework a program execution may be of infinite length, representing the executions of programs such as daemons or servers. Finite executions are made infinite by attaching at their end an infinite repetition of a void action. The use of Rabin automaton is motivated by the need for determinism in order to simplify our method and the associated proofs.

Our approach draws on advances in discrete events system control by [67] and on

related subsequent research by Langar and Mejri [50] and consists in combining two models via the automata product operator: a model representing the system's behavior and another one representing the property to be enforced. In our approach, the model representing the system's behavior is represented by a LTS and the property to be enforced is stated as a Rabin automaton. The LTS representing the program could be built directly from the control flow graph after a control flow analysis [1, 14].

To sum up, our approach either returns an instrumented program, modeled as a labeled transition system, which provably respects the input security policy or terminates with an error message. While the latter case sometimes occurs, it is important to stress that this will never occur if the desired property is a safety property which can be enforced using existing approaches. Our approach is thus strictly more expressive.

The rest of this chapter is organized as follows. In Section 2, we define some concepts that are used throughout the chapter. The elaborated method is presented in Section 3. In Section 4, we discuss the theoretical underpinnings of our method. Some concluding remarks are finally drawn in Section 5 together with an outline of possible future work.

The research has been accomplished in collaboration with H. Chabot [21], and the results have been presented at the 14th Nordic Conference on Secure IT Systems and published in the conference's proceedings [22] and have been accepted for publication in a forthcoming edition of the journal Computers & Security [23].

3.2 Preliminaries

Before moving on, let us briefly review some preliminary definitions.

We express the desired security property as a Rabin automaton.

Definition 3.2.1. *A Rabin automaton $\mathcal{R}$, over alphabet Σ is a tuple (Q, q_0, δ, C) such that*

- *Σ is a finite or countably infinite set of symbols;*
- *Q is a finite set of states;*
- *$q_0 \in Q$ is the initial state;*
- *$\delta : Q \times A \to Q$ is a transition function;*

- $C = \{(L_j, U_j) | j \in J\}$ *is the acceptance set. It is a set of couples* (L_j, U_j) *where* $L_j \subseteq Q$ *and* $U_j \subseteq Q$ *for all* $j \in J$ *and* $J \subseteq \mathbb{N}$.

Let $\mathcal{R}$ stand for a Rabin automaton defined over alphabet Σ. A subset $Q' \subseteq Q$ is *admissible* if and only if there exists a $j \in J$ such that $Q' \cap L_j = \emptyset$ and $Q' \cap U_j \neq \emptyset$.

We refer to the elements defining an automaton or a model following this formalism: the set of states Q of automaton $\mathcal{R}$ is referred to as $\mathcal{R}.Q$ and we leave it as Q when it is clear in the context.

A *path* π, is a finite (respectively infinite) sequence of states $\langle q_1, q_2, ..., q_n \rangle$ (respectively $\langle q_1, q_2, ... \rangle$) such that there exists a finite (respectively infinite) sequence of symbols $a_1, a_2, ..., a_n$ (respectively $a_1, a_2, ...$) called the label of π such that $\delta(q_i, a_i) = q_{i+1}$ for all $i \in \{0, ..., n\}$ (respectively $i \geq 0$). In fact, a path is a sequence of states consisting of a possible run of the automaton, and the label of this path is the input sequence that generates this run. A path is said to be empty if its label is the empty sequence ϵ.

Let $set(\pi)$ denote the set of states visited by the path π. The first state of π is called the origin of π. If π is finite, the last state it visits is called its end; otherwise, if it is infinite, we write $inf(\pi)$ for the set of states that are visited infinitely often in π. A path π is *initial* if and only if its origin is q_0, the initial state of the automaton, and it is *final* if and only if it is infinite and $inf(\pi)$ is admissible.

A path is *successful* if and only if it is both initial and final. A sequence is *accepted* by a Rabin automaton *iff* it is the label of a *successful* path.

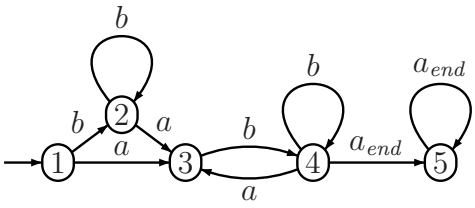
The set of all accepted sequences of $\mathcal{R}$ is the language recognized by $\mathcal{R}$, noted $\mathcal{L}_\mathcal{R}$.

Let $q \in Q$ be a state of $\mathcal{R}$. We say that q is reachable *iff* there exists an initial path (possibly the empty path) that visits q. We say that q is co-reachable *iff* it is the origin of a final path.

Recall from Section 2.2 that executions are modeled as sequences of atomic actions taken from a finite or countably infinite set of actions Σ. The empty sequence is noted ϵ, the set of all finite length sequences is noted Σ^*, that of all infinite length sequences is noted Σ^ω, and the set of all possible sequences is noted $\Sigma^\infty = \Sigma^\omega \cup \Sigma^*$. Let $\tau \in \Sigma^*$ and $\sigma \in \Sigma^\infty$ be two sequences of actions. We write $\tau; \sigma$ for the concatenation of τ and σ. We say that τ is a prefix of σ noted $\tau \preceq \sigma$ *iff* $\tau \in \Sigma^*$ and there exists a sequence σ' such that $\tau; \sigma' = \sigma$.

Let $a \in \Sigma$ be an action symbol. A state $q' \in Q$ is an $a-$successor of q if $\delta(q, a) = q'$. Furthermore, a state q' is a successor of q if there exists a symbol a such that $\delta(q, a) = q'$.

Let $\pi = \langle q_1, q_2, ..., q_n \rangle$ be a finite path in $\mathcal{R}$. This path is a cycle if $q_1 = q_n$. The cycle π is admissible *iff* $set(\pi)$ is admissible. It is reachable *iff* there is a state q in $set(\pi)$ such that q is reachable, and likewise, it is co-reachable *iff* there is a state q in $set(\pi)$ such that q is co-reachable.



$$C = \{(\{3\}, \{4\}), (\emptyset, \{5\})\}$$

Figure 3.1: A Rabin Automaton with acceptance Condition C

Let us consider Figure 3.1. It represents a Rabin automaton. In this figure, all the states are reachable and co-reachable. The paths $\langle 3, 4, 3, 4, 3 \rangle$, $\langle 3, 4, 3 \rangle$ and $\langle 2, 2 \rangle$ are inadmissible cycles, while $\langle 5, 5 \rangle$ is an admissible cycle and both infinite paths $\langle 1, 2, 3, 4, 5, 5, \ldots \rangle$ and $\langle 1, 2, 3, 4, 3, 4, 4, \ldots \rangle$ are initial and final and therefore both are successful.

3.3 Method

In this section we explain our approach and illustrate it with an example. The main algorithm takes as input a Rabin automaton $\mathcal{R}$, which represents a security Policy $\hat{\mathcal{P}}$ and a labeled transition system (LTS) $\mathcal{M}$, which models a program. The algorithm either returns a model of an instrumented program that enforces $\hat{\mathcal{P}}$ on $\mathcal{M}$ or returns an error message. The latter case occurs when it is not possible to produce an instrumented program that both enforces the desired security property and generates all valid sequences of $\mathcal{M}$.

Following [30, 40, 53], we consider that an enforcement mechanism successfully enforces the property if the two following conditions are satisfied. First, the enforcement mechanism must be transparent, meaning that all possible program executions that respect the property must be emitted, i.e. the enforcement mechanism cannot prevent the execution of a sequence satisfying the property. Second, the enforcement mechanism

must be sound, meaning that it must ensure that all observable outputs respects the property. We revisit and expand these ideas in Sections 3.3.3 and 3.4. We illustrate each step of our approach using an example program and a security policy.

3.3.1 Property Encoding

As mentioned earlier, the desired security property is stated as a Rabin automaton. The security property $\hat{\mathcal{P}}$ to which we seek to conform the target program is modeled by the Rabin automaton in Figure 3.1, over the alphabet $\Sigma \cup \{a_{end}\}$ with $\Sigma = \{a, b\}$. The symbol a_{end} is a special token added to Σ to capture the end of a finite sequence, since the Rabin automaton only accepts infinite length sequences. Any finite sequence σ which we desire to include in the security property is thus modeled as $\sigma; (a_{end})^{\omega}$. The language accepted by this automaton is the set of executions that contains only a finite non-empty number of a actions and such that finite executions end with a b action.

For the sake of simplicity, if a sequence $\sigma = \tau; (a_{end})^{\omega}$ with $\tau \in \Sigma^*$ is such that $\hat{\mathcal{P}}(\sigma)$ we say that $\hat{\mathcal{P}}(\tau)$.

3.3.2 Program Abstraction

The program is abstracted as a labeled transition system (LTS). This is a conservative abstraction, widely used in model checking and static analysis, in which a program is abstracted as a graph, whose nodes represent program points, and whose edges are labeled with instructions (or abstractions of instructions, or actions).

Definition 3.3.1. *A labeled transition system $\mathcal{M}$, over alphabet Σ is a deterministic graph (Q, q_0, δ) such that:*

- *Σ is a finite or countably infinite set of actions;*
- *Q is a finite set of states;*
- *q_0 is the initial state;*
- *$\delta : Q \times \Sigma \to Q$ is a transition function. For each $q \in Q$, there must be at least one $a \in \Sigma$ for which $\delta(q, a)$ is defined.*

Here also a finite sequence $\sigma \in \Sigma^*$ is extended with the suffix $(a_{end})^\omega$ yielding the infinite sequence $\sigma; (a_{end})^\omega$.

In general, static analysis tools do not always generate deterministic LTSs. Yet, this restriction can be imposed with no loss of generality. Indeed, a non-deterministic LTS $\mathcal{M}$ over alphabet Σ can be represented by an equivalent deterministic LTS $\mathcal{M}'$ over alphabet $\Sigma \times \mathbb{N}$, which is equivalent to $\mathcal{M}$ (in the sense that it accepts the same language) if we ignore the numbers $i \in \mathbb{N}$ associated with the actions. Each occurrence of an action a is associated with a unique index in $\mathbb{N}$ so as to distinguish it from other occurrences of the same action a. In what follows, we can thus consider only deterministic LTSs. Furthermore, we focus exclusively on infinite length executions.

The example program that we use to illustrate our approach is modeled by the LTS in Figure 3.2, over the alphabet Σ. The issue consisting of how to abstract a program into a LTS is beyond the scope of this study.

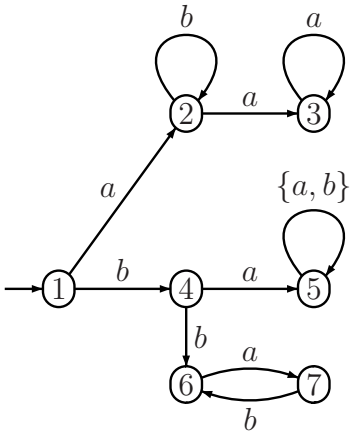


Figure 3.2: Example- Labeled transition system

As with the Rabin Automata, we define a *path* π as a finite or infinite sequence of states $\langle q_1, q_2, \ldots \rangle$ such that there exists a corresponding sequence of actions $(a_1, a_2 \ldots)$ called the label of π, for which the $\delta(q_i, a_i) = q_{i+1}$ for $i \geq 1$.

The set of all labels of infinite paths starting in q_0 is the language generated or emitted by $\mathcal{M}$ and is noted $\mathcal{L}_\mathcal{M}$. This set includes all possible executions of the program. More precisely it represents a superset of all possible executions since the analysis that constructs an LTS from a program is conservative and some of the paths thus do not represent actual possible executions.

3.3.3 Algorithm

In this section, we give a brief overview of our algorithm, which is detailed in Appendix A.1. The algorithm's input consists of the program model $\mathcal{M}$ and a Rabin automaton $\mathcal{R}$ which encodes the property. The output is a instrumented program behaving in the same manner as the original program monitored by a truncation automaton $\mathcal{T}$ representing a model of an inlined monitored program acting exactly identically to the input program for all the executions satisfying the property and halting a bad execution after producing a valid prefix of this execution.

A high level description of the algorithm is as follows:

1. Build a product automaton $\mathcal{R}^P$ whose recognized language is exactly : $\mathcal{L}_{\mathcal{R}^P} = \mathcal{L}_{\mathcal{R}} \cap \mathcal{L}_{\mathcal{M}}$.
2. Build $\mathcal{R}^T$ from $\mathcal{R}^P$ by the application of a transformation allowing it to accept partial executions of the program modeled by $\mathcal{M}$ that satisfy the property $\hat{\mathcal{P}}$.
3. Check if $\mathcal{R}^T$ could be used as a truncation automaton and produce a LTS $\mathcal{T}$ modeling the program instrumented by a truncation mechanism otherwise produce **error**.

The following sections give more details on each step.

Automata Product

The first phase of the transformation is to construct $\mathcal{R}^P$, a Rabin automaton that accepts the intersection of the language accepted by the automaton $\mathcal{R}$ and the language emitted by $\mathcal{M}$. This is exactly the product of these two automata. Thus $\mathcal{R}^P$ accepts the set of executions that both respect the property and represent executions of the target program.

Given a property automaton $\mathcal{R} = (\mathcal{R}.Q, \mathcal{R}.q_0, \mathcal{R}.\delta, \mathcal{R}.C)$ and a LTS $\mathcal{M} = (\mathcal{M}.Q, \mathcal{M}.q_0, \mathcal{M}.\delta)$ the automaton $\mathcal{R}^P$ is constructed as follows:

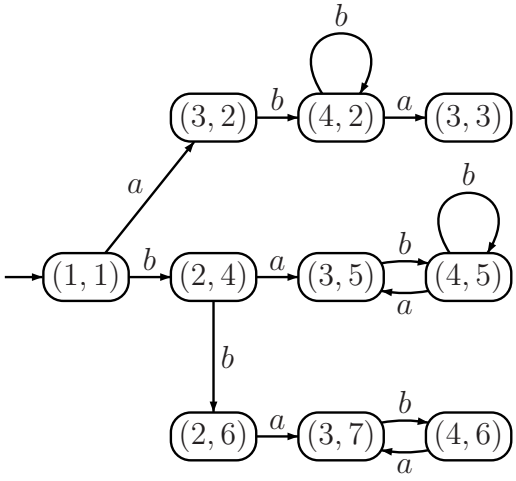
- $\mathcal{R}^P.Q = \mathcal{R}.Q \times \mathcal{M}.Q$
- $\mathcal{R}^P.q_0 = (\mathcal{R}.q_0, \mathcal{M}.q_0)$

- $\forall q \in R.Q, q' \in \mathcal{M}.Q \wedge a \in (\Sigma \cup \{a_{end}\})$

$$\mathcal{R}^P.\delta((q,q'),a) = \begin{cases} (\mathcal{R}.\delta(q,a), \mathcal{M}.\delta(q',a)) & \text{if } \mathcal{R}.\delta(q,a) \text{ and } \mathcal{M}.\delta(q',a) \\ & \text{are defined} \\ \\ \text{undefined} & \text{otherwise} \end{cases}$$

- $\mathcal{R}^P.C = \bigcup_{(L,U)\in\mathcal{R}.C}\{(L \times \mathcal{M}.Q, U \times \mathcal{M}.Q)\}$

The automaton built for our example using the property in Figure 3.1 and the program model presented in Figure 3.2 is given in Figure 3.3. Note that only reachable states are shown in the Figures and listed in the acceptance condition.



$$C = \{(\ \{(3,2),(3,3),(3,5),(3,7)\},\ \{(4,2),(4,5),(4,6)\}\ )\}$$

Figure 3.3: Example - Rabin automaton $\mathcal{R}^P$

Since $\mathcal{R}^P$ accepts the intersection of the languages accepted by the automaton $\mathcal{R}$ and $\mathcal{M}$, it would seem an ideal abstraction from which to build the instrumented program. However, there is no known way to transform such an automaton into a program. Indeed, since the acceptance condition of the Rabin automaton is built around the notion of infinite traces reaching some states infinitely often, a dynamic monitoring system built from such an automaton with no help provided by a prior static analysis, may never be able to determine if a given execution is valid or not.

Instead, we extract a deterministic automaton, $\mathcal{T} = (\mathcal{T}.Q, \mathcal{T}.q_0, \mathcal{T}.\delta)$, from the Rabin automaton $\mathcal{R}^P$. This automaton is the labeled transition system which is returned. It forms in turn the basis of the instrumented program we seek to construct. The instrumented program is expected to work as a program monitored by a truncation automaton meaning that its model $\mathcal{T}$ has to satisfy the following conditions: (1) $\mathcal{T}$ emits each execution of $\mathcal{M}$ satisfying the security property without any modification, (2) for each execution that does not satisfy the property, $\mathcal{T}$ safely halts it after producing a valid partial execution, and (3) $\mathcal{T}$ does not emit anything else apart from those executions described in (1) and (2).

The next step toward this goal is to apply a transformation that allows $\mathcal{R}^P$ to accept partial executions of $\mathcal{M}$ which satisfy the property. Indeed, all finite initial paths in $\mathcal{R}^P$ represent partial executions of $\mathcal{M}$, only some of them satisfy the security property. We add a transition, labeled a_{halt}, to a new state h to every state in R^P where the execution could be aborted after producing a partial execution satisfying the property, i.e. a state (q_1, q_2) for which $\mathcal{R}.\delta(q_1, a_{end})$ is defined. The state h is made admissible by adding the transition (h, a_{halt}, h) to the set of transitions and the pair $(\emptyset, \{h\})$ to the acceptance set. We have to be careful in choosing h and a_{halt} so that $h \notin \mathcal{R}.Q \cup \mathcal{M}.Q$ and $a_{halt} \notin \Sigma$ the alphabet of actions.

We refer to this updated version of $\mathcal{R}^P$ as $\mathcal{R}^T$, built from $\mathcal{R}^P$ as follows :

- $\mathcal{R}^T.Q = \mathcal{R}^P.Q \cup \{h\}$
- $\mathcal{R}^T.q_0 = \mathcal{R}^P.q_0$
- $\mathcal{R}^T.\delta = \mathcal{R}^P.\delta \cup \{(q, a_{halt}, h) | \mathcal{R}^P.\delta(q, a_{end}) \text{ is defined } \} \cup \{(h, a_{halt}, h)\}$.
- $\mathcal{R}^T.C = \mathcal{R}^P.C \cup \{(\emptyset, \{h\})\}$

After this transformation, our example product automaton becomes the automaton depicted in Figure 3.4. The halt state h has been duplicated three times in order to avoid cross edging.

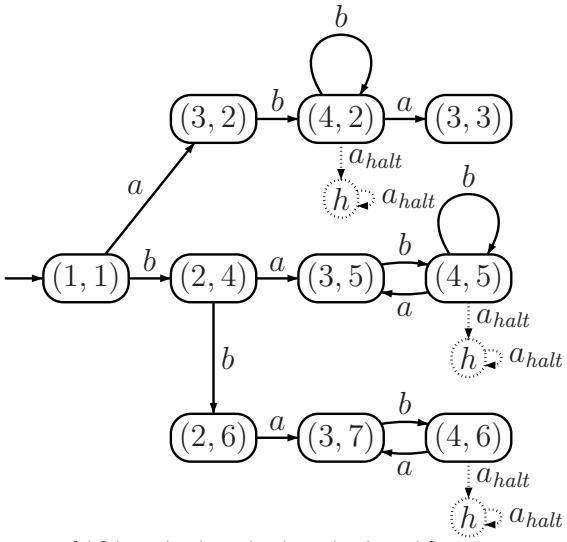


Figure 3.4: Transformed Product Automaton

The language recognized by $\mathcal{R}^T$ is

$$\mathcal{L}_{\mathcal{R}^T} = (\mathcal{L}_{\mathcal{R}} \cap \mathcal{L}_{\mathcal{M}}) \cup \{\tau; (a_{halt})^{\omega} | (\tau \in \Sigma^*) \wedge (\exists \sigma \in \mathcal{L}_{\mathcal{M}} : \tau \preceq \sigma) \wedge (\tau; (a_{end})^{\omega} \in \mathcal{L}_{\mathcal{R}})\}.$$

Extracting a Model of the Instrumented Program

The next phase consists in extracting, if possible, a labeled transition system $\mathcal{T} = (Q, q_0, \delta)$, from the Rabin automaton $\mathcal{R}^T$. This automaton is expected to behave as the original program monitored by a truncation automaton, as define din section 2.4.

To understand the need for this step, first note that the acceptance condition of a Rabin automaton could not be checked dynamically due to its infinite nature. Should we build an instrumented program directly from $\mathcal{R}^T$, by ignoring its acceptance condition, and treating it like a simple LTS, the resulting program would still generate all traces of $\mathcal{M}$ that verify the property $\hat{\mathcal{P}}$ but it would also generate the invalid sequences of $\mathcal{M}$ representing labels of infinite paths in $\mathcal{R}^T$ trapped in non admissible cycles. In other words, the enforcement of the property would be transparent but not sound.

In order to generate $\mathcal{T}$, we prune $\mathcal{R}^T$ of some of its states and transitions, eliminating inadmissible cycles while taking care to preserve the ability to generate all the valid sequences of $\mathcal{L}_\mathcal{M}$. Furthermore, we need to ascertain that $\mathcal{T}$ aborts the execution of every sequence of $\mathcal{L}_\mathcal{M}$ not satisfying $\hat{\mathcal{P}}$ and that $\mathcal{T}$ generates only executions satisfying $\hat{\mathcal{P}}$.

We can now restate the correctness requirements of our approach. In the formulation of these requirements, the actions a_{end} and a_{halt} are ignored, as they merely model the end of a finite sequence.

$$(\forall \sigma \in \mathcal{L}_\mathcal{M} | : (\exists \tau \in \mathcal{L}_\mathcal{T} | : (\tau \preccurlyeq \sigma) \wedge \hat{\mathcal{P}}(\tau) \wedge (\hat{\mathcal{P}}(\sigma) \Longrightarrow (\tau = \sigma)))) \tag{3.3.1}$$

$$\forall \tau \in \mathcal{L}_\mathcal{T} | : ((\exists \sigma \in \mathcal{L}_\mathcal{M} | : ((\tau = \sigma) \vee (\tau \preccurlyeq \sigma))) \wedge \hat{\mathcal{P}}(\tau) \tag{3.3.2}$$

Note that the requirements 3.3.1 and 3.3.2 are not only sufficient to ensure the respect of soundness and transparency requirements introduced at the beginning of Section 3.3 following [30, 40, 53], but also that of a more restrictive requirement. Indeed, requirement 3.3.1 also states that the mechanism is a truncation mechanism. It ensures the compliance to the security property by aborting the execution before a security violation occurs whenever this is needed. We can thus prove that for any invalid sequence present in the original model, the instrumented program outputs a valid prefix of that sequence.

Our enforcement mechanism is not allowed to generate sequences that are not related to sequences in $\mathcal{L}_\mathcal{M}$ either by equality or prefix relation. Furthermore these sequences must satisfy $\hat{\mathcal{P}}$. This is stated in requirement 3.3.2.

Requirements 3.3.1 and 3.3.2 give the guidelines for constructing $\mathcal{T}$ from $\mathcal{R}^T$. The transformations that are performed on $\mathcal{R}^T$ to ensure meeting these requirements are elaborated around the following intuition. The automaton $\mathcal{R}^T$ has to be pruned so as to ensure that it represents a safety property even though $\mathcal{R}$ is not. Note that this is not possible in the general case without violating the requirements. The idea is that admissible cycles are visited infinitely often by executions satisfying $\hat{\mathcal{P}}$ and must thus be included in $\mathcal{T}$. Likewise, any other state or transition that can reach an admissible cycle may be part of such an execution and must be included. On the other hand, inadmissible cycles cannot be included in $\mathcal{T}$ as the property is violated by any trace that goes through such a cycle infinitely often. In some cases their elimination cannot occur without the loss of transparency and our approach fails, returning **error**. The underlying idea of the subsequent manipulation is thus to check whether we can trim $\mathcal{R}^T$

by removing bad cycles but without also removing the states and transitions required to ensure transparency.

The following steps show how we perform the trim procedure.

The next step is to determine the strongly connected components (*scc*) in the graph representing $\mathcal{R}^T$ using Tarjan's algorithm [77]. We then examine each *scc* and mark it as containing either only admissible cycles, only inadmissible cycles, both types of cycles, or no cycles (in the trivial case).

The next step is to construct the quotient graph of $\mathcal{R}^T$ in which each node represents a *scc* and an edge connecting two *scc* c_1 and c_2 indicates that there exists a state q_1 in *scc* c_1 and a state q_2 in *scc* c_2 and an action a such that $\mathcal{R}^T.\delta(q_1, a) = q_2$. We assume, without loss of generality, that all the *scc* states are reachable from the initial node, the *scc* containing q_0.

The nodes of the quotient graph $\mathcal{R}^T$ are then visited in reverse topological ordering. We determine for each one whether it should be kept intact, altered or removed.

In what follows the *scc* containing the halting state h is referred to as H.

A *scc* with no cycle at all is removed with its incident edges if it cannot reach another *scc*. In Figure 3.4 the *scc* consisting of the state $(3, 3)$ is thus eliminated.

A *scc* containing only admissible cycles should be kept, since all the executions reaching it satisfy $\hat{\mathcal{P}}$. Eliminating it would prevent the enforcement mechanism from being transparent. In our example in Figure 3.4 the *scc* consisting of the single state $(4, 2)$ has only admissible cycles and should be kept.

A *scc* containing only non admissible cycles can be removed if it cannot reach another *scc* with admissible cycles. Otherwise, we are generally forced to return **error**. However, in some cases, we can either break the inadmissible cycles or prevent them from reaching H by removing some transitions and keeping the remainder of the *scc*. This occurs when the only successor, having admissible cycles, of this *scc* is H. In our example, the *scc* containing the states $(3, 7)$ and $(4, 6)$ has only non admissible cycles and H is its only successor. We can eliminate this *scc* and halt with **error** at this point. Yet, if we observe that eliminating the transition $((4, 6), a, (3, 7))$ would break the inadmissible cycle, we can eliminate that transition and keep the rest of the *scc*.

A transition can only be removed if its origin has h as immediate successor. This is

because, should the instrumented program attempt to perform the action that corresponds to this transition, its execution would be aborted. However, a partial execution only satisfies the property if it ends in a state that has h as an immediate successor.

A *scc* containing admissible and non admissible cycles may cause good or bad behavior. Actually, an execution reaching this *scc* may be trapped in an inadmissible cycle forever or may leave it to reach an admissible cycle thus satisfying the property $\hat{\mathcal{P}}$. We have no means to dynamically check whether the execution is going to leave a cycle or not. Thus, in this example, it is not possible to enforce the desired property on this program and the application of the algorithm yields **error**. In the example given in Figure 3.4 the *scc* consisting of the two states $(3,5)$ and $(4,5)$ has one admissible cycle, $\langle(4,5),(4,5)\rangle$ and one inadmissible cycle $\langle(3,5),(4,5),(3,5)\rangle$. This last cycle is visited if the invalid sequence $(ba)^\omega$ is generated. Note that the automaton accepts an infinite number of valid traces of the form $ba(ba)^*b^\omega$, and that no truncation automaton can both accept these traces and reject the invalid trace described above. Hence we have to abort the algorithm with **error** in such cases.

After removing all the *scc* with inadmissible cycles and provided the algorithm did not abort, we can be sure that an instrumented program built from $\mathcal{T}$ would not contain any infinite length execution which does not respect the security property. We must still verify that whenever the execution is halted, the partial sequence emitted satisfies $\hat{\mathcal{P}}$.

The last step is to check whether the eliminated states and transitions could not allow invalid partial executions to be emitted. This verification is based on the following observation: if a removed transition has an origin state that is not an immediate predecessor of h this would then allow to emit a partial execution that does not satisfy $\hat{\mathcal{P}}$. Hence, the verification merely consists in checking whether we have removed transitions from states that are not immediate predecessors of h; if such is the case we have to abort with **error**. More precisely, for a state $q = (q_1, q_2)$ in $\mathcal{T}$ we have to check whether it is possible from q_2 in $\mathcal{M}$ to perform actions that are not possible from q; if this is the case, q must have h as immediate successor; otherwise, we have no other option than to terminate the algorithm without returning a suitable LTS and with an error message.

We may also remove the transitions of the form (h, a_{halt}, h) and (q, a_{end}, q), where $q \in \mathcal{R}^T.Q$.

3.3.4 Additional Example

Throughout this section, we have illustrated each step of our approach with an example that was carefully crafted to highlight some of the behavior our algorithm may encounter when in-lining a monitor into a target program. As some aspects of the behavior lead to the rejection of the target program, we were unable to show, using this example, the final result of our approach. We thus introduce in this section another example in which the approach succeeds and returns a model of an instrumented program that verifies the property. The most striking feature of this example is the fact that the property being enforced is not a safety property and, as such, cannot possibly be enforced under existing implementations on formal in-line monitoring frameworks.

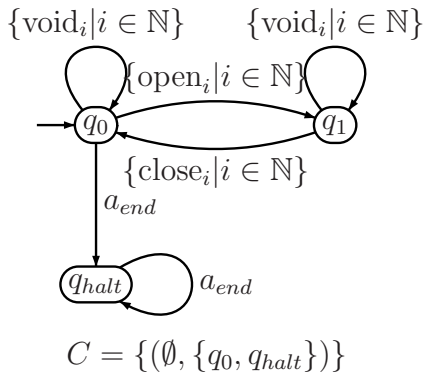


Figure 3.5: Example 2, Rabin automaton

This security property is modeled by the automaton in Figure 3.5, over the alphabet $\Sigma = \{\text{open}, \text{close}, \text{void}\}$. This automaton captures a plausible security requirement for a program that accesses a database, namely that:

- The program can open no more that one connection to the database at any given time.
- The program only closes a connection to the database if it has already been opened.
- Any connection that is opened is eventually closed. This last requirement adds a liveness component to the desired security property.

Note that the first two actions from Σ model the operation of opening and closing the database, while the last is used as a stand in for other program actions that have no bearing on the satisfaction of the security property.

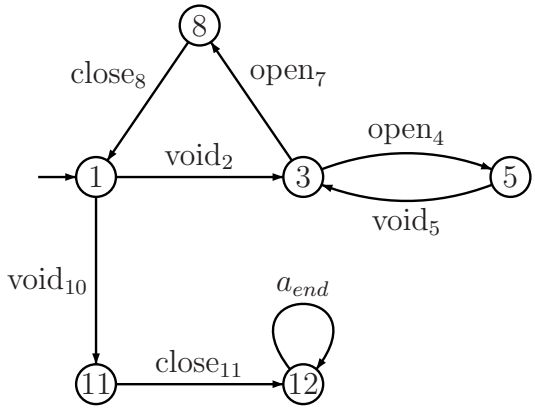


Figure 3.6: Example 2, LTS

Figure 3.6 shows a LTS which approximates the behavior of the program whose execution we wish to monitor. This program could be a remote agent who accesses a database, performs some computations locally and returns a result. The subscripts added to the atomic actions serve only to avoid the presence of non-determinism in the model (each action is associated with a distinct program instruction), and has no bearing on the satisfaction of the security predicate.

The transformed product automaton $\mathcal{R}^T$ of Example 2 is depicted in Figure 3.7.

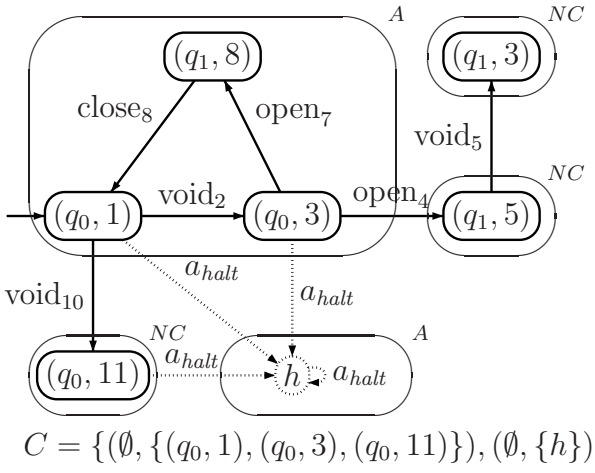


Figure 3.7: Example 2, Transformed product automaton

In Figure 3.7, the strongly connected components are shown and annotated with

either A meaning all the component's cycles are admissible, NC meaning the component has no cycles and N meaning all the cycles of the component are inadmissible.

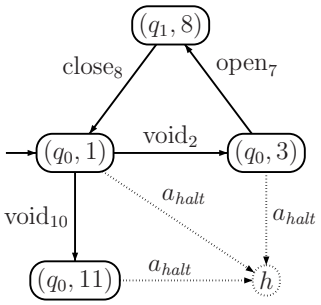


Figure 3.8: Example 2, Truncation automaton

We omit the intermediate steps performed by the algorithm, since they have already been discussed throughout the last section using the preceding example. More relevant is the result returned by our approach in this case, namely a LTS that could be transformed into a provably secure program. This LTS is presented in Figure 3.8.

3.4 Mechanism's Enforcement power

In this section, we show that nonuniform enforcement mechanisms, which occur when the set of possible executions $\mathcal{S}$ is a subset of A^ω, are more powerful than uniform enforcers, i.e. those for which $\mathcal{S} = A^\omega$, in the sense that they are able to enforce a larger class of security properties. This demonstration will reveal that monitors that are tailored to specific programs may be able to enforce a wide set of properties and argues for the use of static analysis in conjunction with monitoring.

Let us begin with a more formal definition of the concepts we discussed in the previous sections. Intuitively, we can think of security enforcement mechanisms as sequence transformers, automata that take a program's actions sequence as input, and outputs a new sequence of actions that respects the security property. This intuition is formalized as follows:

Definition 3.4.1 (Transformation, from [11])**.** *A security automaton $M = (Q, q_0, \delta)$ transforms an execution trace $\sigma \in \Sigma^\infty$ into an execution $\tau \in \Sigma^\infty$, noted $(q_0, \sigma)\Downarrow_M \tau$, if*

and only if

$$\forall q' \in Q, \sigma' \in \Sigma^{\infty}, \tau' \in \Sigma^{*} \mid: ((q_0, \sigma) \stackrel{\tau'}{\Longrightarrow}_M (q', \sigma')) \Longrightarrow \tau' \preccurlyeq \tau \tag{3.4.1}$$

$$\forall \tau' \preccurlyeq \tau \mid: \exists q' \in Q, \sigma' \in \Sigma^{\infty} \mid: (q_0, \sigma) \stackrel{\tau'}{\Longrightarrow}_M (q', \sigma') \tag{3.4.2}$$

We can now state formally what it means for an enforcement mechanism to effectively enforce a security property

Definition 3.4.2 (effective$^{\mathcal{S}}_{\cong}$ Enforcement, from [11])**.** *Let $\mathcal{S} \subseteq \Sigma^{\infty}$ be a set of execution traces and let $\cong$ be an equivalence relation between executions. A security automaton $M = (Q, q_0, \delta)$* enforces effectively$_{\cong}$ *a security property $\hat{\mathcal{P}}$ for $\mathcal{S}$ if and only if for all input trace $\sigma \in \mathcal{S}$ there exists an output trace $\tau \in \Sigma^{\infty}$ such that*

$$(q_0, \sigma) \Downarrow_M \tau \tag{3.4.3}$$

$$\hat{\mathcal{P}}(\tau) \tag{3.4.4}$$

$$\hat{\mathcal{P}}(\sigma) \Longrightarrow \sigma \cong \tau \tag{3.4.5}$$

Informally, a security automaton *enforces effectively*$_{\cong}$ a property for $\mathcal{S}$ *iff* for each execution trace $\sigma \in \mathcal{S}$, it outputs a trace τ such that τ is valid, with respect to the property, and if the input trace σ is itself valid then $\sigma \cong \tau$.

Definition 3.4.3 ($\mathcal{M}^{\mathcal{S}}_{\cong}$-enforceable)**.** *Let $\mathcal{S} \subseteq \Sigma^{\infty}$ be a set of execution traces and $\mathcal{M}$ be a type of security automata. The class $\mathcal{M}^{\mathcal{S}}_{\cong}$-enforceable is the set of security properties for which there exists a security automaton in $\mathcal{M}$ that effectively$_{\cong}$ enforces this property for the traces in $\mathcal{S}$.*

Our approach is built around the idea, first suggested by Ligatti et al. in [11, 53], that the set of properties enforceable by a monitor could sometimes be extended if the monitor has some knowledge of the program's possible behavior and thus can rule out some executions as impossible.

We can now state this idea more formally.

Theorem 3.4.4. *Let $\mathcal{M}$ be a class of security automata and let $\mathcal{S}^{\natural}, \mathcal{S}^{\sharp} \subseteq \Sigma^{\infty}$ be two sets of execution traces $\mathcal{S}^{\natural} \subseteq \mathcal{S}^{\sharp}$ then we have*

$$\mathcal{M}^{\mathcal{S}^{\sharp}}_{\cong}\text{-enforceable} \subseteq \mathcal{M}^{\mathcal{S}^{\natural}}_{\cong}\text{-enforceable} \tag{3.4.6}$$

The proof is quite straightforward, and based upon the intuition that a security mechanism possessing certain knowledge about its target may discard it, and then behave as an enforcement mechanisms lacking this knowledge.

Proof. Let $\mathcal{S}^\natural, \mathcal{S}^\sharp \subseteq \Sigma^\infty$ be two sets of execution traces such that $\mathcal{S}^\natural \subseteq \mathcal{S}^\sharp$ and $\hat{\mathcal{P}}$ be a $\mathcal{M}^{\mathcal{S}^\sharp}_{\cong}$-enforceable security property.

$$
\begin{array}{ll}
& \hat{\mathcal{P}} \in \mathcal{M}^{\mathcal{S}^\sharp}_{\cong}\text{-enforceable} \\
\iff & \langle \text{ Definition 3.4.2 and 3.4.3 } \rangle \\
& \left(\exists M = (Q, q_0, \delta) \in \mathcal{M} \;\middle|\; : \left(\forall \sigma \in \mathcal{S}^\sharp \;\middle|\; : \left(\exists \tau \in \Sigma^\infty \;\middle|\; : \begin{array}{ll} & (q_0, \sigma) \Downarrow_M \tau \\ \wedge & \hat{\mathcal{P}}(\tau) \\ \wedge & \left(\hat{\mathcal{P}}(\sigma) \Longrightarrow \sigma \cong \tau \right) \end{array} \right) \right) \right) \\
\Longrightarrow & \langle \text{ Domain weakening } (\mathcal{S}^\natural \subseteq \mathcal{S}^\sharp) \rangle \\
& \left(\exists M = (Q, q_0, \delta) \in \mathcal{M} \;\middle|\; : \left(\forall \sigma \in \mathcal{S}^\natural \;\middle|\; : \left(\exists \tau \in \Sigma^\infty \;\middle|\; : \begin{array}{ll} & (q_0, \sigma) \Downarrow_M \tau \\ \wedge & \hat{\mathcal{P}}(\tau) \\ \wedge & \left(\hat{\mathcal{P}}(\sigma) \Longrightarrow \sigma \cong \tau \right) \end{array} \right) \right) \right) \\
\iff & \langle \text{ Definition 3.4.2 and 3.4.3 } \rangle \\
& \hat{\mathcal{P}} \in \mathcal{M}^{\mathcal{S}^\natural}_{\cong}\text{-enforceable}
\end{array}
$$

□

Corollary 3.4.5. *Let $\mathcal{M}$ be a class of security automaton. For all execution trace set $\mathcal{S} \subseteq \Sigma^\infty$ we have*

$$\mathcal{M}^{\Sigma^\infty}_{\cong}\text{-enforceable} \subseteq \mathcal{M}^{\mathcal{S}}_{\cong}\text{-enforceable} \tag{3.4.7}$$

Corollary 3.4.5 indicates that any security property that is effectively$_{\cong}$ enforceable by a security automaton in a uniform context ($\mathcal{S} = \Sigma^\infty$) is also enforceable in the nonuniform context ($\mathcal{S} \neq \Sigma^\infty$). It follows that our approach is at least as powerful as those previously suggested in the literature.

It would be interesting to prove that for all security automaton classes, $\mathcal{M}$, for all $\mathcal{S} \subset \Sigma^\infty$ and for all equivalence relations $\cong$, we have $\mathcal{M}^{\Sigma^\infty}_{\cong}$-enforceable $\subset \mathcal{M}^{\mathcal{S}}_{\cong}$-enforceable.

This is unfortunately not the case, as there exists at least one class of security automaton (ex. $\mathcal{M} = \emptyset$), and one equivalence relation (ex. $\tau \cong \sigma \ \forall \tau, \sigma \in \Sigma^\infty$) such that $\mathcal{M}^{\Sigma^\infty}_{\cong}$-enforceable $= \mathcal{M}^{\mathcal{S}}_{\cong}$-enforceable for all sets of traces $\mathcal{S} \subseteq \Sigma^\infty$. However in our approach, we focus both on a specific class of security automata and on a specific equivalence relation. In our particular case, the set of policies enforceable in a nonuniform context is strictly greater than the one that is enforceable in the uniform context.

The monitors used in this study are truncation automata, first described in [69]. These are monitors which, when presented with a potentially invalid sequence, have no option but to abort the execution.

The following theorem gives a characterization of the properties that can be effectively$_{\cong}$ enforced in a nonuniform context.

Theorem 3.4.6. *Let $\mathcal{S} \subseteq \Sigma^\infty$ be a set of execution sequences. A property $\hat{\mathcal{P}}$ is* $\mathsf{T}^{\mathcal{S}}_{\cong}$-enforceable *iff there exists a decidable predicate D over Σ^* such that*

$$(\forall \sigma \in \mathcal{S} \mid \neg\hat{\mathcal{P}}(\sigma) : (\exists \sigma' \preccurlyeq \sigma \mid : D(\sigma'))) \tag{3.4.8}$$

$$(\forall \tau; a \in \Sigma^* \mid D(\tau; a) : \hat{\mathcal{P}}(\tau) \wedge (\forall \sigma \in \mathcal{S} \mid \sigma \not\succcurlyeq \tau; a \wedge \hat{\mathcal{P}}(\sigma) : \sigma \cong \tau)) \tag{3.4.9}$$

$$\neg D(\varepsilon) \tag{3.4.10}$$

Proof. (if direction)
Let $\mathcal{S} \subseteq \Sigma^\infty$ be a set of execution sequences, let $\hat{\mathcal{P}}$ be a property, $\cong$ an equivalence relation over the execution sequences and D be a decidable predicate over Σ^* which satisfies the conditions (3.4.8), (3.4.9) and (3.4.10). We can construct a truncation automaton T which effectively$_{\cong}$enforces $\hat{\mathcal{P}}$ over $\mathcal{S}$.

Let $T = (Q, q_0, \delta)$ be a truncation automaton such that

- $Q = \Sigma^*$;
- $q_0 = \varepsilon$;
- For all $\sigma \in \Sigma^*$, $a \in \Sigma$

$$\delta(\sigma, a) = \begin{cases} \sigma; a & \text{if } \neg D(\sigma; a) \\ \mathit{halt} & \text{else} \end{cases}$$

Since Σ^* is a countably infinite set, so is Q. Furthermore, δ is fully defined and computable since D is decidable.

The automaton T satisfies the following invariant $I(q)$: for any reachable state $q = \sigma$, the execution sequence σ has been emitted so far, $(q_0, \sigma)\Downarrow_T\sigma$ and $(\forall\sigma' \preccurlyeq \sigma \,|:\, \neg D(\sigma'))$. Informally, if the automaton T is in state $q = \sigma$, it did not halt before reaching this state and has emitted exactly the execution sequence σ. T obviously satisfies the $I(q_0)$ since $q_0 = \varepsilon$, $(q_0, \varepsilon)\Downarrow_T\varepsilon$ and $\neg D(\varepsilon)$. An induction on the length of the execution trace shows that $I(q)$ is satisfied for any reachable state $q = \sigma$.

Let $\sigma \in \mathcal{S}$ be an input sequence. We show that T effectively$_{\cong}$ enforces $\hat{\mathcal{P}}$ over $\mathcal{S}$ by showing that T satisfies conditions (3.4.3), (3.4.4) and (3.4.5) for σ.

- if $\neg\hat{\mathcal{P}}(\sigma)$
  By condition (3.4.8) we have that $(\exists\sigma' \preccurlyeq \sigma \,|:\, D(\sigma'))$. Thus, by invariant I, T must halt if its input sequence is σ, and must do so before reaching the state σ'. Let τ be the last state which T reaches when its input is σ. By I and definition 3.4.1 we have that $(q_0, \sigma)\Downarrow_T\tau$. Thus condition (3.4.3) is satisfied. Let $a \in \Sigma$ be an action such that $\tau; a \preccurlyeq \sigma$. Since τ is the last state T reaches when σ is input and by the definition of T we have that $\delta(\tau, a) = \mathit{halt}$ and thus that $D(\tau; a)$. Furthermore, by condition (3.4.9) we have that $\hat{\mathcal{P}}(\tau)$. It follows that condition (3.4.4) is satisfied. Since $\neg\hat{\mathcal{P}}(\sigma)$, condition (3.4.5) is satisfied.
- if $\hat{\mathcal{P}}(\sigma)$
  - if T does not halt on input σ
    then T emits all and every prefix of σ. By definition 3.4.1 we have $(q_0, \sigma)\Downarrow_T\sigma$. Thus, condition (3.4.3) is satisfied. Since $\hat{\mathcal{P}}(\sigma) \cong$ is reflexive, conditions (3.4.4) and (3.4.5) are also satisfied.
  - if T halts on input σ
    Let τbe the last state T reaches when its input is σ. By I and definition 3.4.1 we have that $(q_0, \sigma)\Downarrow_T\tau$. Thus, la condition (3.4.3) is satisfied. Let $a \in \Sigma$ be an action such that $\tau; a \preccurlyeq \sigma$. Since τ is the last state T reaches when σ is input and by the definition of T we have that $\delta(\tau, a) = \mathit{halt}$ and thus that $D(\tau; a)$. By condition (3.4.9) we have $\hat{\mathcal{P}}(\tau)$ and since $\hat{\mathcal{P}}(\sigma)$ we have that $\sigma \cong \tau$. Thus conditions (3.4.4) and (3.4.5) are satisfied.

(elseif direction)
Let $\mathcal{S} \subseteq \Sigma^\infty$ be a set of execution sequences and let $\hat{\mathcal{P}}$ be a property, $\cong$ an equivalence relation over the execution sequences and $T = (Q, q_0, \delta)$ be a truncation automaton which effectively$_{\cong}$ enforces $\hat{\mathcal{P}}$ over $\mathcal{S}$. We construct a decidable D over Σ^* such that the conditions (3.4.8), (3.4.9) and (3.4.10) are satisfied.

This predicate is built in the following manner :

- $D(\varepsilon)$ is false.
- For all $\sigma \in \Sigma^*$, $a \in \Sigma$ we have that $D(\sigma; a)$ is true iff automaton T emits exactly σ when the input sequence is $\sigma; a$.

Since δ is fully defined and computable, and D is only defined over Σ^* we can conclude that D is also fully defined and decidable.

The definition of D, implies that condition (3.4.10) is satisfied.

Furthermore, since T effectively$_{\cong}$ enforces $\hat{\mathcal{P}}$ over $\mathcal{S}$, if it outputs σ the input sequence is $\sigma; a$, then, by condition (3.4.4) we have that $\hat{\mathcal{P}}(\sigma)$. Also, by condition (3.4.5) the fact that T halts before emitting any sequence $\tau \in \mathcal{S}$ such that $\tau \succcurlyeq \sigma; a$, we have that condition (3.4.9) is satisfied.

Finally, let $\sigma \in \mathcal{S}$ be an input trace such that $\neg\hat{\mathcal{P}}(\sigma)$ and assume that $(\forall \sigma' \preccurlyeq \sigma \mid : \neg D(\sigma'))$ as to obtain a contradiction. By the definition of D, automaton T cannot halt on any prefix of σ, and must thus necessarily output all its prefixes. Yet by definition 3.4.1 we have that $(q_0, \sigma)\Downarrow_T \sigma$.

This is a contradiction since T cannot effectively$_{\cong}$ enforce $\hat{\mathcal{P}}$ over $\mathcal{S}$ if there exists an input trace $\sigma \in \mathcal{S}$ such that $(q_0, \sigma)\Downarrow_T \sigma$ and $\neg\hat{\mathcal{P}}(\sigma)$. It follows that condition (3.4.8) is satisfied. □

Finally, since we also restrict ourselves in this study to the use of syntactic equivalence ($=$) as the equivalence relation between valid traces, we give a characterization of the security properties effectively$_{=}$ enforceable by a truncation automaton, in a nonuniform context.

Corollary 3.4.7. *Let $\mathcal{S} \subseteq \Sigma^\infty$ be a set of execution sequences. A property $\hat{\mathcal{P}}$ is* $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable *iff there exists a decidable predicate D over Σ^* such that*

$$(\forall \sigma \in \mathcal{S} \mid \neg\hat{\mathcal{P}}(\sigma) : (\exists \sigma' \preccurlyeq \sigma \mid : D(\sigma'))) \tag{3.4.11}$$

$$(\forall \tau; a \in \Sigma^* \mid D(\tau; a) : \hat{\mathcal{P}}(\tau) \wedge (\forall \sigma \in \mathcal{S} \mid \sigma \succcurlyeq \tau; a : \neg\hat{\mathcal{P}}(\sigma))) \tag{3.4.12}$$

$$\neg D(\varepsilon) \tag{3.4.13}$$

Proof. Let $\mathcal{S} \subseteq \Sigma^\infty$ be a set of execution sequences and let $\hat{\mathcal{P}}$ be a security property such that $\hat{\mathcal{P}} \in \mathsf{T}^{\mathcal{S}}_{=}$-enforceable.

$$
\begin{array}{ll}
& \hat{\mathcal{P}} \in \mathsf{T}^{\mathcal{S}}_{=}\text{-enforceable} \\
\iff & \langle \text{ Theorem 3.4.6 } \rangle \\
& \left(\exists D \;\middle|\; \begin{array}{ll} : & (\forall \sigma \in \mathcal{S} \mid \neg\hat{\mathcal{P}}(\sigma) : (\exists \sigma' \preccurlyeq \sigma \mid : D(\sigma'))) \\ : \wedge & (\forall \tau; a \in \Sigma^* \mid D(\tau; a) : \hat{\mathcal{P}}(\tau) \wedge (\forall \sigma \in \mathcal{S} \mid \sigma \succcurlyeq \tau; a \wedge \hat{\mathcal{P}}(\sigma) : \sigma = \tau)) \\ \wedge & \neg D(\varepsilon) \end{array} \right) \\
\iff & \langle \text{ Transfer } \rangle \\
& \left(\exists D \;\middle|\; \begin{array}{ll} : & (\forall \sigma \in \mathcal{S} \mid \neg\hat{\mathcal{P}}(\sigma) : (\exists \sigma' \preccurlyeq \sigma \mid : D(\sigma'))) \\ : \wedge & (\forall \tau; a \in \Sigma^* \mid D(\tau; a) : \hat{\mathcal{P}}(\tau) \wedge (\forall \sigma \in \mathcal{S} \mid \sigma \succcurlyeq \tau; a : \hat{\mathcal{P}}(\sigma) \implies \sigma = \tau)) \\ \wedge & \neg D(\varepsilon) \end{array} \right) \\
\iff & \langle \; \sigma \succcurlyeq \tau; a \implies \sigma \succ \tau \implies \sigma \neq \tau \text{ and } (p \implies \mathsf{false}) \iff (\neg p) \; \rangle \\
& \left(\exists D \;\middle|\; \begin{array}{ll} : & (\forall \sigma \in \mathcal{S} \mid \neg\hat{\mathcal{P}}(\sigma) : (\exists \sigma' \preccurlyeq \sigma \mid : D(\sigma'))) \\ : \wedge & (\forall \tau; a \in \Sigma^* \mid D(\tau; a) : \hat{\mathcal{P}}(\tau) \wedge (\forall \sigma \in \mathcal{S} \mid \sigma \succcurlyeq \tau; a : \neg\hat{\mathcal{P}}(\sigma))) \\ \wedge & \neg D(\varepsilon) \end{array} \right)
\end{array}
$$

□

The intuition behind the above proof is simply to apply conditions given in theorem 3.4.6 in a context where $\mathcal{S}$ is equal to Σ^∞.

We can now state the central theorem of this chapter, that the enforcement power of the truncation automaton is strictly greater in the nonuniform context than in the uniform context, when we consider =-enforcement.

Theorem 3.4.8. *For all set of traces $\mathcal{S} \subset \Sigma^\infty$ we have*

$$\mathsf{T}^{\Sigma^\infty}_{=}\text{-enforceable} \subset \mathsf{T}^{\mathcal{S}}_{=}\text{-enforceable} \tag{3.4.14}$$

Proof. The proof is based on the following observations. First, it has been shown in [11, 69] that a property is $\mathsf{T}^{\Sigma^\infty}_{=}$-enforceable *iff* it is a safety property. Second, let $\hat{\mathcal{P}}$ be a security property, $\hat{\mathcal{P}}$ is trivially enforceable on $\mathcal{S}$ *iff* for every sequence $\sigma \in \mathcal{S}$, $\hat{\mathcal{P}}(\sigma)$.

Let $T = (Q, q_0, \delta)$ be a truncation automaton such that for every sequence $\sigma \in \Sigma^\infty$ we have $(q_0, \sigma)\Downarrow_T \sigma$. It is easy to see that this automaton trivially enforces over the set $\mathcal{S}$ any trivially enforceable property over this same set. The remainder of this proof consists in showing that there exist some properties which are trivially enforceable over $\mathcal{S}$ but are not reasonable, safety and decidable.

Let $\upsilon \in \Sigma^\infty$ be an execution sequence such that $\upsilon \notin \mathcal{S}$ and let $\hat{\mathcal{P}}$ be the security property stating that for any sequence $\sigma \in \Sigma^\infty$ we have

$$\hat{\mathcal{P}}(\sigma) \Longleftrightarrow (\sigma \neq \upsilon)$$

This property is $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable, and the following automaton, $T = (Q, q_0, \delta)$ with $\delta(q_0, a) = q_0$ for all actions $a \in \Sigma$ effectively enforces it over Σ .

We show that $\hat{\mathcal{P}} \notin \mathsf{T}^{\Sigma^\infty}_{=}$-enforceable by contradiction. Were it the case that $\hat{\mathcal{P}} \in \mathsf{T}^{\Sigma^\infty}_{=}$-enforceable there would necessarily be a decidable predicate as defined in Corollary 3.4.7. However, such a predicate cannot for the property $\hat{\mathcal{P}}$.

Assume that $\hat{\mathcal{P}} \in \mathsf{T}^{\Sigma^\infty}_{=}$-enforceable, by definition, there exists a predicate D over Σ^* such that conditions (3.4.11), (3.4.12) et (3.4.13) are satisfied.

$$
\begin{array}{ll}
& \hat{\mathcal{P}} \in \mathsf{T}^{\Sigma^\infty}_{=}\text{-enforceable} \\
\Longleftrightarrow & \langle \text{ Corrolairy (3.4.7) and definition of } \hat{\mathcal{P}} \text{ } \rangle \\
& \qquad (\forall \sigma \in \Sigma^\infty \mid \sigma = \upsilon : (\exists \sigma' \preccurlyeq \sigma \mid : D(\sigma'))) \\
(\exists D \mid : & \wedge \quad (\forall \tau; a \in \Sigma^* \mid D(\tau; a) : \tau \neq \upsilon \wedge (\forall \sigma \in \Sigma^\infty \mid \sigma \succcurlyeq \tau; a : \sigma = \upsilon)) \ ) \\
& \wedge \qquad\qquad \neg D(\varepsilon) \\
\Longrightarrow & \langle \text{ Axiom of choice on the first } \forall.\ \tau; a \succcurlyeq \tau; a,\ \tau; a; a \succcurlyeq \tau; a \text{ and domain} \\
& \text{weakening on the third } \forall \text{ } \rangle \\
& \qquad (\exists \sigma' \preccurlyeq \upsilon \mid : D(\sigma')) \\
(\exists D \mid : & \wedge \quad (\forall \tau; a \in \Sigma^* \mid D(\tau; a) : \tau \neq \upsilon \wedge \tau; a = \upsilon \wedge \tau; a; a = \upsilon) \ ) \\
& \wedge \qquad\qquad \neg D(\varepsilon) \\
\Longrightarrow & \langle \text{ Definition of } \preccurlyeq \text{ and domain Weakening on the second } \exists. \text{ Contradiction} \\
& \text{and transfer for } \forall \text{ } \rangle \\
& \qquad (\exists \sigma' \in \Sigma^* \mid : D(\sigma')) \\
(\exists D \mid : & \wedge \quad (\forall \tau; a \in \Sigma^* \mid : D(\tau; a) \Longrightarrow \mathsf{false}) \ ) \\
& \wedge \qquad\qquad \neg D(\varepsilon) \\
\Longleftrightarrow & \langle \text{ Renaming. } p \Longrightarrow \mathsf{false} \Longleftrightarrow \neg p \text{ } \rangle \\
(\exists D \mid : & (\exists \sigma \in \Sigma^* \mid : D(\sigma)) \wedge (\forall \tau; a \in \Sigma^* \mid : \neg D(\tau; a)) \wedge \neg D(\varepsilon))
\end{array}
$$

$\iff$ $\langle$ Domain Splitting $\rangle$

$(\exists D \mid : (\exists \sigma \in \Sigma^* \mid : D(\sigma)) \wedge (\forall \sigma \in \Sigma^* \mid : \neg D(\sigma)))$

$\iff$ $\langle$ Contradiction and $(\exists x \mid R : \mathsf{false}) \iff \mathsf{false}$ $\rangle$

false

□ □

Having determined that the set of monitorable properties is indeed increased by relying on static analysis to narrow the set of possible executions, one would naturally wonder if this improvement occurs monotonously, i.e. if every time a sequence υ is removed from a set $\mathcal{S}$, a new property is added to the set of $\mathsf{T}_{=}^{\mathcal{S}}$-enforceable properties. This would be desirable, as it would imply that any effort made to perform or refine a static analysis of the target program would payoff in the form of an increase in the set of enforceable properties.

Unfortunately, this does not bear out, and there are cases where reducing the size of the set of possible sequences does not result in any advantage. As a counter-example, consider a simple system that can only perform two sequences, each containing only one action, either a or b, or output nothing, thus $\mathcal{S} = \{\epsilon, a, b\}$. Further, let $\mathcal{S}' = \{\epsilon, a\}$. We can limit our analysis to properties that differ only with respect to the sequences present in $\mathcal{S}$. Eight such sequence sets exist. Because the set of possible properties is finite, it is tractable to examine each of them and determine that w.r.t. the criteria given in theorem 3.4.6, the sets of properties enforceable by each is indeed the same. The details of this analysis are given in table 3.1. Each line represents a different property, and each column a set of possible execution sequences. We write ✓ to indicate that the property is truncation enforceable for a given set $\mathcal{S}$ or $\mathcal{S}'$ and $\mathcal{X}$ if the property is not enforceable. As can be shown from this table, in each case, either the property is in both $\mathsf{T}_{=}^{\mathcal{S}'}$-enforceable and $\mathsf{T}_{=}^{\mathcal{S}}$-enforceable or it is in neither set. For example, a property containing all three sequences, ϵ, a and b is trivially enforceable in both cases, since every possible sequence is valid. Properties for which $\hat{\mathcal{P}} \cap \mathcal{S} = \{\epsilon, a\}$, $\hat{\mathcal{P}} \cap \mathcal{S} = \{\epsilon, b\}$ or $\hat{\mathcal{P}} \cap \mathcal{S} = \{\epsilon\}$ are safety properties (for a monitor for which only the sequences in $\mathcal{S}$ are possible) and are also enforceable in both cases. However, if the empty sequence ϵ is disallowed, a truncation-based mechanism cannot enforce the property since some sequences have no valid prefixes.

In order to increase the set of enforceable properties, at least one of the following three conditions must be met.

Theorem 3.4.9. (Constraining $\mathcal{S}$) *Let* $\mathcal{S} \subset \mathcal{S}' \subseteq \Sigma^\infty$, $\mathsf{T}_{=}^{\mathcal{S}'}$-enforceable $\subset$ $\mathsf{T}_{=}^{\mathcal{S}}$-enforceable

	$\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable	$\mathsf{T}^{\mathcal{S}}_{=}$-enforceable
$\hat{\mathcal{P}} \cap \mathcal{S} = \{\epsilon, a, b\}$	✓	✓
$\hat{\mathcal{P}} \cap \mathcal{S} = \{\epsilon, a\}$	✓	✓
$\hat{\mathcal{P}} \cap \mathcal{S} = \{\epsilon, b\}$	✓	✓
$\hat{\mathcal{P}} \cap \mathcal{S} = \{a, b\}$	✗	✗
$\hat{\mathcal{P}} \cap \mathcal{S} = \{\epsilon\}$	✓	✓
$\hat{\mathcal{P}} \cap \mathcal{S} = \{a\}$	✗	✗
$\hat{\mathcal{P}} \cap \mathcal{S} = \{b\}$	✗	✗
$\hat{\mathcal{P}} \cap \mathcal{S} = \emptyset$	✗	✗

Table 3.1: Enforceable properties when $\mathcal{S} = \{\epsilon, a, b\}$ and $\mathcal{S}' = \{\epsilon, a\}$

iff at least one of the three following conditions are met:

1. $\exists \upsilon \in \mathcal{S}' \backslash \mathcal{S} : \exists \tau \preceq \upsilon : \tau \neq \epsilon \wedge \exists \upsilon' \succeq \tau : \upsilon' \in \mathcal{S}' \wedge \upsilon \neq \upsilon'$
2. $\epsilon \notin \mathcal{S}$
3. $\exists \upsilon \in \mathcal{S}' \backslash \mathcal{S}$: *there does not exist a decidable function* $f : \mathcal{S}' \times \mathbb{B}$ *s.t.*$\forall \tau \in \mathcal{S}$: $f(\tau_0) =$ ***true*** *if* $\tau = \upsilon$ *and* ***false*** *otherwise.*

Informally, theorem 3.4.9 states that the set of enforceable properties is increased in three cases. First, if there exits a sequence υ, removed from the set of possible sequences $\mathcal{S}'$ which has a prefix that is both different from ϵ and has an extension in $\mathcal{S}'$ other than υ. Second, if the set of possible sequences does not contain the empty sequence ϵ. Finally, if a sequence υ has been removed for which there does not exist a decidable predicate with which can the monitor, at the onset of the execution, determine that the ongoing execution is υ. The intuition of the proof is to exhibit in each case an example of a property that is not in $\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable but is in $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable. In the first two cases, this occurs because there is an invalid sequence with no valid prefix on which the execution can be aborted. In the latter case, it is because the monitor cannot discover in time that the ongoing execution is indeed invalid.

Proof. (if direction) We show that in each of the cases given above, at least one property is in $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable but not in $\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable.

1. $\exists \upsilon \in \mathcal{S}' \backslash \mathcal{S} : \exists \tau \preceq \upsilon : \tau \neq \epsilon \wedge \exists \upsilon' \succeq \tau : \upsilon' \in \mathcal{S}' \wedge \upsilon \neq \upsilon'$

There are three cases to consider :

case $\upsilon \prec \upsilon'$**:** Let $\hat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma = \epsilon \vee \sigma = \upsilon')$. This property is not in $\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable since the valid sequence υ' has an invalid prefix in $\mathcal{S}'$. The property is enforceable over $\mathcal{S}$ by an automaton which aborts the every execution on ϵ, the only valid sequences in $\mathcal{S}$.

case $\upsilon' \prec \upsilon$ **:** Likewise, in this case, the property $\hat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma = \epsilon \vee \sigma = \upsilon)$ is not in $\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable, but is $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable.

case $\upsilon' \not\preceq \upsilon \vee \upsilon \not\preceq \upsilon'$ **:** Let υ'' be the smallest sequence in $\mathcal{S}'\backslash\mathcal{S}$ s.t. $\upsilon'' \succ \tau \wedge \upsilon'' \preceq \upsilon'$. Let $\hat{\mathcal{P}}$ be the property $\hat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma \preceq \tau \wedge \sigma \succeq \upsilon'')$. This property is not in $\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable since the monitor cannot abort on any prefix of υ'' without losing either transparency or correctness. The property is trivially enforceable over $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable.

2. $\epsilon \notin \mathcal{S}$
   There are two cases to consider :

   case $\exists \upsilon' \in \mathcal{S} : \upsilon' \prec \upsilon$**:** As was the case above, the property $\hat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma = \epsilon \vee \upsilon)$ is not in $\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable, but is $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable.

   case $\neg\exists \upsilon' \in \mathcal{S} : \upsilon' \prec \upsilon$**:** Let $\hat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma \not\preceq \upsilon)$. This property is not in $\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable since sequence υ has no valid prefix. The property is trivially $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable.

3. $\forall \upsilon \in \mathcal{S}'\backslash\mathcal{S}$: there exists a decidable function$f : \mathcal{S}' \times \mathbb{B}$ s.t.$\forall \tau \in \mathcal{S} : f(\tau_0) =$ **true** if $\tau = \upsilon$ and **false** otherwise.
   Let $\hat{\mathcal{P}}$ be defined such that $\hat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma \not\preceq \upsilon \vee \sigma = \epsilon)$. Let the input sequence be υ. Assume further that none of the two cases mentioned above apply. If the function described above does not exist, then the monitor cannot recognize on the first action of the sequence that the input is invalid. Since ϵ is the only valid sequence prefix of the input, the monitor has no detector to indicate when the abort the execution, and thus cannot enforce the property. This property is trivially enforceable over $\mathcal{S}$.

(elseif direction)

We show that the set of $\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable properties = $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable properties if the following three conditions are met

1. $\forall \upsilon \in \mathcal{S}'\backslash\mathcal{S} : \forall \tau \preceq \upsilon : \tau \neq \epsilon \Rightarrow \exists \upsilon' \succeq \tau : \upsilon' \in \mathcal{S}' \wedge \upsilon \neq \upsilon'$
2. $\epsilon \in \mathcal{S}$
3. $\forall \upsilon \in \mathcal{S}'\backslash\mathcal{S}$: there exists a decidable function$f : \mathcal{S}' \times \mathbb{B}$ s.t.$\forall \tau \in \mathcal{S} : f(\tau_0) =$ **true** if $\tau = \upsilon$ and **false** otherwise.

Let $\hat{\mathcal{P}}$ be a property in $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable. By definition, there exists a decidable predicate D over the sequences of $\mathcal{S}$ meeting the requirements of theorem 3.4.6. For all $\sigma \in \mathcal{S}', a \in \Sigma, \upsilon \in \mathcal{S}'\backslash\mathcal{S}$, we define the predicate $D^\star$ as follows

$$D^\star(\tau; a) = \begin{cases} D(\tau; a), & \text{if it is defined ;} \\ \mathsf{true}, & \text{if} \neg\hat{\mathcal{P}}(f(\tau; a)) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

The intuition behind the proof is to show that for any property that is $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable, we can exhibit a decidable predicate over Σ^* meeting the criteria given in Theorem 3.4.6, thus rendering the property $\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable. First, observe that any property $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable is necessarily reasonable since $\epsilon \in \mathcal{S}$. Condition 3 above ensures that a decidable predicate exists which can halt the execution on some prefix of any sequence $\upsilon \in \mathcal{S}'\backslash\mathcal{S}$, and since, $\hat{\mathcal{P}}(\epsilon)$, this prefix is valid. Furthermore, from condition one, we have that ϵ cannot be prefix of any sequence other than υ. Finally, observe that it must be possible to detect from the onset that the current execution is a prefix of υ, since otherwise, a property for which υ has no valid prefix other than ϵ would be $\mathsf{T}^{\mathcal{S}}_{=}$-enforceable but not $\mathsf{T}^{\mathcal{S}'}_{=}$-enforceable.

$$\begin{array}{ll} & \hat{\mathcal{P}} \in \mathsf{T}^{\mathcal{S}}_{=}\text{-enforceable} \\ \iff & \langle \text{ Theorem 3.4.6 } \rangle \\ & \left(\exists D \;\middle|\; \begin{array}{cc} & (\forall \sigma \in \mathcal{S} \mid \neg\hat{\mathcal{P}}(\sigma) : (\exists \sigma' \preccurlyeq \sigma \mid : D(\sigma'))) \\ : \wedge & (\forall \tau; a \in \Sigma^* \mid D(\tau; a) : \hat{\mathcal{P}}(\tau) \wedge (\forall \sigma \in \mathcal{S} \mid \sigma \succcurlyeq \tau; a \wedge \hat{\mathcal{P}}(\sigma) : \sigma = \tau)) \\ \wedge & \neg D(\varepsilon) \end{array} \right) \\ \iff & \langle \text{ from cond. 3, } \forall \sigma \in \mathcal{S}'\backslash\mathcal{S} : \exists \sigma' \preceq \sigma : D^\star(\sigma') \rangle \\ & \left(\exists D \;\middle|\; \begin{array}{cc} & (\forall \sigma \in \mathcal{S}' \mid \neg\hat{\mathcal{P}}(\sigma) : (\exists \sigma' \preccurlyeq \sigma \mid : D(\sigma'))) \\ : \wedge & (\forall \tau; a \in \Sigma^* \mid D(\tau; a) : \hat{\mathcal{P}}(\tau) \wedge (\forall \sigma \in \mathcal{S} \mid \sigma \succcurlyeq \tau; a \wedge \hat{\mathcal{P}}(\sigma) : \sigma = \tau)) \\ \wedge & \neg D(\varepsilon) \end{array} \right) \\ \iff & \langle \text{ since } \epsilon \in \mathcal{S}, \forall \hat{\mathcal{P}} \in \mathsf{T}^{\mathcal{S}}_{=}\text{-enforceable we have that } \hat{\mathcal{P}}(\epsilon) \text{ and from cond.1} \\ & \text{and the definition of } D^\star \text{ we have that } D^\star(\sigma) \Rightarrow\neq \exists \sigma' \succeq \sigma : \hat{\mathcal{P}}(\sigma') \rangle \end{array}$$

$$
\left(\exists D \;\middle|\; \begin{array}{ll} : \wedge & (\forall \sigma \in \mathcal{S}' \mid \neg\hat{\mathcal{P}}(\sigma) : (\exists \sigma' \preccurlyeq \sigma \mid : D(\sigma'))) \\ & (\forall \tau; a \in \Sigma^* \mid D(\tau; a) : \hat{\mathcal{P}}(\tau) \wedge (\forall \sigma \in \mathcal{S}' \mid \sigma \succcurlyeq \tau; a \wedge \hat{\mathcal{P}}(\sigma) : \sigma = \tau)) \\ \wedge & \neg D(\varepsilon) \end{array}\right)
$$

$\iff$ ⟨ Theorem 3.4.6 ⟩

$\hat{\mathcal{P}} \in \mathsf{T}^{\mathcal{S}'}_{=}$-enforceable

□

3.4.1 Complexity

A distinctive aspect of the method under consideration is that unlike other code instrumentation methods, ours induces no added runtime overhead. However, the size of the instrumented program is increased in the order $\mathcal{O}(m \times n)$, where m is the size of the original program and n is the size of the property. The worst case would occur if every program instruction modifies the state of the security automaton, a situation which is arguably unlikely. The instrumentation algorithm itself runs in time $\mathcal{O}(p \times c)$, where p is the size of the automaton's acceptance condition and c is the number of cycles in the product automaton. In practice, graphs that abstract programs have a comparatively small number of cycles.

3.5 Conclusion and Future Work

The main contribution of this chapter is the elaboration of a method aiming at in-lining a security enforcement mechanism in an untrusted program. The security property to be enforced is expressed by a Rabin automaton and the program is modeled by a LTS. The inlined monitoring mechanism is actually a truncation mechanism allowing valid executions to run normally while halting bad executions before they violate the property.

In our approach, the monitor's enforcement power is extended by giving it access to statically gathered information about the program's possible behavior. This allows us to enforce non-safety properties for some programs. Nevertheless, several cases still exist where our approach fails to find a suitable instrumented code. These are cases where an execution may alternate between satisfying the property or not and could halt

in an invalid state, or cases where an invalid execution contains no valid prefixes where the execution could be aborted without also ruling out some valid executions.

Another contribution of this chapter is to provide a proof that a truncation mechanism that effectively enforces a security property under the equality as an equivalence relation is strictly more powerful in a non uniform context than in a uniform one.

Giving the monitor more varied means to alter the execution could allow us to ensure the satisfaction of the security property in at least some cases where doing so is currently not feasible. For example, the monitor could suppress a sub-sequence of the program, and keep it under observation until it has determined that the program actually satisfies the property and output it all at once. Alternatively, the monitor could be allowed to insert some actions at the end of an invalid sequence in order to guarantee that the sequence is aborted in a valid state. Such monitors are suggested in [11], their use would extend this approach to a more powerful framework. Another question that remains open is to determine how often the algorithm will succeed in finding a suitable instrumented code when tested on real programs. We are currently developing an implementation to investigate this question further and hope to gain insights as to which of the above suggested extensions would provide the greatest increase in the set of enforceable properties.

Chapter 4

Monitoring With Equivalence Relations

4.1 Introduction

In the last chapter, we proposed a new method to inline a monitor into an untrusted program to produce a new version of this program which provably respects a security property. Any valid execution present in the original program is also present in the instrumented program, while invalid executions are truncated to a valid prefix. Other implementations can be more transformative. For instance, in [63], runtime test are added to both valid and invalid executions. It is thus necessary to ensure that the transformed execution remains equivalent to the original one, despite the transformations performed by the monitor.

The question of identifying the set of properties enforceable by monitors able to transform invalid executions was raised several times in the literature [11, 40, 54, 69]. While these studies observe that this ability considerably extends the monitor's enforcement power, they do not provide a more specific characterization of the set of enforceable properties w.r.t equivalence relations other than syntactic equality. This results from the lack of a framework constraining the ability of a monitor to transform its input. This point is concisely explained by Ligatti et. al. in [54]. "*A major difficulty with semantic equivalence is its generality: for any reasonable property* $\hat{\mathcal{P}}$ *there exists a sufficiently helpful equivalence relation that enables a security automaton to enforce* $\hat{\mathcal{P}}$".

Indeed, the authors go on to note that if all valid sequences can be thought of as being equivalent to one another, any security policy can be enforced simply by always outputting an arbitrarily chosen valid sequence. This strictly meets the definition of enforcement but does not provide a meaningful enforcement of the desired policy.

This problem is compounded by the fact that the definition of effective$_{\cong}$ enforcement does not place any restriction on the output of the monitor when the observed sequence does not respect the property, as long as this output is itself valid. This means that once the monitor determines that a sequence is irremediably invalid, it can ignore the target program's behavior and output anything, even if this output is completely unrelated to the observed execution. Once again, such a behavior would fit the definition of enforcement, but would not provide a useful enforcement of the desired policy, where one would prefer that an invalid sequence be corrected in a more systematic or predictable manner.

This point was also raised in the literature, in [18] Bielova et al. write: "*What distinguishes an enforcement mechanism is not what happens when traces are good, because nothing should happen! The interesting part is how precisely bad traces are converted into good ones.*".

For example, consider a system managing online purchases, and a security policy forbidding a user from browsing certain merchandize without prepaying. A monitor could abort the execution as soon as this is attempted. But the property would also be enforced by replacing the input sequence with any sequence of actions respecting the policy, even if it contains purchases unrequested by any users, or by outputting nothing, depriving legitimate users of the ability to use the system.

In this chapter, we suggest a framework to study the enforcement power of monitors capable of transforming their input. The key insight behind our work is to state certain criteria which must be met for an equivalence relation to be useful in monitoring. We then give two examples of such equivalence relations, and show which security properties are enforceable with their use. The research presented in this chapter has been presented at the Fifth International Conference Mathematical Methods, Models, and Architectures for Computer Networks Security, and published in the conference's proceedings [42].

Intuitively, this enforcement parading models a behavior that is closer to that which would be encountered in practice, in which the actions taken by the monitor are constrained by a limitation that certain elements present in the original sequence be preserved. In this chapter, we give two examples of such equivalence relations, and show

which security properties are enforceable with their use.

The contributions of this chapter are as follows: first, we develop a framework of enforcement, termed corrective$_{\cong}$, enforcement to reason about the enforcement power of monitors bounded to produce an output which is semantically equivalent to their input with respect to some equivalence relation $\cong$. We suggest two possible examples of such relations and give the set of enforceable security policies as well as examples of real policies for each. Finally, we show that the set of enforceable properties defined in [54] for effective enforcement can be considered as special cases of our more general framework.

The remainder of this chapter is organized as follows. In Section 2, we define some concepts and notations that are used throughout the chapter. In Section 3, we show under what conditions equivalence relations can be used to transform sequences and ensure the satisfaction of the security policy. The set of security policies that can be enforced in this manner is examined in Section 4. In Section 5, we give two examples of possible equivalence relations and show that they can serve as the basis for the enforcement of meaningful security properties. In section 6, we investigate how an a priori knowledge of the target program's behavior would increase the monitor's enforcement power. In section 7, we discuss some limitations of our framework. Concluding remarks and avenues for future work are laid out in Section 8.

4.2 Preliminaries

The notation used to describe and manipulate executions, properties and systems is the same was used in previous chapters.

A multiset, or bag [73] is a generalization of a set in which each element may occur multiple times. A multiset $\mathcal{A}$ can be formally defined as a pair $\langle A, f\rangle$ where A is a set and $f : A \to \mathbb{N}$ is a function indicating the number of occurrences of each element of A in $\mathcal{A}$. Note that $a \notin A \Leftrightarrow f(a) = 0$. Thus, by using this insight, to define basic operations on multisets one can consider a universal set A and different functions of type $A \to \mathbb{N}$ associated with it to form different multisets. Given two multisets $\mathcal{A} = \langle A, f\rangle$ and $\mathcal{B} = \langle A, g\rangle$, the multiset union $\mathcal{A} \cup \mathcal{B} = \langle A, h\rangle$ where $\forall a \in A : h(a) = f(a) + g(a)$. Furthermore, $\mathcal{A} \subseteq \mathcal{B} \Leftrightarrow \forall a \in A : f(a) \leq g(a)$. The removal of an element $a \in A$ from multiset $\mathcal{A}$ is done by updating function f so that $f(a) = \max(f(a) - 1, 0)$.

- the multiset intersection $\mathcal{A} \cap \mathcal{B} = \langle A, h\rangle$ where $\forall a \in A : h(a) = \min(f(a), g(a))$,

- the multiset removal of $\mathcal{B}$ from $\mathcal{A}$, noted $\mathcal{A} \backslash \mathcal{B} = \langle A, h \rangle$ where $\forall a \in A : h(a) = \max(f(a) - g(a), 0)$.

Finally, we formalize the set of *transactional* properties, suggested in [54], which will be of use in Section 4.5. A transactional property is one where any valid sequence consists of a concatenation of valid finite transactions. Such properties can model, for example, the behavior of systems which repeatedly interact with clients using a well-defined protocol, such as a system managing the allocation of resource or the access to a database. Let Σ be an action set and $\mathcal{T} \subseteq \Sigma^*$ be a set of finite transactions, $\hat{\mathcal{P}}_{\mathcal{T}}$ is a transactional property over set $\mathcal{T}$ iff

$$\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}_{\mathcal{T}}(\sigma) \Leftrightarrow \sigma \in \mathcal{T}^\infty \qquad \text{(transactional)}$$

This definition is subtly different, and indeed forms a subset of the set of iterative properties defined in [18]. Transactional properties also form a subset of the set of Renewal properties, and include some but not all safety properties, liveness properties as well as properties which are neither safety nor liveness.

4.3 Monitoring with Equivalence Relations

The idea of using equivalence relations to transform execution sequences was first suggested in [40]. The equivalence relations are restricted to those that are *consistent* with the security policy under consideration. Let $\hat{\mathcal{P}}$ be a security policy, the consistency criterion for an equivalence relation $\cong$ is given as:

$$\forall \sigma, \sigma' \in \Sigma^\infty : \sigma \cong \sigma' \Rightarrow \hat{\mathcal{P}}(\sigma) \Leftrightarrow \hat{\mathcal{P}}(\sigma'). \qquad \text{(consistency)}$$

Yet, upon closer examination, this criterion seems too restrictive for our purposes. If any two equivalent sequences always meet this criterion, an invalid prefix can never be made valid by replacing it with another equivalent one. It is thus impossible to "correct" an invalid prefix and output it.

It is still necessary to impose some restrictions on equivalence relations and their relation to properties. Otherwise, as discussed above, any property would be enforceable, but not always in a meaningful manner.

In this study, we suggest the following alternative framework.

Following previous work in monitoring by Fong [36], we use an abstraction function $\mathcal{F} : \Sigma^* \to \mathcal{I}$, to capture the property of the input sequence which the monitor must preserve throughout its manipulation. The set $\mathcal{I}$ can be any abstraction of the program's behavior. Fong focuses on shallow (unordered) history of the execution and makes use of this abstraction to reduce the overhead of the monitor. In the following sections, we suggest other abstractions and show how they can be used as the basis for our equivalence relations. Such abstractions can capture any property of relevance. This may be, for example, the presence of certain subwords or factors or any other semantic property of interest. We expect the property to be consistent with this abstraction rather than with the equivalence relation itself. Formally:

$$\mathcal{F}(\sigma) = \mathcal{F}(\sigma') \Rightarrow \hat{\mathcal{P}}(\sigma) \Leftrightarrow \hat{\mathcal{P}}(\sigma') \tag{4.3.1}$$

We wish to use the abstraction to restrict the possible behavior of the monitor. To this end, we let $\leq$ stand for some partial order over the values of $\mathcal{I}$. We define $\sqsubseteq$ as the partial order defined as $\forall \sigma, \sigma' \in \Sigma^* : \sigma \sqsubseteq \sigma' \Leftrightarrow \mathcal{F}(\sigma) \leq \mathcal{F}(\sigma')$. We equivalently write $\sigma' \sqsupseteq \sigma$ and $\sigma \sqsubseteq \sigma'$.

The transformation performed by the monitor on a given sequence τ produces a new sequence τ' s.t. $\tau' \sqsubseteq \tau$. To ease the monitor's task in finding such a suitable replacement, we impose the following two constraints on the equivalence relations used in monitoring.

First, if two sequences are equivalent, any intermediary sequence over $\sqsubseteq$ is also equivalent to them.

$$\sigma \sqsubseteq \sigma' \sqsubseteq \sigma'' \wedge \sigma \cong \sigma'' \Rightarrow \sigma \cong \sigma' \tag{4.3.2}$$

Second, two sequences cannot be equivalent if they do not share a common greatest lower bound. Conversely, the greatest lower bound of two equivalent sequences is also equivalent to them. These last two criteria are stated together as:

$$\forall \sigma, \sigma' \in \Sigma^* : \sigma \cong \sigma' \Rightarrow \exists \tau \in \Sigma^* : \tau = (\sigma \sqcap \sigma') \wedge \tau \cong \sigma \tag{4.3.3}$$

where $(\sigma \sqcap \sigma') = \tau$ s.t. $\tau \sqsubseteq \sigma \wedge \tau \sqsubseteq \sigma' \wedge \neg \exists \tau' \sqsupseteq \tau : \tau' \sqsubseteq \sigma \wedge \tau' \sqsubseteq \sigma'$

The intuition behind the above two restrictions is that, if an equivalence relation meets these two criteria, a monitor looking for a valid sequence equivalent to an invalid input simply has to iteratively perform a certain transformation until such a sequence is found or until every equivalent sequence has been examined.

We define our equivalence relations over finite sequences first. Two infinite sequences are equivalent iff they have infinitely many valid equivalent prefixes.

Let $\cong$ be an equivalence relation over the sequences of Σ^*

$$\forall \sigma, \sigma' \in \Sigma^\omega : \sigma \cong \sigma' \Leftrightarrow \forall \tau \prec \sigma : \exists \upsilon \succeq \tau : \exists \tau' \prec \sigma' : \upsilon \cong \tau' \tag{4.3.4}$$

It is easy to see that an equivalence between infinite sequence not meeting this criterion would be of no use to a monitor, which is bound to transform its input in finite time.

Finally, we impose the following closure restriction:

$$\tau \cong \tau' \Rightarrow \tau; \sigma \cong \tau'; \sigma \tag{4.3.5}$$

This may, at first sight, seem like an extremely restrictive condition to be imposed but in fact every meaningful relation that we examined has this property.

Furthermore, no security property can be enforced using an equivalence relation lacking this property. Consider for example what would happen if a monitor is presented with an invalid prefix τ of an input sequence for which there exists a valid equivalent sequence τ'. It would be natural for the monitor to transform τ into τ'. Yet it would also be possible that the full original sequence $\sigma \succ \tau$ be actually valid, but that there exists no equivalent sequence for which τ' is a prefix.

In fact, $\sqsubseteq$ organizes the sequences according to some semantic framework, using values given by an abstraction function $\mathcal{F}$, $\hat{\mathcal{P}}$ establishes that only certain values of $\mathcal{F}$ are valid or that a certain threshold must be reached, while $\cong$ groups the sequences if their abstractions are equivalent. In Section 4.5, we give examples that show how the framework described in this section can be used to model desirable security properties of programs and meaningful equivalence relations between their executions.

4.4 Corrective Enforcement

In this section, we present the automata-based model used to study the enforcement mechanism, and give a more formal definition of our notion of enforcement. We begin by reviewing some definitions we previously gave in Chapter 2, to better illustrate the novelty of our approach.

Recall that the edit automaton [11, 54] is the most general model of a monitor. It captures the behavior of a monitor capable of inserting or suppressing any action, as well as halting the execution in progress.

Definition 4.4.1. *An edit automaton is a tuple* $\langle \Sigma, Q, q_0, \delta \rangle$ *where*[1]*:*

[1]This definition, taken from [76], is equivalent to the one given in [11].

- Σ *is a finite or countably infinite set of actions;*
- Q *is a finite or countably infinite set of states;*
- $q_0 \in Q$ *is the initial state;*
- $\delta : (Q \times \Sigma) \rightarrow (Q \times \Sigma^\infty)$ *is the transition function, which, given the current state and input action, specifies the automaton's output and successor state. At any step, the automaton may accept the action and output it intact, suppress it and move on to the next action, outputting nothing, or output some other sequence in* Σ^∞*. If at a given state the transition for a given action is undefined, the automaton aborts.*

Let $\mathcal{A}$ be an edit automaton, we let $\mathcal{A}(\sigma)$ be the output of $\mathcal{A}$ when its input is σ.

Most studies on this topic have focused on effective enforcement. A mechanism effectively enforces a security property iff it respects the two following principles, from [11]:

1. *Soundness* : All output must respect the desired property.
2. *Transparency* : The semantics of executions that already respect the property must be preserved. This naturally requires the use of an equivalence relation, stating when one sequence can be substituted for another.

Definition 4.4.2. *Let* $\mathcal{A}$ *be an edit automaton.* $\mathcal{A}$ *effectively*$_\cong$ *enforces the property* $\hat{\mathcal{P}}$ *iff* $\forall \sigma \in \Sigma^\infty$

1. $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$ *(i.e.* $\mathcal{A}(\sigma)$ *is valid)*
2. $\hat{\mathcal{P}}(\sigma) \Rightarrow A(\sigma) \cong \sigma$

The above definition is equivalent to the one given in Chapter 2.

In the literature, the only equivalence relation $\cong$ for which the set of effectively$_\cong$ enforceable properties has been formally studied is syntactic equality[11]. Yet, effective enforcement is only one paradigm of enforcement that has been suggested.

In this study, we introduce a new paradigm of security property enforcement, termed corrective$_\cong$ enforcement. An enforcement mechanism correctively$_\cong$ enforces the desired property if every output sequence is both valid and equivalent to the input sequence.

This captures the intuition that the monitor is both required to output a valid sequence, and forbidden from altering the semantics of the input sequence. Indeed, it is not always reasonable to accept, as do preceding studies of monitor's enforcement power, that the monitor is allowed to replace an invalid execution with any valid sequence, even ϵ. A more intuitive model of the desired behavior of a monitor would rather require that only minimal alterations be made to an invalid sequence, for instance by releasing a resource or adding an entry in a log. Those parts of the input sequence that are valid, should be preserved in the output, while invalid behaviors should be corrected or removed. It is precisely these corrective behaviors that we seek to model using our equivalence relations. The enforcement paradigm thus ensures that the output is always valid, and that all valid behavior intended by the user in the input, is present in the monitor's output.

Definition 4.4.3. *Let $\mathcal{A}$ be an edit automaton and let $\cong$ be an equivalence relation satisfying 4.3.2- 5.2.2. $\mathcal{A}$ correctively$_{\cong}$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \Sigma^{\infty}$*

1. $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$
2. $\mathcal{A}(\sigma) \cong \sigma$

A monitor can correctively$_{\cong}$ enforce a property iff for every possible sequence there exists an equivalent valid sequence which is either finite or has infinitely many valid prefixes, and the transformation into this sequence is decidable. We write enforceable$_{\cong}$ for the set of properties which are correctively$_{\cong}$ enforceable.

Theorem 4. *A property $\hat{\mathcal{P}}$ is correctively$_{\cong}$ enforceable iff*

1. $\exists \hat{\mathcal{P}}' : \hat{\mathcal{P}}' \subseteq \hat{\mathcal{P}} \wedge \hat{\mathcal{P}}' \subseteq$ *Renewal*
2. $\hat{\mathcal{P}}$ *is reasonable*
3. *There exists a decidable function* $\gamma : \Sigma^{\infty} \rightarrow \hat{\mathcal{P}}' : \forall \sigma \in \Sigma^{\infty} : \gamma(\sigma) \cong \sigma$.
4. $\forall \sigma' \preceq \sigma : \gamma(\sigma') \preceq \gamma(\sigma)$

Proof. (if direction) By construction of the following automaton.
$\mathcal{A} = \langle \Sigma, Q, q_0, \delta \rangle$ where

- $Q = \Sigma^*$, the sequence of actions seen so far.

- $q_0 = \epsilon$
- The transition function δ is given as $\delta(\sigma, a) = (\sigma; a, \gamma(\sigma; a))$

  Note that from Condition 3 of Theorem 4 we have that $\gamma(\sigma; a)$ is always defined, and from condition 4 that it takes the recursive form described above.

The automaton maintains the following invariants INV(q): At state $q = \sigma$, $\gamma(\sigma)$ has been output so far, this output is valid and equivalent to σ.
The invariant holds initially, as by definition, ϵ is valid and equivalent to itself. An induction can then show that the invariant is preserved by the transition relation.

(elseif direction) Let $\gamma(\sigma)$ be whatever the automaton outputs on input σ. By definition, γ is a decidable function. Furthermore, we have that $\hat{\mathcal{P}}(\sigma)$ and $\mathcal{A}(\sigma) \cong \sigma$.

We need to show that the image of γ is a property $\hat{\mathcal{P}}'$ included in $\hat{\mathcal{P}}$ and in Renewal. That the image of γ is a subset of $\hat{\mathcal{P}}$ follows trivially from the assumptions $\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}(\mathcal{A}(\sigma))$. Furthermore, were the output not in Renewal, it would include valid sequences with only finitely many valid prefixes. Yet, since the automaton's transition function is restricted to outputing finite valid sequences by the requirement that the finite input be equivalent to the output and equation 4.3.4 , this is impossible. It follows that the image of γ is a subset of $\hat{\mathcal{P}}$ and Renewal. It is also easy to see that $\hat{\mathcal{P}}(\epsilon)$, since if it were not the case, a violation would occur even in the absence of any input action. Finally, since γ is applied recursively to every prefix of the input, it is thus unavoidable that $\forall \sigma' \preceq \sigma : \gamma(\sigma') \preceq \gamma(\sigma)$. □

An equivalence relation $\cong$ over a given set Σ^* can be seen as a set of pairs (x, y), with $x, y \in \Sigma^*$. This allows equivalence relations over the same sets to be compared. Relation $\cong_1$ is a refinement of relation $\cong_2$, noted $\cong_1 < \cong_2$ if the set of pairs in $\cong_1$ is a strict subset of those in $\cong_2$.

Theorem 5. *Let $\cong_1$, $\cong_2$ be two equivalence relations and let enforceable$_\cong$ stand for the set of correctively$_\cong$ enforceable properties, then $\cong_1 < \cong_2 \Rightarrow$ enforceable$_{\cong_1} \subset$ enforceable$_{\cong_2}$.*

Proof. It is easy to see that any property which is correctively$_{\cong_1}$ enforceable is also correctively$_{\cong_2}$ enforceable, since every pair of sequences that are equivalent w.r.t. $\cong_1$ are also equivalent w.r.t. $\cong_2$. The property can thus be correctively$_{\cong_2}$ enforced using the same transformation function γ as was used in its correctively$_{\cong_1}$ enforcement.

Let $[\sigma]_{\cong}$ stand for the set of sequences equivalent to σ with respect to relation $\cong$. By assumption, there is a σ s.t. $[\sigma]_{\cong_1} \subset [\sigma]_{\cong_2}$. Let $\hat{\mathcal{P}}$ be the property defined s.t. $\neg\hat{\mathcal{P}}(\tau) \Leftrightarrow \tau \in [\sigma]_{\cong_1}$. This property is not correctively$_{\cong_1}$ enforceable as there exists no valid equivalent sequences which the monitor can output when its input is σ. The property can be correctively$_{\cong_2}$ enforced by outputting a sequence in $[\sigma]_{\cong_2} \backslash [\sigma]_{\cong_1}$ when the input is σ. □

It follows from this theorem that the coarser the equivalence relation used by the monitor, the greater the set of enforceable$_{\cong}$ properties.

The following lemma is used in the next section to set an upper bound to the set of enforceable properties with specific equivalence relations.

Lemma 6. *Let $\cong$ be an equivalence relation and $\hat{\mathcal{P}}$ be some correctively$_{\cong}$ enforceable property. Then, for all $\hat{\mathcal{P}}'$ s.t. $\hat{\mathcal{P}} \subseteq \hat{\mathcal{P}}'$ we have that $\hat{\mathcal{P}}'$ is correctively$_{\cong}$ enforceable.*

The monitor has only to simulate its enforcement of $\hat{\mathcal{P}}$ in order to correctively$_{\cong}$ enforce $\hat{\mathcal{P}}'$.

4.5 Equivalence Relations

In this section, we consider two examples of the equivalence relation $\cong$, and examine the set of properties enforceable by each.

4.5.1 Factor equivalence

The first equivalence relation we will consider is factor equivalence, which models the class of transactional properties introduced in section 4.2. A word $\tau \in \Sigma^*$ is a factor of a word $\omega \in \Sigma^\infty$ if $\omega = \upsilon;\tau;\upsilon'$, with $\upsilon \in \Sigma^*$ and $\upsilon' \in \Sigma^\infty$. Two sequences τ, τ' are factor equivalent, w.r.t. a given set of valid factors $\mathcal{T} \subseteq \Sigma^*$ if they both contain the same multiset of factors from $\mathcal{T}$. We use a multiset rather than simply comparing the set of factors from $\mathcal{T}$ occurring in each sequence so as to be able to distinguish between sequences containing a different number of occurrences of the same subset of factors. This captures the intuition that if certain valid transactions are present in the input sequence, they must still be present in the output sequence, regardless of any other transformation made to ensure compliance with the security property. In

this context, the desired behavior of the system can be defined by a multiset of valid transactions. A valid run of this system consists of a finite or infinite sequence of well-formed transactions, while an invalid sequence is a sequence containing invalid or incomplete transactions. One may reasonably consider all sequences exhibiting the same multiset of valid transactions to be equivalent to each other. Transactional properties form a subset to the class of Renewal properties which can be effectively$_{=}$ enforced [54]. In [17], Bielova et. al. propose an alternate enforcement paradigm, which allows all valid transactions to be output. Corrective$_{\cong}$ enforcement can be seen as a generalization of their work.

Let $valid_{\mathcal{T}}(\sigma)$, which stands for the multiset of factors from the sequence σ which are present in $\mathcal{T}$, be the abstraction function $\mathcal{F}$. The partial order $\sqsubseteq$ used to correctively enforce this property is thus given as $\forall \sigma, \sigma' \in \Sigma^{\infty} : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_{\mathcal{T}}(\sigma) \subseteq valid_{\mathcal{T}}(\sigma')$. Intuitively, a sequence is smaller than another on the partial order if it has strictly fewer transactions. Finally, two sequences σ and σ' are *transaction equivalent* w.r.t. the set of transactions $\mathcal{T}$, noted $\sigma \cong_{\mathcal{T}} \sigma'$ iff they they share the same valid transactions. Formally, $\forall \sigma, \sigma' \in \Sigma^{\infty} : \sigma \cong_{\mathcal{T}} \sigma' \Leftrightarrow valid_{\mathcal{T}}(\sigma) = valid_{\mathcal{T}}(\sigma')$. This equivalence relation captures the intuition that any valid transaction present in the original sequence must also be present in the monitor's output.

For example, let $\Sigma = \{\texttt{open}, \texttt{close}, \texttt{log}\}$ be a set of atomic actions and let $\mathcal{T} = \{\texttt{open}; \texttt{log}; \texttt{close}\}$ be the set containing the only allowed transaction. If the input sequence is given as $\sigma = \texttt{log}; \texttt{open}; \texttt{log}; \texttt{close}; \texttt{log}; \texttt{open}; \texttt{close}; \texttt{open}; \texttt{log}; \texttt{close}$, then $valid_{\mathcal{T}}(\sigma)$ is the multiset containing two instances of the factor $\texttt{open}; \texttt{log}; \texttt{close}$.

We now turn our attention to the set of properties that are correctively$_{\cong_{\mathcal{T}}}$ enforceable. Intuitively, a monitor can enforce this property by first suppressing the execution until it has seen a factor in $\mathcal{T}$, at which point the factor is output, while any invalid transaction is suppressed. This method of enforcement is analogous to the one described in [17] as delayed all-or-nothing enforcement. Any sequence output in this manner would preserve all its factors in $\mathcal{T}$, and thus be equivalent to the input sequence, but is composed of a concatenation of factors from $\mathcal{T}$, and hence is valid.

Let $\mathcal{T} \subseteq \Sigma^*$ be a set of factors and let $\hat{\mathcal{P}}_{\mathcal{T}}$ be a transactional property as defined in section 4.2. Note first that all properties enforceable by this approach are in Renewal, as they are formed by a concatenation of valid finite sequences. Also, the property must necessarily be reasonable, (i.e. $\hat{\mathcal{P}}(\epsilon)$) as the monitor will not output anything if the input sequence does not contain any factors in $\mathcal{T}$. Finally, for the property $\hat{\mathcal{P}}_{\mathcal{T}}$ to be correctively$_{\cong_{\mathcal{T}}}$ enforceable in the manner described above, the following restriction, termed *unambiguity* must be imposed on $\mathcal{T}$:

$$\forall \sigma, \sigma' \in \mathcal{T} : \forall \tau \in \mathit{pref}(\sigma) : \forall \tau' \in \mathit{suf}(\sigma') : \tau \neq \epsilon \wedge \tau' \neq \epsilon \Rightarrow \tau'; \tau \notin \mathcal{T} \quad \text{(unambiguity)}$$

To understand why this restriction is necessary, consider what would happen in its absence: it would be possible for the monitor to receive as input a sequence which can be parsed either as the concatenation of some valid transactions, or as a different valid transaction bracketed with invalid factors. That is, let $\sigma_1; \sigma_2 = \tau_1; \sigma_3; \tau_2$ be the monitor's input, with $\sigma_1, \sigma_2, \sigma_3 \in \mathcal{T}$ and $\tau_1, \tau_2 \notin \mathcal{T}$. If the monitor interprets the sequence as a concatenation of the valid transactions σ_1 and σ_2, then it has to preserve both factors in its output. However, if it parses the sequence as $\tau_1; \sigma_3; \tau_2$, then it must output only the equivalence sequence σ_3. Since the two sequences are syntactically identical, the monitor has no information of which to base such a decision.

Theorem 7. *A transactional property $\hat{\mathcal{P}}_{\mathcal{T}}$ is correctively$_{\cong_{\mathcal{T}}}$ enforceable if it is transactional, reasonable, and $\mathcal{T}$ is unambiguous.*

Proof. From theorem 4, the property $\hat{\mathcal{P}}$ is correctively$_{\cong}$ enforceable iff there exists a function $\gamma : \Sigma^\infty \to \hat{\mathcal{P}}'$, where $\hat{\mathcal{P}}$' is a subset of $\hat{\mathcal{P}}$ and is in Renewal, $\hat{\mathcal{P}}$ is reasonable, and $\forall \sigma, \sigma' \preceq \sigma \in \Sigma^* : \gamma(\sigma') \preceq \gamma(\sigma) \wedge \gamma(\sigma) \cong \sigma$. We prove this theorem by exhibiting such a function.

Let $\gamma : \Sigma^\infty \to \hat{\mathcal{P}}_{\mathcal{T}}$. We define γ recursively as follows. $\forall \sigma \in \Sigma^\infty$.

$$\gamma(\sigma) = \begin{cases} \tau; \gamma(\sigma') & \text{if there exists a } \tau \text{ in } \mathcal{T} : \sigma = \tau; \sigma' \\ \gamma(\sigma[2..]) & \text{otherwise} \end{cases}$$

It is easy to show that the image of this function is $\hat{\mathcal{P}}_{\mathcal{T}}$, as all transactions not in $\mathcal{T}$ are deleted. The image is also in Renewal as $\mathcal{T}$ is formed by a concatenation of finite sequences, and any infinite valid sequence not in Renewal would necessarily have finitely many valid prefixes. Conversely, the output of γ is $\cong_{\mathcal{T}}$ equivalent to its input since only factors not in $\mathcal{T}$ are removed. From the fact that γ is applied to the input iteratively we have $\forall \sigma, \sigma' \in \Sigma^\infty : \sigma' \preceq \sigma : \gamma(\sigma') \preceq \gamma(\sigma)$. Transactional properties are reasonable by definition. □

We have only to refer to lemma 6 in order to state a precise upper bound to the set of enforceable properties.

Theorem 8. *A property $\hat{\mathcal{P}}$ is correctively$_{\cong_\mathcal{T}}$ enforceable iff $\hat{\mathcal{P}}_\mathcal{T} \subseteq \hat{\mathcal{P}}$ and $\mathcal{T}$ is unambiguous.*

Proof. (if direction) Follows directly from theorem 7 and lemma 6.
(if direction) We show that every sequence in $\hat{\mathcal{P}}_\mathcal{T}$ must be present in any correctively$_{\cong_\mathcal{T}}$ enforceable property by contradiction. Let $\sigma \in \hat{\mathcal{P}}_\mathcal{T}$ be an input sequence such that $\neg\hat{\mathcal{P}}(\sigma)$. The monitor may not enforce the property by removing or adding a transaction in $\mathcal{T}$ to σ, as the output would no longer be equivalent to the input. To understand why the monitor would be incapable of outputing a valid sequence containing exactly the same transactions as σ, even if one such sequence exists consider the following. Let $\sigma' \cong_\mathcal{T} \sigma \wedge \hat{\mathcal{P}}(\sigma')$. Let τ be the longest common prefix of σ and σ'. For it to be possible that σ' be both $\in \hat{\mathcal{P}}_\mathcal{T}$ and equivalent to σ, there must be at least two sequences $\tau', \tau'' \in \mathcal{T}$ which are appended to τ in a different order to produce σ and σ'. Without loss of generality, let τ' occur first in σ and second in σ'. Let the input sequence be the invalid sequence $\tau;\tau';\omega$ where ω is an infinite suffix which does not contain any valid transactions. After having output τ, the monitor may not output τ' as this would result in an invalid sequence if the following valid transactions in the input occur in the same order as they do in σ. Yet if it does not output τ', the output sequence is not $\cong_\mathcal{T}$ equivalent to the input.

Likewise, let $\mathcal{T}$ be not unambiguous. By assumption there exists valid sequences $\sigma_1, \sigma_2, \sigma_3 \in \mathcal{T}$ and invalid sequences $\tau, \tau' \notin \mathcal{T}$ s.t. $\tau;\sigma_3;\tau' = \sigma_1;\sigma_2$ and $\sigma_1, \sigma_2, \sigma_3$ are valid transactions. Let the infinite sequence $\sigma_1;\upsilon;\sigma_2;\upsilon;\sigma_1;\upsilon...$ be the input sequence, where υ does not contain nay valid transactions (possibly $\upsilon = \tau$ or $\upsilon = \tau'$). There cannot be a valid equivalent sequence since any concatenation $\sigma_1;\sigma_2$ also contains the transaction σ_3. Such a property is thus unenforceable. □

4.5.2 Prefix Equivalence

In this section, we show that Ligatti et al.'s result from [54], namely that the set of properties effectively$_=$ enforceable by an edit automaton corresponds to the set of reasonable Renewal properties with a computability restriction added[2], can be stated as a special case of our framework.

[2]Actually, the authors identified a corner case in which a property that is not in the set described above. This occurs when the monitor reaches a point where only one valid continuation is possible. The input can then be ignored and this single continuation is output. We have neglected to discuss this case here as it adds comparatively little to the range of enforceable properties.

First, we need to align our definitions of enforcement. Using effective enforcement, they only require that the monitor's output be equivalent to its input when the latter is valid, and while placing no such restriction on the output otherwise. The semantics of their monitor however, do impose that the output remain a prefix of the input in all cases, and indeed, that the longest valid prefix always be output [33]. This characterization can be translated in our formalism by instantiating $\cong$ to $\cong_{\preceq} \stackrel{def}{=} \forall \sigma, \sigma' \in \Sigma^* : \sigma \cong_{\preceq} \sigma' \Leftrightarrow pref(\sigma) \cap \hat{\mathcal{P}} = pref(\sigma') \cap \hat{\mathcal{P}}$. Using this relation, two sequences are equivalent, w.r.t. a given property $\hat{\mathcal{P}}$ iff they have the same set of valid prefixes.

Theorem 9. *A property $\hat{\mathcal{P}}$ is effectively$_{=}$ enforceable iff it is correctively$_{\cong_{\preceq}}$ enforceable.*

Proof. (if direction) From [54], we have that a property is effectively$_{=}$ enforceable iff 1) $\hat{\mathcal{P}}(\sigma) \Rightarrow \mathcal{A}(\sigma) = \sigma$ and 2) $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$. Conditions 2 is present in identical form in the definition of corrective enforcement. For condition 1, we must consider two cases. If $\hat{\mathcal{P}}(\sigma)$, then it is trivial to show that $\sigma \cong_{\preceq} \mathcal{A}(\sigma)$, since both sequences are syntactically identical. Otherwise, the semantics of the enforcement mechanism described in [54] ensure that the longest valid prefix is output. It follows that $pref(\sigma) \cap \hat{\mathcal{P}} = pref(\mathcal{A}(\sigma)) \cap \hat{\mathcal{P}}$ and from the definition of $\cong_{\preceq}$, that $\sigma \cong_{\preceq} \mathcal{A}(\sigma)$.
(else if direction) We must show that $\sigma \cong_{\preceq} \mathcal{A}(\sigma) \wedge \hat{\mathcal{P}}(\sigma) \Rightarrow \mathcal{A}(\sigma) = \sigma$. It is sufficient to observe that if a sequence σ is valid, there can exist no $\sigma' \prec \sigma : \sigma' \cong_{\preceq} \sigma$. Since the enforcement mechanism described above only outputs a sequence that is prefix or equal to its input we have that $\sigma \cong_{\preceq} \mathcal{A}(\sigma)$. □

It would be intuitive to instantiate the partial order $\sqsubseteq$ to $\preceq$. Other possibilities can be considered, which would more closely follow the specific property being enforced.

Theorem 10. *A property $\hat{\mathcal{P}}$ is correctively$_{\cong_{\preceq}}$ enforceable iff it is in Renewal, reasonable and decidable.*

Proof. Immediate from theorem 9 and theorem 3 of [54]. □

As discussed in [54], this set includes a wide range of properties, including all safety properties, some liveness properties such as the "eventually audits" properties requiring that an action eventually be logged, and properties which are neither safety nor liveness such as the transactional properties described in section 4.3. Furthermore, if the behavior of the target system is known to consist only of finite executions, then every sequence is in Renewal.

4.6 Nonuniform Enforcement

In this section, we investigate the possibility of extending the set of enforceable properties by giving the monitor some knowledge of the target program's possible behavior. This question was first raised in [69]. In [11], the authors distinguish between the uniform context, in which the monitor must consider that every sequence in Σ^∞ can occur during the target program's execution, from the nonuniform context, in which the set of possible executions is a subset of Σ^∞. They further show that in some case, the set of properties enforceable in a nonuniform context is greater than that which is enforceable in a uniform context. Later Chabot et. al. [22] showed that while this result did not apply to all runtime enforcement paradigms, it did apply to that of truncation-based monitor. Indeed, they show that in this monitoring context, a monitor operating with a subset of Σ^∞ is always more powerful than one which considers that every sequence can be output by its target. In the previous chapter, we reviewed their work and gave the conditions which must be met for it to be possible to extend the set of enforceable properties by static analysis.

Let $\mathcal{S}$ stand for the set of sequences which the monitor considers as possible executions of the target program. $\mathcal{S}$ is necessarily an over approximation, built from static analysis of the target. We write correctively$^{\mathcal{S}}_{\cong}$ enforceable, or just enforceable$^{\mathcal{S}}_{\cong}$, to denote the set of properties that are correctively$_{\cong}$ enforceable, when only sequences from $\mathcal{S} \subseteq \Sigma^\infty$ are possible executions of the target program. A property is correctively$^{\mathcal{S}}_{\cong}$ enforceable iff for every sequence in $\mathcal{S}$, the monitor can return a valid and equivalent sequence.

Definition 4.6.1. *Let $\mathcal{A}$ be an edit automaton and let $\mathcal{S} \subseteq \Sigma^\infty$ be a subset of executions. $\mathcal{A}$ correctively$^{\mathcal{S}}_{\cong}$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \mathcal{S}$*

1. $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$
2. $\mathcal{A}(\sigma) \cong \sigma$

Theorem 11. *A property $\hat{\mathcal{P}}$ is correctively$^{\mathcal{S}}_{\cong}$ enforceable iff*

1. *$\hat{\mathcal{P}}$ is Reasonable*
2. *$\exists \hat{\mathcal{P}}' \subseteq \hat{\mathcal{P}} : \hat{\mathcal{P}}' \in$ Renewal $: (\exists \gamma \in \mathcal{S} \to \hat{\mathcal{P}}' : (\forall \sigma \in \mathcal{S} : \gamma(\sigma) \cong \sigma) \wedge (\forall \sigma, \sigma' \in \mathcal{S} : \sigma' \preceq \sigma \Rightarrow \gamma(\sigma') \preceq \gamma(\sigma)) \wedge \gamma$ is decidable)*

Proof. The proof follows exactly as that of Theorem 4. □

Lemma 12. *Let $\mathcal{S} \subseteq \Sigma^\infty$ and $\hat{\mathcal{P}}$ be a reasonable property $\hat{\mathcal{P}}$ is trivially correctively$^{\mathcal{S}}_{\cong}$ enforceable iff $\mathcal{S} \subseteq \hat{\mathcal{P}}$. If this is the case, the monitor can enforce the property by always returning the input sequence.*

It would be desirable if the set of enforceable properties increased monotonously each time a sequence was removed from $\mathcal{S}$. This means that any effort made to perform or refine a static analysis of the target program would payoff in the form of an increase in the set of enforceable properties. This is unfortunately not the case. As a counterexample, consider the equivalence relation defined as $\forall \sigma, \sigma' \in \Sigma^\infty : \sigma \cong \sigma'$. It is obvious that any satisfiable property can be trivially enforced in this context, simply by always outputting any valid sequence, which is necessarily equivalent to the input. No benefit can then be accrued by restricting $\mathcal{S}$.

There are, of course, some instances where constraining the set $\mathcal{S}$ does result in a increase in the set of correctively$^{\mathcal{S}}_{\cong}$ enforceable properties. This occurs when invalid sequences with no valid equivalent are removed from $\mathcal{S}$. Indeed, for any subsets, $\mathcal{S}, \mathcal{S}'$ of Σ^∞ s.t. $\mathcal{S} \subseteq \mathcal{S}' \wedge \mathcal{S}' \backslash \mathcal{S} \neq \{\epsilon\}$, there exists an equivalence relation $\cong$ for which enforceable$^{\mathcal{S}'}_{\cong} \subset$ enforceable$^{\mathcal{S}}_{\cong}$.

Theorem 13. *Let $\mathcal{S} \subset \mathcal{S}' \subseteq \Sigma^\infty \wedge \mathcal{S}' \backslash \mathcal{S} \neq \{\epsilon\}$. There exists an equivalence relation $\cong$ s.t. enforceable$^{\mathcal{S}'}_{\cong} \subset$ enforceable$^{\mathcal{S}}_{\cong}$.*

Proof. Let $\cong$ be defined s.t. $\exists \sigma \in \mathcal{S}' \backslash \mathcal{S} : \{\sigma\} \cap \mathcal{S} \neq \emptyset$. Let $\hat{\mathcal{P}}$ be the property defined as $\hat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma \notin \mathcal{S} \wedge \sigma \neq \epsilon)$. This property is not enforceable$^{\mathcal{S}'}_{\cong}$ since there exists sequences in $\mathcal{S}'$ with no valid equivalent. The property is trivially enforceable$^{\mathcal{S}}_{\cong}$. □

A final question of relevance on the topic of nonuniform enforcement is whether there exists some equivalence relations $\cong$ for which every reduction of the size of $\mathcal{S}$ monotonously increases the set of properties that are correctively$^{\mathcal{S}}_{\cong}$ enforceable. In other words, if there exists some $\cong$ for which $\mathcal{S} \subset \mathcal{S}' \Rightarrow$ enforceable$^{\mathcal{S}'}_{\cong} \subset$ enforceable$^{\mathcal{S}}_{\cong}$. Anyone operating under such an equivalence relation would have an added incentive to invest in static analysis of the target, as he would be guaranteed an increase in the set of enforceable properties. Unfortunately, it can be shown that this result holds only when $\cong$ is syntactic equality and at least one sequence different from ϵ is removed from the set of possible sequences.

Theorem 14. $(\sigma \cong \sigma' \Leftrightarrow \sigma = \sigma') \Leftrightarrow \forall \mathcal{S}, \mathcal{S}' \subseteq \Sigma^\infty : \mathcal{S} \subset \mathcal{S}' \wedge \mathcal{S}' \backslash \mathcal{S} \neq \{\epsilon\} :$ *enforceable$^{\mathcal{S}'}_{\cong} \subset$ enforceable$^{\mathcal{S}}_{\cong}$*

Proof. (if direction)
Let $\hat{\mathcal{P}}$ be defined such that $\hat{\mathcal{P}}(\sigma) \Leftrightarrow (\sigma \notin \mathcal{S}' \backslash \mathcal{S})$. This property cannot be correctively$^{\mathcal{S}'}_{\cong}$ enforceable since any sequence in $\mathcal{S}' \backslash \mathcal{S}$ does not have a valid equivalent. The property is trivially correctively$^{\mathcal{S}}_{\cong}$ enforceable.
(else if direction)
By contradiction, let $\cong$ be different than syntactic equality. This implies there exists $\sigma, \sigma' \in \mathcal{S}' : \sigma \cong \sigma' \wedge \sigma \neq \sigma'$. Further, let $\mathcal{S}' = \{\sigma, \sigma'\}$ and $S = \{\sigma\}$. We show that any property that is correctively$^{\mathcal{S}}_{\cong}$ enforceable is also correctively$^{\mathcal{S}'}_{\cong}$ enforceable. There are five cases to consider. :

- $\sigma, \sigma' \in \hat{\mathcal{P}}$: In this case, the property is always trivially enforceable.
- $\sigma \in \hat{\mathcal{P}} \wedge \sigma' \notin \hat{\mathcal{P}}$: Such a property would be both correctively$^{\mathcal{S}}_{\cong}$ enforceable and correctively$^{\mathcal{S}'}_{\cong}$ enforceable by automaton $\mathcal{A}$ for which $\mathcal{A}(\tau) = \sigma$ for all τ in the input set.
- $\sigma' \in \hat{\mathcal{P}} \wedge \sigma \notin \hat{\mathcal{P}}$: Such a property would be both correctively$^{\mathcal{S}}_{\cong}$ enforceable and correctively$^{\mathcal{S}'}_{\cong}$ enforceable by automaton $\mathcal{A}$ for which $\mathcal{A}(\tau) = \sigma'$ for all τ in the input set.
- $\sigma, \sigma' \notin \hat{\mathcal{P}} \wedge \exists \sigma'' \cong \sigma : \hat{\mathcal{P}}(\sigma'')$: Such a property would be both correctively$^{\mathcal{S}}_{\cong}$ enforceable and correctively$^{\mathcal{S}'}_{\cong}$ enforceable by automaton $\mathcal{A}$ for which $\mathcal{A}(\tau) = \sigma''$ for all τ in the input set.
- $\sigma, \sigma' \notin \hat{\mathcal{P}} \wedge \neg \exists \tau \cong \sigma : \hat{\mathcal{P}}(\tau)$: This property can neither be correctively$^{\mathcal{S}}_{\cong}$ enforceable nor can it be correctively$^{\mathcal{S}'}_{\cong}$ enforceable since there exists some sequences with no valid equivalent.

Finally, observe that since only reasonable sequences are enforceable, no possible gain can be accrued from removing only ϵ from the set of possible sequences. □

4.7 Limitations

Like all monitors, the ones described in this paper are limited by memory and computational constraints, which we have not taken into account in our analysis. The results given in theorems 8 and 10 should thus be seen as upper bounds to the set of properties that are enforceable using the enforcement paradigm proposed here.

The main limitation of the approach is the difficulty of stating meaningful equivalence relations. This problem is especially acute when the monitor inserts actions into the input stream, rather then suppressing them or truncating the execution. When this is the case, an equivalence relation often implies that several distinct valid sequence, which are possible transformations of an invalid sequences, must be equivalent. Likewise, it requires that several invalid sequences be considered equivalent if a single valid sequence is a valid alternative to both.

For example, consider the possible equivalence relation of a monitor enforcing the transactional property of section 4.5.1, but by correcting invalid transactions, rather than simply suppressing them. Whenever this monitor is presented with an invalid transaction τ, it can complete it by adding whichever actions are necessary to transform this invalid transaction into a valid one. For this to be permissible in a corrective$_{\cong}$ enforcement framework, the completed valid transaction must be thought of as equivalent to its incomplete factor. The equivalence relation given in 4.5.1 is thus no longer adequate. If an invalid sequence τ can be extended into two possible valid transactions, then both of these transactions must be thought of as equivalent. Furthermore, any other invalid sequence which can be corrected by transforming it into one of these valid sequences must in turn be thought of as equivalent to τ.

A possible solution to this problem is to replace equivalence relations with partial orders, thus organizing executions according to a relation which is reflexive, transitive, but not symmetric. Actually, instead of grouping together sequences whose abstraction is similar, a partial order organizes them in a gradual manner. In the example of transactional properties, a sequence with more valid transactions would be higher on the partial order than one with less such transactions. The transparency requirement would then be stated by imposing than the output always be higher on the partial order than the input. Not only would this solve the problem highlighted above but it would allow us to objectively describe one valid execution as better than another, which in turn could lead to comparing execution monitors along these lines. We are currently elaborating such an enforcement framework along these lines.

4.8 Conclusion and Future Work

In this chapter, we propose a framework to analyze the security properties enforceable by monitors capable of transforming their input. By imposing constraints on the enforcement mechanism to the effect that some behaviors existing in the input sequence must still be present in the output, we are able to model the desired behavior of real-life

monitors in a more realistic and effective way. We also show that real life properties are enforceable in this paradigm, and give prefix equivalence and factor equivalence as possible examples of realistic equivalence relations which could be used in a monitoring context. The set of properties enforceable using these two equivalence relations is related to previous results in the field.

Future work will focus on other equivalence relations. Two meaningful equivalence relations which we are currently studying are subword equivalence and permutation equivalence. The first adequately models the behavior of a monitor that is allowed to insert actions into the program's execution, but may not subtract anything from it. The second models the behavior of a monitor which can reorder the actions performed by its target, but may not add or remove any of them. An even more general framework that could be envisioned would be one in which the behavior that the monitor must preserve is stated in a temporal logic.

Chapter 5

Monitoring With Preorders

5.1 Introduction

As was shown in the previous chapters of this study, monitors form powerful security policy enforcement paradigm that can allow the enforcement of a wide variety of security policies. Of particular interest are monitors capable of transforming their input sequences, rather than simply aborting them, as such monitors have been shown to be amongst the most powerful. But for this kind of enforcement to be meaningful, constraints must be imposed on the monitor's ability to transform sequences. Otherwise, the monitor could enforce the property by replacing any execution with an arbitrarily chosen valid sequence.

In the previous chapter, we proposed to solve this problem by sorting out sequences into equivalence classes and imposing that the monitor's output be equivalent to the original sequence in either some or all cases. In this context, the equivalence relation represents a semantic property of the original sequence that must be preserved throughout any transformation performed by the monitor. Although this approach solves the problem described above, it also limits the monitor's ability to take certain corrective actions, since the equivalence between the monitor's inputs and outputs must be always maintained. Furthermore, it is often difficult to define a suitable equivalence relation.

In this chapter, we propose a new framework to model the corrective capabilities of monitors. Our key insight is to organize executions into preorders, rather than equivalence classes, and impose that an execution be replaced only by a sequence that is at least as high on the preorder than itself. As we argue in section 5.2, we find that preorders are a more natural way to model the restrictions we wish to impose

on the monitor than equivalence relations. We also illustrate our framework with four examples of real life properties. Finally, since corrective enforcement is sufficiently flexible to allow several different enforcement alternatives of the same property, we suggest metrics that allow a user to compare monitors objectively and choose the best enforcement paradigm for a given situation. The research presented in this chapter has been presented at the 7th International Workshop on Formal Aspects of Security & Trust (FAST2010) [41].

The remainder of this chapter is organized as follows. Section 2 reviews the various notions of enforcement which have been suggested in the literature, and proposes an alternative enforcement paradigm, based on the notion of preorders. We show how preorders can form the basis of a corrective monitoring framework, and motivate the use of such a framework by comparing it to other enforcement paradigms. In Section 3, we give four examples of security properties that can be enforced in this manner and propose metrics that could be used to compare alternative enforcements of the same security policy. Concluding remarks and avenues for future work are laid out in Section 4.

5.2 Monitoring with Preorders

This chapter extends the body of research that seeks to study the notion of security policy enforcement by monitors and to identify the set of properties enforceable by monitors under various constraints.

Recall that these issues were first investigated by Schneider in [69], who formalized the notions of monitoring and enforcement. He focused on specific classes of monitors that observe the execution of a target program with no knowledge of its possible future behavior and with no ability to affect it, except by aborting the execution. Under these conditions, a monitor can enforce the class of security policies that are identified in the literature as *safety* properties.

In [11], Ligatti, Bauer and Walker show that if this definition of enforcement is used, the added power of some monitors to transform the executions they monitor (by inserting or suppressing program actions) does not result in an increase in the set of enforceable properties. The authors suggest the alternative notion of effective$_{\cong}$ enforcement instead. A monitor effectively$_{\cong}$ enforces a property if any execution respecting the property is replaced by an equivalent execution, w.r.t. some equivalence relation $\cong$. Subsequently, in [54], the authors delineate the set of properties that are effectively$_{\cong}$

enforceable for a specific equivalence relation, syntactic equality. Alternative definitions of enforcement are given in [17] and [56].

Most previous work in monitoring has focused on effective$_{\cong}$ enforcement. This definition allows the monitor to replace an invalid execution with any valid sequence, even ϵ. A more intuitive model of the desired behavior of a monitor would rather require that only minimal alterations be made to an invalid sequence, for instance by releasing a resource or adding an entry in a log. Those parts of the input sequence which are valid should be preserved in the output, while invalid behaviors should be corrected or removed. This is the enforcement paradigm which we proposed in the previous chapter, and which we termed corrective$_{\cong}$ enforcement. Informally, an enforcement mechanism correctively$_{\cong}$ enforces the desired property if every output sequence is both valid and equivalent to the input sequence. The equivalence relation used is formulated in such a way that the valid behavior of the input sequence is preserved.

However, this definition also raises some difficulties. In particular, it implies that several distinct valid sequences, which are possible transformations of an invalid sequences, must be equivalent. Likewise, it requires that several invalid sequences be considered equivalent if a single valid sequence is a valid alternative to both.

In this chapter, we examine an alternative notion of enforcement, termed corrective$_{\sqsubseteq}$ enforcement. Just as we did in the previous chapter, we use an abstraction function $\mathcal{F} : \Sigma^* \rightarrow \mathcal{I}$, to capture the property of the input sequence that the monitor must preserve throughout its manipulation. Once again, we use these abstractions as the basis for determining which transformations the monitor is or is not allowed to perform on both valid and invalid sequences. The main idea is to use this abstraction to capture the semantic property of the execution corresponding to the valid, or desired behavior of the target program. This may be, for example, the number of occurrences of certain subwords or factors or any other semantic property of interest.

We wish to constrain the behavior of the monitor so that an invalid sequence is corrected in such a way that the monitor preserves all valid behaviors present in it. An intuitive solution to this problem would be to impose that the output sequences always have the same value of $\mathcal{F}$. The abstraction function would thus form the basis of a partition of the sequences of Σ^∞ into equivalence classes. But, as discussed above, this limits the monitor's ability to use the same valid sequences as a potential solution to several unrelated invalid sequences. Instead, we let $\sqsubseteq$ be a preorder over sequences of Σ^*, s.t. $\forall \sigma, \sigma' : \sigma \sqsubseteq \sigma' \Leftrightarrow \mathcal{F}(\sigma) \leq \mathcal{F}(\sigma')$. The monitor is allowed to transform an input σ into another sequence σ' iff $\sigma \sqsubseteq \sigma'$. By defining $\sqsubseteq$ appropriately, we can ensure that sequences which are greater on the preorder are always adequate replacements for any

inferior sequences, in the sense that they preserve the valid behaviors present in those sequences. The use of preorders also allows us to connect monitoring to refinement specifications, which are stated using preorders. We also find that partial orders are a more natural way to state most security policies than equivalence relations. Let τ, τ' be two sequences s.t. $\tau \sqsubseteq \tau'$, we write that τ is lower than τ' or conversely, that τ' is higher than τ.

Formally :

Definition 5.2.1. *Let $\mathcal{A}$ be an edit automaton and let $\sqsubseteq$ be a preorder over the sequences of Σ^∞. $\mathcal{A}$ correctively$_\sqsubseteq$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \Sigma^\infty$*

1. $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$
2. $\sigma \sqsubseteq \mathcal{A}(\sigma)$

A monitor often operates in a context in which it knows that certain executions cannot occur. This is because the monitor can benefit from a static analysis of its target, that provides it with a model of the target's possible behavior. Prior research [11] has shown that a monitor operating in such a context can enforce a significantly larger range of properties than one that considers every sequence in Σ^∞ to be a possible input. To take into account the possibility that the monitor might operate in a nonuniform context, we adapt the preceding definition as follows:

Definition 5.2.2. *Let $\mathcal{S} \subseteq \Sigma^\infty$ be a subset of sequences, let $\sqsubseteq$ be a preorder over the sequences of Σ^∞ and let $\mathcal{A}$ be an edit automaton. $\mathcal{A}$ correctively$^{\mathcal{S}}_\sqsubseteq$ enforces the property $\hat{\mathcal{P}}$ iff $\forall \sigma \in \mathcal{S}$*

1. $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$
2. $\sigma \sqsubseteq \mathcal{A}(\sigma)$

We write corrective$_\sqsubseteq$ enforcement when $\mathcal{S} = \Sigma^\infty$ or $\mathcal{S}$ is obvious from context.

As was the case in the previous chapter, we define preorders over finite sequences and two infinite sequences compared by way of their prefixes using the following equation.

$$\forall \sigma, \sigma' \in \Sigma^\omega : \sigma \sqsubseteq \sigma' \Leftrightarrow \forall \tau \prec \sigma : \exists \upsilon \succeq \tau : \exists \tau' \prec \sigma' : \upsilon \sqsubseteq \tau' \tag{5.2.1}$$

Finally, since the monitor operates by transforming sequences, we must impose that every preorder respects the following closure restriction.

$$\tau \sqsubseteq \tau' \Rightarrow \tau;\sigma \sqsubseteq \tau';\sigma \tag{5.2.2}$$

To understand the need for this restriction consider the possible behavior of a monitor which is presented with an invalid an invalid prefix τ of a longer input sequence. It may opt to transform τ into a valid higher sequence τ'. However, if the cloture restriction given in equation 5.2.2 is not respected by the preorder, then it's possible that the full input sequence $\sigma \succ \tau$ is actually valid, but that there is no valid extension of τ' that is greater than σ. The monitor would have inadvertently ruined a valid sequence.

5.3 Examples

In this section, we illustrate the use of corrective$_{\sqsubseteq}$ enforcement using four real-life security properties : transactional properties, the assured pipeline property, the Chinese wall property and general availability. When several possible monitors can enforce the same property, we show how these different enforcement paradigms can be compared.

5.3.1 Transactional Properties

The first class of properties we wish to correctively$_{\sqsubseteq}$ enforce is that of transactional properties introduced in the previous chapter. Recall that if Σ is an action set and $\mathcal{T} \subseteq \Sigma^*$ is a set of finite transactions, $\hat{\mathcal{P}}_{\mathcal{T}}$ is a *transactional* property over set Σ^∞ iff

$$\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}_{\mathcal{T}}(\sigma) \Leftrightarrow \sigma \in \mathcal{T}^\infty \tag{transactional}$$

Transactional properties form a subset of the class of Renewal properties which can be effectively$_{=}$ enforced [54], in a manner that allows the longest valid prefix to be output [12]. In [17], Bielova et. al. propose an alternative enforcement paradigm, that allows all valid transactions to be output. Corrective$_{\sqsubseteq}$ enforcement can be seen as a generalization of their work.

We only consider transactional properties built from a set of sequences $\mathcal{T}$ which meets the unambiguity criterion suggested in the previous chapter.

As was the case when enforcing this property with equivalence relations, we use an abstraction function which returns the multiset of factors occurring in a given sequence rather than simply comparing the set of factors from $\mathcal{T}$ occurring in each sequence so as to be able to distinguish between sequences containing a different number of occurrences of the same subset of factors. For any two sequences σ and σ', we write $\sigma \sqsubseteq \sigma'$ iff σ' has more valid factors (w.r.t. $\mathcal{T}$) than σ, and σ' is thus an acceptable replacement sequence for σ, provided that the σ' is valid and σ is not. This captures the intuition that if certain valid transactions are present in the input sequence, they must also be present in the output sequence, regardless of any other transformation made to ensure compliance with the security property. The monitor may add valid transactions and remove invalid ones, but may not remove any valid transactions present in the original execution.

Let $valid_{\mathcal{T}}(\sigma)$, which stand for the multiset of factors from the sequence σ which are present in $\mathcal{T}$, be the abstraction function $\mathcal{F}$. The preorder $\sqsubseteq$ used to correctively enforce this property is thus given as $\forall \sigma, \sigma' \in \Sigma^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_{\mathcal{T}}(\sigma) \subseteq valid_{\mathcal{T}}(\sigma')$. This preorder captures the intuition that any valid transaction present in the original sequence must also be present in the monitor's output.

The following automaton correctively enforces transactional properties by suppressing any invalid transaction present in the input sequence, while allowing every valid transaction to be output. Let $\mathcal{A}_t = \langle \Sigma, Q, q_0, \delta \rangle$ where

- Σ is a finite or countably infinite action set.
- $Q = \Sigma^* \times \Sigma^*$, is the set of automaton states. Each state consists of a pair $\langle \sigma_o, \sigma_s \rangle$ where σ_o is the sequence which has been output so far, and σ_s is a sequence which the monitor has suppressed, and may either eventually output or delete.
- $q_0 = \langle \epsilon, \epsilon \rangle$, is the initial state.
- The transition function $\delta : Q \times \Sigma \to Q \times \Sigma^\infty$ is given as:

$$\delta(\langle \sigma_o, \sigma_s \rangle, a) = \begin{cases} \langle \sigma_o; \tau, \epsilon \rangle & \text{if } \exists \tau \in suf(\sigma_s; a) : \tau \in \mathcal{T} \wedge \tau \neq \epsilon \\ \langle \sigma_o; \sigma_s; a, \epsilon \rangle & \text{if } \exists \tau \in \mathcal{T} : \exists \tau' \in \mathcal{T}^* : \sigma_o; \sigma_s; a = \tau'; \tau \wedge \\ & |\tau| \geq |\sigma_s; a| \\ \langle \sigma_o, \sigma_s; a \rangle & \text{otherwise} \end{cases}$$

Proposition 5.3.1. *Let $\mathcal{T} \subseteq \Sigma^\infty$ be a subset of sequences, let $\hat{\mathcal{P}}_{\mathcal{T}}$ be the corresponding transactional property and let $\sqsubseteq$ be a preorder over the sequences of Σ^∞ defined such that $\forall \sigma, \sigma' \in \Sigma^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_{\mathcal{T}}(\sigma) \subseteq valid_{\mathcal{T}}(\sigma')$. The automaton $\mathcal{A}_t$ correctively$_{\sqsubseteq}$ enforces $\hat{\mathcal{P}}_{\mathcal{T}}$.*

Proof. By theorem 5.2.1, we have that the automaton correctively$_{\sqsubseteq}$ enforces the property iff $\forall\sigma \in \Sigma^\infty : \mathcal{A}(\sigma) \in \hat{\mathcal{P}}_\mathcal{T} \wedge \sigma \sqsubseteq \mathcal{A}(\sigma)$. It is easy to see that the output of $\mathcal{A}$ is in $\hat{\mathcal{P}}_\mathcal{T}$ since by construction, at every step, the monitor only outputs a sequence if it is in $\mathcal{T}$, or if appending it to the sequence the monitor has already output produces a sequence in $\mathcal{T}^\infty$. Likewise, to see why for all input sequences σ, $\sigma \sqsubseteq \mathcal{A}(\sigma)$, simply observe that any factor τ of σ present in $\mathcal{T}$ will necessarily be output by the monitor. □

The method above thus shows how a transactional property could be enforced in such a manner that the output is always valid, and always contains as many or more valid transactions than the input. In this particular example, we may be able to prove an even stronger enforcement paradigm, namely that the output will always contain exactly the same valid transactions as the input. This is unsurprising, since equivalence relations are a special case of preorders, and imposing such a constraint would be tantamount to using the equivalent$_\cong$ enforcement proposed in section 5.2. The method presented here can thus be seen as a generalization of this framework.

But this example also highlights why equivalent$_\cong$ enforcement is also too rigid to be useful in many practical cases. Consider what would happen, for example, if the restriction that the set $\mathcal{T}$ be unambiguous is lifted [1]. Even though the only transformation performed by the monitor is to remove invalid factors, we cannot guarantee that exactly those valid sequences which are present in the original sequence will be present in the output. As a counterexample, consider the case of valid sequences $\mathcal{T} = \{\sigma_1, \sigma_2, \sigma_3\}$ and invalid sequences $\tau, \tau' \notin \mathcal{T}$ s.t. $\tau;\sigma_3;\tau' = \sigma_1;\sigma_2$. Let the infinite sequences $\sigma_1;\upsilon;\sigma_2;\upsilon;\sigma_1;\upsilon...$ be the input sequence, where υ is an invalid transaction (possibly τ or τ'). The multiset of valid transactions present in $\sigma_1;\sigma_2,\sigma_1;\sigma_2,\sigma_1;\sigma_2...$ contains an infinite number of factors σ_3, not present in the original sequence. While it may be difficult to imagine a real-life, transactional property exhibiting this behavior, this example does illustrate why using equivalence relations rather than a preorder would unduly restrict the transformations available to a monitor. Furthermore, one may wish to consider other means by which a monitor may enforce a transactional property, for instance, by inserting actions to correct an incomplete or transactions, or simply to log the occurrence of certain possibly malicious factors.

5.3.2 Assured Pipelines

In the previous section, we show how transactional properties can be enforced by an edit automaton that simply suppresses some actions from the input stream. Yet, part

[1] This may involve altering the definition of iterative properties.

of the power of the edit automaton resides in its ability to insert actions not present in the input to correct an invalid sequence. Naturally, this ability must be constrained for the enforcement to remain meaningful, otherwise the monitor may simply replace any invalid sequence in its entirety by some unrelated valid sequence. In this section, we propose three possible enforcement paradigms for the Assured Pipeline policy based on truncation, suppression and insertion, and compare them using the metrics given in the previous section.

The assured pipeline property was suggested in [19, 81] to ensure that data transformations are performed in a specific order. Let O be a set of data objects, and S be a set of transformations. We assume that S contains a distinguished member **create**. Finally, let $E : \langle S \times O \rangle$ be a set of access events. $\langle s, o \rangle \in E$ denotes the application of transformation s to the data object o. An assured pipeline policy restricts the application of transformations from S to data objects using an enabling relation $e : S \times S$, with the following two restrictions: the relation e must define an acyclic graph, and the **create** process can only occur at the root of this graph. The presence of a pair $\langle s, s' \rangle$ in e, is represented in the graph by the occurrence of a edge between the vertice s and the vertice s', and indicates that any action of the form $\langle s', o \rangle$ is only allowed if s is the last process that accessed o. Because of the restriction that e must be an acyclic graph, each action $\langle s', o \rangle$ can occur at most once during an execution.

For the purpose of this example, we add another restriction namely that the enabling relation be linear. This condition makes it easier for insertion monitors to add actions to the output, without compromising transparency.

A truncation-based monitor was suggested by Fong [36] and Talhi [76] to enforce this property. Their monitors effectively$_=$ enforces the assured pipeline property by aborting the execution if an unauthorized data transaction is encountered. Following their ideas, we propose the automaton $\mathcal{A}^t_{ap}$ bellow which enforces the assured pipelines in this manner. $\mathcal{A}^t_{ap} = \langle E, Q, q_0, \delta_t \rangle$ where

- E is a set of access events over objects from O and transformations from S, as defined above.
- $Q : \wp(E)$ is the state space. Each state is an unordered set of access events from E.
- $q_0 = \emptyset$ is the initial state.
- $\delta_t : Q \times E \to Q \times E \cup \{\epsilon\}$ is given as :

$$\delta_t(q, \langle s, o\rangle) = \begin{cases} (q \cup \{\langle s, o\rangle\}, \langle s, o\rangle) & \text{if } s = create \wedge \langle create, o\rangle \notin q \\ (q \cup \{\langle s, o\rangle\}, \langle s, o\rangle) & \text{if } \exists s' \in S : \langle s', o\rangle \in q \wedge \langle s', s\rangle \in e \wedge \\ & \langle s, o\rangle \notin q \\ \text{undefined} & \text{otherwise} \end{cases}$$

Proposition 5.3.2. *Let e be an enabling relation, defining an assured pipeline policy $\hat{\mathcal{P}}$ over a set of objects O and subjects S. The automaton $\mathcal{A}^t_{ap}$ effectively$_=$ enforces $\hat{\mathcal{P}}$.*

Proof. By definition 4.4.2, we have that the automaton effectively$_=$ enforces the property iff $\forall \sigma \in \Sigma^\infty : \mathcal{A}(\sigma) \in \hat{\mathcal{P}} \wedge, \hat{\mathcal{P}}(\sigma) \Rightarrow \mathcal{A}(\sigma) = \sigma$. Since the execution is aborted as soon as an illicit action is encountered, we necessarily have $\hat{\mathcal{P}}(\mathcal{A}(\sigma))$. Conversely, since valid actions are always output in lockstep with the input, we have $\hat{\mathcal{P}}(\sigma) \Rightarrow \mathcal{A}(\sigma) = \sigma$. □

At each step of the execution, the monitor either outputs the current action if it is valid, or else aborts by attempting an undefined transition otherwise. It follows that if the input sequence is invalid, the monitor outputs its longest valid prefix. Can this enforcement paradigm be improved upon? A corrective enforcement framework allows the monitor to continue the execution after an illicit transformation has been attempted.

As was the case with transactional properties, the preorder defining the desired behavior of the program is stated in terms of the presence of valid actions, i.e. those occurring in a manner allowed by the enabling relation. Since each action can only occur once, a function returning the set of valid atomic actions is an adequate abstraction function. We write $valid_e(\sigma)$ for the set of valid transformations (w.r.t. enabling relation e) occurring in σ. We write $\sigma \sqsubseteq \sigma' \Leftrightarrow valid_e(\sigma) \subseteq valid_e(\sigma')$.

Instead of simply aborting the execution, a corrective monitor may suppress an invalid action, and allow the execution to proceed. We only need to make minor adjustments to $\mathcal{A}^t_{ap}$ to create automaton $\mathcal{A}^s_{ap}$, which enforces the property in this manner.

Let $\mathcal{A}^s_{ap} = \langle E, Q, q_0, \delta_s\rangle$ where Σ, Q and q_0 are defined as in $\mathcal{A}^t_{ap}$ and δ_s is given as follows.

$$\delta_s(q, \langle s, o\rangle) = \begin{cases} (q \cup \{\langle s, o\rangle\}, \langle s, o\rangle) & \text{if } s = create \wedge \langle create, o\rangle \notin q \\ (q \cup \{\langle s, o\rangle\}, \langle s, o\rangle) & \text{if } \exists s' \in S : \langle s', o\rangle \in q \wedge \langle s', s\rangle \in e \wedge \\ & \langle s, o\rangle \notin q \\ (q, \epsilon) & \text{otherwise} \end{cases}$$

Proposition 5.3.3. *Let e be an enabling relation, defining an assure pipeline policy $\hat{\mathcal{P}}$ over a set of actions E, and let $\sqsubseteq$ be a preorder over the sequences of E^∞ defined such that $\forall \sigma, \sigma' \in E^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_e(\sigma) \subseteq valid_e(\sigma')$. The automaton $\mathcal{A}^s_{ap}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}$.*

Proof. By definition 5.2.1, we have that the automaton correctively$_\sqsubseteq$ enforces the property iff $\forall \sigma \in \Sigma^\infty : \mathcal{A}(\sigma) \in \hat{\mathcal{P}} \wedge \sigma \sqsubseteq \mathcal{A}(\sigma)$. Soundness is ensured by the fact that the automaton only outputs valid actions. Furthermore, since every valid action occurring in the input sequence is immediately output, we also have $\sigma \sqsubseteq \mathcal{A}(\sigma)$. □

The execution thus either outputs an action if it is valid, or it suppresses it before allowing the execution to continue. Yet, the edit automaton is capable of not only suppressing or outputting the actions present in the input, but also of adding actions not present in the execution to the input. It may be reasonable, for instance, for a monitor to suppress only those transformations which have already occurred, or those manipulating an object which has not yet been created. Otherwise, if an action occurs in the input sequence before the actions preceding it in e have occurred, the monitor can enforce the property by adding the required actions. The following automaton $\mathcal{A}^e_{ap}$ enforces the property as described above.

To simplify the notation, we use the predicate $path_e(\tau)$ to indicate that τ is a factor of a valid sequence according to e and that every action in τ manipulates the same object. Formally , $path_e(\tau) \Leftrightarrow$

- $\exists o \in O : \forall \langle s, o' \rangle \in acts(\tau) : o' = o$
- $\forall i : 2 \leq i \leq |\tau| : \tau_{i-1} = \langle s, o \rangle \wedge \tau_i = \langle s', o \rangle \Rightarrow \langle s, s' \rangle \in e$

Let $\mathcal{A}^e_{ap} = \langle E, Q, q_0, \delta_e \rangle$ where Σ, Q and q_0 are defined as in $\mathcal{A}^e_{ap}$ and δ_e is given as follows.

$$\delta_e(q, \langle s, o \rangle) = \begin{cases} (q \cup \{\langle s, o \rangle\}, \langle s, o \rangle) & \text{if } s = create \wedge \langle create, o \rangle \notin q \\ (q \cup \{\langle s, o \rangle\}, \langle s, o \rangle) & \text{if } \exists s' \in S : \langle s', o \rangle \in q \wedge \langle s', s \rangle \in e \wedge \\ & \langle s, o \rangle \notin q \\ (q \cup acts(\tau; \langle s, o \rangle), \tau; \langle s, o \rangle) & \text{if } \langle s, o \rangle \notin q \wedge \langle create, o \rangle \in q \\ & \text{and } \tau \text{ is the longest sequence s.t.} \\ & \tau_0 \notin q \wedge path(\tau; \langle s, o \rangle) \\ (q, \epsilon) & \text{otherwise} \end{cases}$$

Proposition 5.3.4. *Let e be an enabling relation, defining an assured pipeline policy $\hat{\mathcal{P}}$ over a set of actions E, and let $\sqsubseteq$ be a preorder over the sequences of E^∞ defined such that $\forall \sigma, \sigma' \in E^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow valid_e(\sigma) \subseteq valid_e(\sigma')$. The automaton $\mathcal{A}^e_{ap}$ correctively$_\sqsubseteq$ enforces $\hat{\mathcal{P}}$.*

Proof. Similarly to the proof of theorem 5.3.3, soundness is ensured by the fact that the automaton only outputs valid sequences while the fact that every valid action occurring in the input sequence is immediately output, implies that $\sigma \sqsubseteq \mathcal{A}(\sigma)$. □

We now have three possible execution monitors for the assured pipeline policy, two of them are based upon corrective enforcement and the other one on effective$_=$ enforcement. In what follows, we propose metrics that allow us to compare these enforcement paradigms, and enable us to select the most adequate monitor given each situation.

The first metric which we consider is based on the preorder used by the monitor to correctively$_\sqsubseteq$ enforce the property. Since this preorder is designed to capture the desired behavior of the target program, it is natural to also rely upon it to compare enforcement paradigm among themselves.

Definition 5.3.5. *Let $\sqsubseteq$ be a preorder over a set of sequences from Σ^∞, let $\mathcal{S} \subseteq \Sigma^\infty$ be a set of input sequences and let $\mathcal{A}, \mathcal{A}'$ be an edit automata enforcing the same property. $\mathcal{A} \sqsubseteq^{\mathcal{S}} \mathcal{A}' \Leftrightarrow \forall \sigma \in \mathcal{S} : \mathcal{A}(\sigma) \sqsubseteq \mathcal{A}'(\sigma)$. We write $\mathcal{A} \equiv^{\mathcal{S}} \mathcal{A}' \Leftrightarrow \mathcal{A} \sqsubseteq^{\mathcal{S}} \mathcal{A}' \wedge \mathcal{A}' \sqsubseteq^{\mathcal{S}} \mathcal{A}$ and $\mathcal{A} \sqsubset^{\mathcal{S}} \mathcal{A}' \Leftrightarrow \mathcal{A} \sqsubseteq^{\mathcal{S}} \mathcal{A}' \wedge \neg(\mathcal{A}' \sqsubseteq^{\mathcal{S}} \mathcal{A})$ and $\mathcal{A} \equiv^{\mathcal{S}} \mathcal{A}' \Leftrightarrow \mathcal{A} \sqsubseteq^{\mathcal{S}} \mathcal{A}'$.*

Proposition 5.3.6. *Let Σ be a set of atomic actions, $\mathcal{A}^t_{ap} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^s_{ap} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^e_{ap}$*

Proof. For any valid sequence, all three monitors behave similarly, and output the input sequence in lockstep. Let σ be an invalid sequence, it necessarily contains at least one invalid action a. Upon encountering this action, $\mathcal{A}^t_{ap}$ aborts while $\mathcal{A}^s_{ap}$ suppresses a. It follows that $\mathcal{A}^t_{ap}(\sigma) = \mathcal{A}^s_{ap}(\sigma)$ iff a is the last action of the sequence if every subsequent action is also invalid. Otherwise, if there are valid actions left in the sequence, these actions are included in $\mathcal{A}^s_{ap}(\sigma)$ but not in $\mathcal{A}^t_{ap}(\sigma)$. In both cases, we have $\mathcal{A}^t_{ap}(\sigma) \sqsubseteq \mathcal{A}^s_{ap}(\sigma)$ and thus $\mathcal{A}^t_{ap} \sqsubseteq \mathcal{A}^s_{ap}$.

When faced with an an invalid action a, $\mathcal{A}^e_{ap}$ behaves in the same manner as $\mathcal{A}^s_{ap}$ if this action represents a transformation which has already been output. Otherwise, $\mathcal{A}^e_{ap}$ inserts in the output the transformations needed for a to be valid. If these actions occur after a in the input sequence, they are suppressed by $\mathcal{A}^e_{ap}$ (since they have already been output). In all cases, we have $\mathcal{A}^s_{ap}(\sigma) \sqsubseteq \mathcal{A}^e_{ap}(\sigma)$ and thus $\mathcal{A}^s_{ap} \sqsubseteq \mathcal{A}^e_{ap}$.

□

In some cases, one may favor a monitor that allows as much as possible of the input sequence to be output, while also preserving the ordering between these input actions. This is particularity desirable when the user is not assumed to be malicious, as is the case, for example, of a monitor allocating resources between trusted users. A monitor $\mathcal{A}$ is more permissive than another monitor $\mathcal{A}'$, noted $\mathcal{A} \trianglelefteq_p^{\mathcal{S}} \mathcal{A}'$, iff, for any input sequence in a set $\mathcal{S}$, the output of $\mathcal{A}$ contains more actions present in the input, regardless of any other actions that are inserted.

A finite word $\tau \in \Sigma^*$ is said to be a subsword of a word ω, noted $\tau \lhd \omega$ iff $\tau = a_0a_1a_2a_3...a_k$ and $\omega = \Sigma^*a_0\Sigma^*a_1\Sigma^*a_2\Sigma^*a_3...\Sigma^*a_k\Sigma^\omega$ where Σ^* and Σ^ω are used to denote sequences from these sets. The set of subwords of sequence σ is given as $sub(\sigma)$. Let τ, σ be sequences form Σ^*. We write $cs_\tau(\sigma)$ to denote the longest subword of τ which is also a subword of σ.

Definition 5.3.7. *Let $\mathcal{S} \subseteq \Sigma^\infty$ be a set of input sequences and let $\mathcal{A}, \mathcal{A}'$ be edit automata enforcing the same property. $\mathcal{A} \trianglelefteq_p^{\mathcal{S}} \mathcal{A}' \Leftrightarrow$*

- $\forall \sigma \in \mathcal{S} \cap \Sigma^* : |cs_\sigma(\mathcal{A}(\sigma))| \leq |cs_\sigma(\mathcal{A}'(\sigma))|$
- $\forall \sigma \in \mathcal{S} \cap \Sigma^\omega : \forall \tau \prec \sigma : \exists \tau' \preceq \sigma : \tau' \succeq \tau : |cs'_\tau(\mathcal{A}(\tau'))| \leq |cs'_\tau(\mathcal{A}'(\tau'))|$

We write $\mathcal{A} \triangleleft_p^{\mathcal{S}} \mathcal{A}' \Leftrightarrow \mathcal{A} \trianglelefteq_p^{\mathcal{S}} \mathcal{A}' \wedge \neg(\mathcal{A}' \trianglelefteq_p^{\mathcal{S}} \mathcal{A})$ and $\mathcal{A} \equiv_p^{\mathcal{S}} \mathcal{A}' \Leftrightarrow \mathcal{A} \trianglelefteq_p^{\mathcal{S}} \mathcal{A}' \wedge \mathcal{A}' \trianglelefteq_p^{\mathcal{S}} \mathcal{A}$.

Proposition 5.3.8. *Let Σ be a set of atomic actions, $\mathcal{A}^t_{ap} \trianglelefteq_p^{\Sigma^\infty} \mathcal{A}^s_{ap} \trianglelefteq_p^{\Sigma^\infty} \mathcal{A}^e_{ap}$*

Proof. For any valid sequence, all three monitors behave in the same manner, and output the input sequence in lockstep. Let σ be an invalid input sequence, $\mathcal{A}^t_{ap}$ will output the longest valid prefix of σ. This prefix will also be output by $\mathcal{A}^s_{ap}(\sigma)$, as well as any subsequent valid action. It follows that $\mathcal{A}^t_{ap} \trianglelefteq_p^{\Sigma^\infty} \mathcal{A}^s_{ap}$.

Any subword of σ present in $\mathcal{A}^s_{ap}(\sigma)$ is also be present in the output of $\mathcal{A}^e_{ap}(\sigma)$ since the transitions function δ_e outputs strictly more actions from the input than δ_s. The output of $\mathcal{A}^e_{ap}$ may, however, contain subwords of σ not present in $\mathcal{A}^s_{ap}(\sigma)$, since δ_e sometimes outputs actions which are suppressed by δ_s. □

5.3.3 Chinese Wall

As a third example, we consider the Chinese Wall policy, suggested in [20] to avoid conflicts of interest. In this model, a user which accesses a data object o is forbidden to simultaneously accessing certain other data objects that are identified as being in conflict with o. Several implementations of this model have been suggested in the literature. In this study, we consider the framework of Sobel et al. [72], which includes a useful notion of data relinquishing.

Let S be a set of subjects, and O a set of objects. The set of conflicts of interests is given as a set of pairs $C : O \times O$. The presence of $(o_i, o_j) \in C$ indicates that objects o_i and o_j conflict. Naturally, C is a symmetric set ($(o_i, o_j) \in C \Leftrightarrow (o_j, o_i) \in C$). For all objects $o_i \in O$, we write C_{o_i} for the set of objects which are in conflict with o_i. Let $O' \subseteq O$ be a subset of objects, overloading the notation we write $C_{O'}$ for $\bigcup_{o \in O'} C_o$. An action of the form (**acq** $, s, o$) indicates that subject s holds the right to acquire the right to access object o. After this action is performed the subject can freely access the resource but can no longer access an object which conflicts with o.

It can often be too restrictive to impose on subjects that they never again access any data that conflict with any data objects they have previously accessed. In practice, the involvement of a subject with a given entity will eventually come to an end. Once this occurs, the subject should no longer be prevented from collaborating with the competitors of his former client. In [72], the model is enriched along this line by giving subjects the capacity to relinquish previously acquired data. This can be modeled by the action (rel, s, o), indicating that subject s relinquishes a previously accessed object o. After this action occurs in a sequence, s is once again allowed to access object that conflict with o, as long as they do not conflict with any other objects previously accessed by s and not yet relinquished. To simplify the notation, we define the function $live : S \times \Sigma^* \to \wp(O)$ as follows : let s be a subject and τ be a finite sequence, $live(s, \tau)$ return the set of objects which s has accessed in τ and has not yet released.

In the presence of a relinquish action, the security property predicate is stated in the following manner: $\forall \sigma \in \Sigma^\infty : \hat{\mathcal{P}}(\sigma) \Leftrightarrow$

$$\forall s \in S : \forall o \in O : \forall i \in N : \sigma_i = (\textbf{access}\ , s, o) : \neg \exists o' \in live(s, \sigma[..i-1]) : o \in C_{o'}$$

Where $\Sigma = \{\textbf{access,rel}\} \times S \times O$ is the set of atomic actions. Both Fong [36] and Talhi [76] suggest that this property be enforced by truncation, using an automaton that aborts the execution as soon as a conflicting data access is attempted. The following automaton enforces the property in this manner; and its transition function only allows

a action to be output if it is a release action, or if it is an access action which does not cause a conflict of interest to occur. This automaton always either immediately output the action it has received (if this is a release action, or an access to which does not cause a conflict of interest to occur) or else the execution is aborted because the transition is undefined.

$\mathcal{A}^t_{cw} = \langle E, Q, q_0, \delta_t \rangle$ where

- Σ : {**access,rel**} $\times S \times O$ is the set of all possible access and release events over objects from O and subjects from S,
- $Q : \Sigma^*$ is the state space. Each state is the finite sequence which has been output so far.
- $q_0 = \epsilon$ is the initial state.
- $\delta_t : Q \times \Sigma \to Q \times \Sigma \cup \{\epsilon\}$ is given as :

$$\delta_t(q,a) = \begin{cases} (q;a,a) & \text{if } a = (\textbf{access}\ , s, o) \wedge o \notin \mathit{live}(s,q) \\ (q;a,a) & \text{if } a = (\textbf{rel}\ , s, o) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Proposition 5.3.9. *Let C be a set of conflicts of interests, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^t_{cw}$ effectively$_=$ enforces $\hat{\mathcal{P}}_C$.*

Proof. The proof of this proposition proceeds exactly as that of proposition 5.3.2. □

Since the monitor enforces the property by truncation, any authorized data access present in the input sequence after a conflicting data access has occurred is absent from the output sequence. Corrective enforcement can provide a more flexible enforcement paradigm. First, we define the preorder $\sqsubseteq$. Analogously to the preorder proposed in section 5.3.1 to weed out invalid transactions while preserving valid ones, we impose that authorized data accesses be preserved while conflicting ones be deleted. Let σ be a sequence and let C be the corresponding conflict of interest class, we write $\mathit{valid}_C(\sigma)$ for the multiset of actions from σ occurring in that sequence in a manner consistent with C. The preorder is defined as $\forall \sigma, \sigma' \in \Sigma^\infty : \sigma \sqsubseteq \sigma' \Leftrightarrow \mathit{valid}_C(\sigma) \sqsubseteq \mathit{valid}_C(\sigma')$. The most intuitive manner to correctively$_\sqsubseteq$ enforce this property is to suppress conflicting data access, but allow the execution to proceed afterward. Only a slight adjustment needs to be made to $\mathcal{A}^t_{cw}$ to create a monitor enforcing the Chinese wall property in this manner. The monitor $\mathcal{A}^s_{cw}$ thus behaves likes $\mathcal{A}^t_{cw}$ in all cases except when a conflicting action is encountered, where it suppresses the execution rather than abort.

Let $\mathcal{A}^s_{cw} = \langle E, Q, q_0, \delta_s \rangle$ where Σ, Q and q_0 are defined as in $\mathcal{A}^t_{cw}$ and $\delta_s$is given as follows.

$$\delta_s(q,a) = \begin{cases} (q; a, a) & \text{if } a = (\textbf{access }, s, o)) \wedge o \notin \mathit{live}(s,q) \\ (q; a, a) & \text{if } a = (\textbf{rel }, s, o) \\ (q, \epsilon) & \text{otherwise} \end{cases}$$

Proposition 5.3.10. *Let C be a set of conflicts of interests, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^s_{ap}$ correctively$_{\sqsubseteq}$ enforces $\hat{\mathcal{P}}_C$.*

Proof. The proof of this proposition proceeds exactly as that of proposition 5.3.3. □

As discussed above, the involvement of any subject with any given entity eventually ends. When this occurs, objects which conflict with that entity become available to this subject for access, provided that they do not conflict with any other object currently held by this subject. Thus it is natural to suggest an alternative enforcement paradigm in which the monitor keeps track of data access that have been denied, and inserts them after a release action makes them available to the subject which has requested them. In fact, access to conflicting objects is delayed rather than suppressed.

The automaton $\mathcal{A}^e_{cw}$ enforces the property in this manner.

$\mathcal{A}^e_{cw} = \langle E, Q, q_0, \delta_e \rangle$ where

- $\Sigma : \{access, rel\} \times S \times O$ is the set of all possible access and release events over objects from O and subjects from S,
- $Q : \Sigma^* \times \Sigma^*$ is the state space. Each state is a pair $\langle \sigma_o, \sigma_s \rangle$ of finite sequences, where σ_o is the sequence which has been output so far, and σ_s is the sequence which has been seen and suppressed.
- $q_0 = \langle \epsilon, \epsilon \rangle$ is the initial state.
- $\delta_e : Q \times \Sigma \to Q \times \Sigma \cup \{\epsilon\} \times \Sigma^\infty$ is given as :

$$\delta_e(\langle \sigma_o, \sigma_s \rangle, a) = \begin{cases} (\langle \sigma_o; a, \sigma_s \tau \rangle) & \text{if } a = (\textbf{access }, s, o)) \wedge o \notin \mathit{live}(s, \sigma_o) \\ (\langle \sigma_o; a; \tau, \sigma_s \backslash \tau \rangle) & \text{if } a = (\textbf{rel }, s, o) \rangle \text{and } f(s, \sigma_o, \sigma_s) = \tau \\ (\langle \sigma_o, \sigma_s; a \rangle) & \text{otherwise} \end{cases}$$

where the function f examines the sequences have been suppressed and output up to that point and returns a sequence composed of all actions which have previously been suppressed, but can now be output without causing a conflict. This automaton operates in the following manner: As long as the actions it receives as input as access actions which do not cause a conflict of interest to occur, the input actions are output immediately. When a conflict of interest does occur, the monitor suppresses the requested access, and stores it in memory. Finally, when a release action is encountered, the targeted resource is released, and the monitor searches through its memory and outputs any data access stored therein for which there no longer is any conflict of interest.

Proposition 5.3.11. *Let C be a set of conflicts of interests, and let $\hat{\mathcal{P}}_C$ be the corresponding Chinese Wall property. The automaton $\mathcal{A}^e_{ap}$ correctively$_{\sqsubseteq}$ enforces $\hat{\mathcal{P}}_C$.*

Proof. By theorem 5.2.1, we have that the automaton correctively$_{\sqsubseteq}$ enforces the property iff $\forall \sigma \in \Sigma^\infty : \mathcal{A}^e_{ap}(\sigma) \in \hat{\mathcal{P}} \wedge \sigma \sqsubseteq \mathcal{A}^e_{ap}(\sigma)$. Every output of $\mathcal{A}^e_{ap}$ is in the property since the transition function of $\mathcal{A}^e_{ap}$ only outputs valid factors. Any authorized access present in the input sequence is immediately output, which guarantees that $\forall \sigma \in \Sigma^\infty : \sigma \sqsubseteq \mathcal{A}^e_{ap}(\sigma)$ □

Once again, we can compare the three proposed enforcement paradigms using several metrics. Let us begin with the question of which is more corrective, and which is more conservative.

Proposition 5.3.12. $\mathcal{A}^t_{cw} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^s_{cw} \sqsubseteq^{\Sigma^\infty} \mathcal{A}^e_{cw}$

Proof. By theorem 5.2.1, we have that the automaton correctively$_{\sqsubseteq}$ enforces the property iff $\forall \sigma \in \Sigma^\infty : \mathcal{A}(\sigma) \in \hat{\mathcal{P}} \wedge \sigma \sqsubseteq \mathcal{A}(\sigma)$. Since, δ_e only outputs valid factors, we necessarily have that $\mathcal{A}(\sigma) \in \hat{\mathcal{P}}$. Furthermore, since every valid action present in the input sequence is output, it follows necessarily that $\forall \sigma \in \Sigma^\infty : \sigma \sqsubseteq \mathcal{A}^e_{cw}(\sigma)$ □

Proposition 5.3.13. $\mathcal{A}^t_{cw} \trianglelefteq_p^{\Sigma^\infty} \mathcal{A}^s_{cw} \trianglelefteq_p^{\Sigma^\infty} \mathcal{A}^e_{cw}$

Proof. That $\mathcal{A}^t_{cw} \trianglelefteq_p^{\Sigma^\infty} \mathcal{A}^s_{cw}$ follows immediately from the automata's transition functions, since $\mathcal{A}^s_{cw}$ always either outputs the same sequence as $\mathcal{A}^t_{cw}$, or outputs a strictly longer sequence. Likewise, $\mathcal{A}^e_{cw}$'s transition function always outputs any action, and thus any subword, output by $\mathcal{A}^s_{cw}$, and may also output some more. This occurs when an action is suppressed by $\mathcal{A}^e_{cw}$, and later output after a release action allows it. □

Another aspect that must be taken into consideration while selecting an enforcement paradigm is the amount of memory that the monitor may require. The monitor may need to store the execution that has occurred so far, or an abstraction of it, as well as any sequence that has been suppressed, but not yet output. The question of how much memory is required for the enforcement was first raised by Ligatti et al. [54] and first addressed by Fong [36] by way of the Shallow history automata. Later Tahli et al. [76] examine the enforcement power of a memory bounded monitor, while Beauquier et. al. [12] considered that of a finite automaton.

Since different enforcement paradigms for the same security property can vary w.r.t. the amount of memory they require, it is natural that this factor also be used to compare enforcement mechanisms.

Definition 5.3.14. *Let $mem(\mathcal{A})$ stand for the space efficiency of the monitoring algorithm $\mathcal{A}$ (measured in O notation). Let $\mathcal{A}, \mathcal{A}'$ be edit automaton enforcing the same property. We write $\mathcal{A} \trianglelefteq_m^{\mathcal{S}} \mathcal{A}' \Leftrightarrow mem(\mathcal{A}) \subseteq mem(\mathcal{A}')$.*

We write $\mathcal{A} \triangleleft_m^{\Sigma^\infty} \mathcal{A}' \Leftrightarrow \mathcal{A} \trianglelefteq_m^{\Sigma^\infty} \mathcal{A}' \wedge \neg(\mathcal{A}' \trianglelefteq_m^{\Sigma^\infty} \mathcal{A})$ and $\mathcal{A} \equiv_m^{\Sigma^\infty} \mathcal{A}' \Leftrightarrow \mathcal{A} \trianglelefteq^{\Sigma^\infty} \mathcal{A}' \wedge \mathcal{A}' \trianglelefteq \mathcal{A}$.

Proposition 5.3.15. $\mathcal{A}_{cw}^e \triangleleft_m^{\Sigma^\infty} \mathcal{A}_{cw}^s \equiv_m^{\Sigma^\infty} \mathcal{A}_{cw}^t$

Proof. Follows immediately from the automata constructions given above. Observe that $\mathcal{A}_{cw}^s$ and $\mathcal{A}_{cw}^t$ necessitate the same amount of memory since $\mathcal{A}_{cw}^s$ simply deletes conflicting data accesses, with no intent to reinsert them later. Thus this automaton does not keep track of the set of suppressed data accesses. □

5.3.4 General Availability

The final security property that we examine is general availability; a policy requiring that any resource that is acquired is eventually released. Despite its apparent simplicity, the property is remarkably difficult to monitor in a system with more than one resource, and is given by Ligatti et. al. as an example of a property that is not effectively$_=$ enforceable.

Modifying Ligatti's formulation slightly, we define the property as follows. Let $(ac, i), (use, i)$ and (rel, i) stand for accessing, using and releasing a resource i from a set of resources $\mathcal{I}$. The set of possible actions is given as $\Sigma : \{ac, use, rel\} \times \mathcal{I}$. The property states that only acquired resources are used, and that any acquired resource

is eventually released. Note that this seemingly straightforward property combines a liveness component that cannot be monitored [54], with a safety component.

From the onset of the study of formal monitors, there has been an interest in examining how restricting the set of possible executions can increase that of enforceable properties. The intuition is that if the monitor knows, from an a priori static analysis, that certain execution paths cannot occur in its target, it should be able to use this information to enforce properties that would otherwise not be enforceable. This question was first raised in [69], and was later addressed in [11, 56].

It this section, we use the availability property proposed above to show that an a priori knowledge of the program's possible execution paths not only increases the set of properties enforceable by the monitor, but also to provide a different, and possibly preferable, enforcement of a given property. First, consider the difficulties facing a monitor trying to enforce the general availability property in a context in which $\mathcal{S} = \Sigma^\infty$. The monitor may not simply abort an invalid execution since the liveness component implies that some invalid sequences can be corrected. Nor can it suppress a potentially invalid sequence only to output it when a valid prefix is reached. As observed in [54], an infinite sequence of the form $(ac, 1); (ac, 2); (rel, 1); (ac, 3); (rel, 2); (ac, 4); (rel, 3)...$ is valid but has no finite valid prefix. The property can only be enforced by a monitor transforming the input more aggressively.

Since the desired behavior of the program is given in terms of the presence of (use, i) actions bracket by ac_i and (rel, i) actions, a preorder based on the number of occurrences of such actions is a natural way to compare sequences. We can designate such *use* actions as valid. Let the function $valid(\sigma)$, which returns the multiset of actions (use, i) which occur in sequence σ and are both preceded by a (ac, i) action and followed by a (rel, i) action be the abstraction function $\mathcal{F}$. We thus have that $\forall \sigma, \sigma' \in \Sigma^* : \sigma \sqsubseteq \sigma' \Leftrightarrow valid(\sigma) \leq valid(\sigma')$.

A trivial way to enforce this property is for the monitor to insert an (ac, i) action prior to each (use, i) action, and follow it with a (rel, i) action. The (ac, i) and rel_i actions present in the original sequence can than simply be suppressed, as every (use, i) action is made available by a pair of actions (ac, i) and (rel, i) inserted by the monitor. While theoretically feasible, it is obvious that a monitor which opens and closes a resource after every program step would be of limited use in practice. Furthermore, for many applications, a sequence of the form $(ac, i); (use, i); (use, i); (rel, i)$ cannot be considered equivalent to the sequence $(ac, i); (use, i); (rel, i); (ac, i); (use, i); (rel, i)$. We thus limit this study to monitors which insert (ac, i) actions a finite number of times.

We first propose a monitoring framework capable of enforcing the general availability property in a uniform context, where every sequence in Σ^∞ can occur in the target program. A monitor can enforce the property in this context by suppressing every acquire action, as well as subsequent use actions for the same resource, and output them only when a release action is reached. Any use action not preceded by a corresponding acquire action is simply suppressed and never inserted again. A monitor enforcing the property in this manner needs to keep track only of the actions that have been suppressed and not output so far. The automaton $\mathcal{A}_{ga}^{\Sigma^\infty}$ enforces the property as described above.

Let $\mathcal{A}_{ga}^{\Sigma^\infty} = \langle \Sigma, Q, q_0, \delta_e \rangle$ where

- $\Sigma : \{aq, rel, use\} \times \mathcal{I}$ is the set of all possible acquire, release and use actions for all objects;
- $Q : \Sigma^*$ is the sequence which has been suppressed so far;
- $q_0 = \epsilon$ is the initial state;
- $\delta_{\Sigma^\infty} : Q \times \Sigma \to Q \times \Sigma \cup \{\epsilon\}$ is given as:

$$\delta_e(\tau, a) = \begin{cases} (\tau, \epsilon) & \text{if } a = (use, i) \wedge (ac, i) \notin acts(\tau) \\ (\tau; a, \epsilon) & \text{if } a = (ac, i) \vee (a = (use, i) \wedge \\ & (ac, i) \in acts(\tau)) \\ (\tau \backslash (f_i(\tau)), f_i(\tau); (rel, i)) & \text{if } a = (rel, i) \end{cases}$$

The purpose of the function f_i is simply to examine the suppressed sequence and retrieve the actions over resource i.

Proposition 5.3.16. *The automaton $\mathcal{A}_{ga}^{\Sigma^\infty}$ correctively$_\sqsubseteq$ enforces the general availability property.*

Proof. It is easy to see that the output of $\mathcal{A}_{ga}^{\Sigma^\infty}$ is always valid, since the monitor only outputs finite valid factors. Furthermore, the transition function outputs every valid *use* action, as soon as it is released in the input sequence. This ensures that the property is correctively$_\sqsubseteq$ enforced. □

A static analysis can often determine that all computations of a program are *fair* meaning certain actions must occur infinitely often in infinite paths. For the purposes of the general availability property, a static analysis could determine that any action

(ac, i) is eventually followed by a (rel, i) action, or by the end of the execution. The property can thus be violated only by the presence of *use* actions that are not preceded by a corresponding *ac* action. If the monitor is operating in a fair context, a new, more conservative enforcement method becomes possible, as is illustrated by automaton $\mathcal{A}_{ga}^{fair}$.

Let $\mathcal{A}_{ga}^{fair} = \langle \Sigma, Q, q_0, \delta_e \rangle$ where Σ is defined as above and

- $Q : \wp(\mathcal{I})$ is the set of resources which have been acquired but not yet released;
- $q_0 = \emptyset$ is the initial state;
- $\delta_{\Sigma^\infty} : Q \times \Sigma \to Q \times \{\Sigma \cup \epsilon\}$ is given as :

$$\delta_e(q,a) = \begin{cases} (q, \epsilon) & \text{if } a = (use, i) \wedge (ac, i) \notin acts(\tau) \\ (q \cup \{i\}, a) & \text{if } a = (ac, i) \\ (q \backslash i, a) & \text{if } a = (rel, i) \\ (q, a) & \text{if } a = (use, i) \wedge i \in q \\ (\epsilon, f(q)) & \text{if } a = a_{end} \end{cases}$$

Where $f : \wp(\mathcal{I}) \to \Sigma^*$ is a function returning a sequence of the form $rel_i, rel_j, rel_k ...$ for all resources present in its input. Informally, this automaton operates by keeping a list of the resources that have been acquired and not yet released. This list is updated each time an *ac* or a *rel* action is encountered, and such actions are immediately output, as are *use* actions for resources present on the list. *use* actions for resources are simply suppressed. When the monitor encounters the end of execution token action, every resource still present on the list is released.

Proposition 5.3.17. *Let $\mathcal{S} \subseteq \Sigma^\infty$ be a subset of sequences s.t. $\forall \sigma \in \mathcal{S} \cap \Sigma^\omega : \forall i \in \mathcal{I} : \forall j \in \mathbb{N} : \sigma_j = (ac, i) \Rightarrow \exists k > j : \sigma_k = (rel, i)$. The automaton $\mathcal{A}_{ga}^{fair}$ correctively$_{\sqsubseteq}^{\mathcal{S}}$ enforces the general availability property.*

Proof. Every sequence output by the monitor is valid, since the monitor only outputs *use* actions which have been acquired, and a static analysis assures every acquire action is eventually followed by a corresponding release action. Furthermore, the output is necessarily equal to the input on the preorder as the monitor only suppresses a *use* action with the certainty that it is not valid, since it has not yet been acquired. $\mathcal{A}_{ga}^{fair}$ thus correctively$_{\sqsubseteq}$ enforces the general availability property. □

Finally, a static analysis may determine that every possible execution of the target program eventually terminates. If this is the case, an even more corrective enforcement paradigm is available to the monitor, which can acquire resources as needed, and close them at the end of the execution. Since every execution is finite, the number *ac* actions inserted into the sequence will also necessarily be finite.

Let $\mathcal{A}_{ga}^{\Sigma^*} = \langle \Sigma, Q, q_0, \delta_e \rangle$ where Σ is defined as above and

- $Q : \wp(N)$ is the set of resources which have been acquired but not yet released;
- $q_0 = \emptyset$ is the initial state;
- $\delta_{\Sigma^\infty} : Q \times \Sigma \to Q \times \{\Sigma \cup \epsilon\}$ is given as:

$$\delta_e(q,a) = \begin{cases} (q \cup \{i\}, (ac,i); a) & \text{if } a = (use,i) \wedge (ac,i) \notin acts(\tau) \\ (q \cup \{i\}, a) & \text{if } a = (ac,i) \\ (q \backslash i, a) & \text{if } a = (rel,i) \\ (q,a) & \text{if } a = (use,i) \wedge i \in q \\ (\epsilon, f(q)) & \text{if } a = a_{end} \end{cases}$$

Where $f : \wp(\mathcal{I}) \to \Sigma^*$ is a function returning a sequence of the form $rel_i, rel_j, rel_k ...$ for all resources present in its input.

This automaton inserts an *ac* action as needed into the input stream so that every *use* action can be output as soon as it is received. As was the case with $\mathcal{A}_{ga}^{fair}$, the automaton $\mathcal{A}_{ga}^{\Sigma^*}$ also keeps track of every resorcin that has been acquired, so that such resources can be released when the end of execution action is encountered.

Proposition 5.3.18. *The automaton $\mathcal{A}_{ga}^{\Sigma^*}$ correctively$_{\sqsubseteq}^{\Sigma^*}$ enforces the general availability property.*

Proof. Every sequence output by the monitor is valid, since the monitor only outputs *use* actions for resources which have already been acquired, and every achieved resource is closed when the sequence terminates. Since every use action present in the input sequence is also present in the output, we also have that $\forall \sigma \in \Sigma^* : \mathcal{A}_{ga}^{\Sigma^*}(\sigma) \sqsubseteq \sigma$. By theorem 5.2.2, this implies that the property is correctively enforced. □

The three monitors suggested here differ with respect to the set of sequences that can be produced by the target program if the monitor correctively$_{\sqsubseteq}$ enforces the property.

The monitor $\mathcal{A}_{ga}^{\Sigma^\infty}$ can enforce the property in all cases while $\mathcal{A}_{ga}^{\Sigma^*}$ and $\mathcal{A}_{ga}^{fair}$ can only do so if a static analysis successfully rules out some execution paths. It is natural to use this restriction on the monitor's possible use a basis for comparing monitors. We say that a monitor is more versatile than another iff it can enforce the same property when strictly more sequences are present.

Definition 5.3.19. *Let $\mathcal{A}, \mathcal{A}'$ be edit automata enforcing the same property $\hat{\mathcal{P}}$.*
$\mathcal{A} \trianglelefteq_v \mathcal{A}' \Leftrightarrow \forall S \in \wp(\Sigma^\infty) : \forall \sigma \in S : (\hat{\mathcal{P}}(\mathcal{A}(\sigma)) \wedge \sigma \sqsubseteq \mathcal{A}(\sigma)) \Rightarrow (\hat{\mathcal{P}}(\mathcal{A}'(\sigma)) \wedge \sigma \sqsubseteq \mathcal{A}'(\sigma))$.

We write $\mathcal{A} \lhd_v \mathcal{A}' \Leftrightarrow \mathcal{A} \trianglelefteq_v \mathcal{A}' \wedge \neg(\mathcal{A}' \trianglelefteq_v \mathcal{A})$ and $\mathcal{A} \equiv_v \mathcal{A}' \Leftrightarrow \mathcal{A} \trianglelefteq_v \mathcal{A}' \wedge \mathcal{A}' \trianglelefteq_v \mathcal{A}$.

Proposition 5.3.20. $\mathcal{A}_{ga}^{\Sigma^*} \lhd_v \mathcal{A}_{ga}^{fair} \lhd_v \mathcal{A}_{ga}^{\Sigma^\infty}$

Proof. From propositions 5.3.16, 5.3.17 and 5.3.18, we have $\mathcal{A}_{ga}^{\Sigma^*} \trianglelefteq_v \mathcal{A}_{ga}^{fair} \trianglelefteq_v \mathcal{A}_{ga}^{\Sigma^\infty}$. Further, observe that $\mathcal{A}_{ga}^{\Sigma^*}$ cannot enforce the property if the input set contains a sequence from $\Sigma^{fair} \backslash \Sigma^*$, as the acquire actions added by the monitor may be unmatched by release actions. Finally, observe that $\mathcal{A}_{ga}^{fair}$ cannot enforce the property if the input contains sequences from $\Sigma^\infty \backslash \Sigma^{fair}$. It follows that $\mathcal{A}_{ga}^{\Sigma^*} \lhd_v \mathcal{A}_{ga}^{fair} \lhd_v \mathcal{A}_{ga}^{\Sigma^\infty}$. □

The automaton enforcing the general availability property can also naturally be compared using the metrics suggested in the previous sections.

Proposition 5.3.21. $\mathcal{A}_{ga}^{fair} \equiv^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^\infty} \sqsubseteq^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^*}$

Proof. From the transition functions of the automata above, we have that any *use* action present in the input of $\mathcal{A}_{ga}^{\Sigma^*}$ will be present in its output, while only valid *use* actions present in the input of $\mathcal{A}_{ga}^{fair}$ and $\mathcal{A}_{ga}^{\Sigma^\infty}$ will be present in these automata's output. □

Proposition 5.3.22. $\mathcal{A}_{ga}^{\Sigma^\infty} \trianglelefteq_m^{\Sigma^*} \mathcal{A}_{ga}^{fair} \equiv_m^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^*}$

Proof. Follows immediately from the automata constructions given above. □

Proposition 5.3.23. $\mathcal{A}_{ga}^{fair} \equiv_p^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^\infty} \lhd_p^{\Sigma^*} \mathcal{A}_{ga}^{\Sigma^*}$

Proof. Since $\mathcal{A}_{ga}^{\Sigma^*}$ does not suppress invalid *use* actions, it will naturally exhibit more subwords of the input than the other two monitors. Furthermore, any subword of the input present in the output of $\mathcal{A}_{ga}^{fair}$ is also present in the output of $\mathcal{A}_{ga}^{\Sigma^\infty}$. This follows directly from the fact that *use* action present in a fair sequence will either be output by both monitors, or suppressed by both. □

5.4 Conclusion and Future Work

In this chapter, we propose a framework to analyze the security properties enforceable by monitors capable of transforming their input. By imposing constraints on the enforcement mechanism to the effect that some behaviors existing in the input sequence must still be present in the output, we are able to model the desired behavior of real-life monitors in a more realistic and effective way. We also show that real life properties are enforceable in this paradigm, and give four examples of relevant real-life properties. Finally, we propose metrics that can be used to compare alternative enforcements of the same security property.

The framework presented in this paper allows us to transform a program execution to ensure its compliance with a security policy, while also preserving the semantics of the execution. We believe this framework to be sufficiently flexible to be useful in other program rewriting contexts, and even in situations where security is not the main concern, such as controller synthesis or specification refinement. In future work, we hope to adapt our corrective framework to such contexts.

Chapter 6

Conclusion And Future Work

This thesis seeks to bring a new insight to the issue of monitoring and the question of which security policies they can enforce. Previous studies that have addressed this issue have shown that monitors that can draw on a static analysis of their target can be more powerful than similar monitors that lack this ability. However, most implementations of formal monitors did not draw upon this capacity to extend the range of policies that they can enforce.

In this study, we develop a new framework to inline a monitor into a potentially untrusted program, and thus ensure its compliance with the security policy captured by this monitor. By drawing upon static analysis of the target program, we can enforce strictly more properties than previous in-lining approaches. We also state and prove several theorems related to the enforcement of security properties by monitors aided by an abstraction of the program's possible behavior.

Prior research have also shown that monitors that possesses the ability to transform their target execution to be more powerful than others lacking this ability, but only if the notion of enforcement is sufficiently flexible to allow the monitor to replace a valid sequence with another equivalent sequence. However, the models present in the literature did not adequately capture the restrictions that must be imposed on the monitor's behavior, leading to monitors which could, in theory, enforce any property, but often not in a desirable way.

In this thesis we propose two new frameworks to study the enforcement power of monitors capable of transforming their input. The first is based on the insight of using equivalence relations to constrain the behavior of the monitor. The second framework relies on preorders, which allows us to better capture the corrective behavior

of a monitor that transforms its input. We study the enforcement power of a monitor operating within these frameworks and give examples of real life properties that can be enforced if they are used.

Finally, we suggest metrics that allow a user to compare several alternative enforcement monitors for the same property and select the most appropriate one for a given situation.

The research in this study has been the subject of presentations at three conferences: the 14th Nordic Conference on Secure IT Systems [22], the Fifth International Conference Mathematical Methods, Models, and Architectures for Computer Networks Security [42] and the 7th International Workshop on Formal Aspects of Security & Trust (FAST2010) [41] and is accepted for publication in the Journal of Computers & Security [23].

Several avenues for future research remain open. First, the work presented in Chapter 3 can be extended in a number of directions. There are several cases where the procedure fails to produce a suitable instrumented code. This is because the program abstraction is too coarse to allow enforcement by a truncation monitor. A more powerful monitor, capable, for instance, of suppressing a sub-sequence or of inserting some actions, could enforce the property in at least some of these cases.

The corrective enforcement frameworks presented in Chapter 4 and 5 are a very versatile and elegant way to capture transformations performed on a program to ensure its compliance to a security policy. We believe this framework is sufficiently flexible to be useful in other program rewriting contexts, and even in situations where security is not the main concern, such as controller synthesis or specification refinement. In future work, we hope to adapt our corrective framework to such contexts.

Appendix A

Algorithm and Proof of Correction

A.1 Algorithm

In this appendix, we sketch out the algorithm that performs the transformations described in the chapter 3. The most interesting function is $Trim$, which eliminates the inadmissible cycles if any and which aborts with **error** if it is not possible to do so.

Algorithm A.1.1 Synthesizing the Instrumented Program LTS

Input: $\mathcal{R}$ /* *The input Rabin automaton* */
Input: $\mathcal{M}$ /* *The LTS* */
Output: $\mathcal{T}$

1: **let** $\mathcal{R}^P \leftarrow \text{AutomataProduct}(\mathcal{R}, \mathcal{M})$
2: **let** $\mathcal{R}^T \leftarrow \text{AddHalt}(\mathcal{R}^P)$ /* *Adding the halt state* */
3: **let** $\mathcal{T} \leftarrow \text{Trim}(\mathcal{R}^T)$ /* *Removing non admissible cycles from* $\mathcal{R}^T$ *with all incident transitions* */

Algorithm A.1.2 Trim

Input: $\mathcal{R}^T$

Output: $\mathcal{T}$ /* **return** $\mathcal{T}$ *or abort with* **error** */

1: **let** $QT \leftarrow$ BuildScc($\mathcal{R}^{\mathcal{T}}$) /* *Detect the strongly connected components and build the quotient graph* QT */
2: DetectCycles($QT, \mathcal{R}^T$) /* *Detect all the cycles in* $\mathcal{R}^T$ */
3: Annotate($QT, \mathcal{R}^T$) /* *Annotate each scc as A (all cycles admissible), N (all cycles non admissible), B (both), or NC (no cycles)* */
4: SortScc(QT) /* *Sort the* scc *in reverse topological ordering* */
5: **for all** *scc* c in QT **do**
6: CheckForRemove(c) /* *Visit* sccs *according to the reverse topological ordering* */
7: **end for**
8: **let** $\mathcal{T} \leftarrow$ *update*($QT, \mathcal{R}^T$) /* *Build* $\mathcal{T}$ *with states and transitions that have not been removed from* QT */
9: **if** CheckRemovedTransitions($\mathcal{T}, \mathcal{M}$) **then**
10: **return** $\mathcal{T}$
11: **else**
12: **abort** and **return error**
13: **end if**

- CheckRemovedTransitions is a function that scans all the states in $\mathcal{T}$. Let $q = (q_1, q_2) \in \mathcal{T}.Q$ a state in $\mathcal{T}$. Note that $q_1 \in \mathcal{R}.Q$ and $q_2 \in \mathcal{M}.Q$. Let $L(q) = \{a|(\exists q' \in \mathcal{T}.Q| : (q, a, q') \in \mathcal{T}.\delta\})$ and $L'(q) = \{a|(\exists q'_2 \in \mathcal{M}.Q| : (q_2, a, q'_2) \in \mathcal{M}.\delta)\}$.

  If there exists at least one state $q \in \mathcal{T}.Q$ such that $L(q) \subset L'(q)$ and h is not an immediate successor of q then exit and return **false**.

Algorithm A.1.3 CheckForRemove(c)

Input: c

```
1: let Annot ← GetAnnotation(c) /* Get the scc annotation */
2: if Annot = A then
3:    noop /* Leave unchanged */
4: else if Annot = B then
5:    abort and return error
6: else if Annot = NC then
7:    if Succ(c) = ∅ then
8:       Remove(c) /* Remove c with its incident edges. Succ(c) is the set of the
         successors of c in QT */
9:    end if
10: else if Annot = N then
11:    CheckN(c)
12: end if
```

Algorithm A.1.4 CheckN(c)

Input: c

```
1: if Succ(c) = ∅ then
2:    Remove(c)
3: else if AllAnnotA(Succ(c)) = {H} then
4:    TryRemoveTransitions(c)
5:    RemoveRemainingCycles(c)
6: else
7:    abort and return error
8: end if
```

Where

- AllAnnotA(c) returns the set of all successors of c annotated A,
- TryRemoveTransitions(c) removes the transitions connecting states in c that satisfy the following: (q, a, q') is removable if q has h as an immediate successor and if q' is in c,
- RemoveRemainingCycles(c) removes the remaining cycles in c with all their incident edges. This procedure also removes the states that are no longer accessible with their incident edges.

A.2 Proof of Correction

In what follows, we give a sketch of a proof showing that whenever the algorithm succeed in constructing an LTS $\mathcal{T}$ requirements 3.3.1 and 3.3.2 hold.

Proof of requirement 3.3.1

There are two cases to consider, namely the case where $\hat{\mathcal{P}}(\sigma)$ and the case where $\neg\hat{\mathcal{P}}(\sigma)$.

- Case 1, $\hat{\mathcal{P}}(\sigma)$:

  We begin by showing that $\sigma = \tau$. By contradiction, if $\sigma \neq \tau$ then there exists a transition t in $\mathcal{R}^{\mathcal{T}}$, used by σ, but absent in τ. Yet, such a transition could not have been eliminated during the transformation phase of R^T. We will show that this is the case both for transitions that occur inside a *scc* or connecting two different *scc*s. Note that $\hat{\mathcal{P}}(\sigma)$ means that σ reaches an admissible cycle starting from the initial state.

  - $t = (q1, a, q2)$ with $q1, q2$ in the same *scc* c. The *scc*s are treated by our algorithm based on whether the cycles they contain are admissible or not. We examine each possibility in turn.
    - c contains only admissible cycles: In such a case, the *scc* is preserved in $\mathcal{T}$ by algorithm A.1.3 lines 1-2. Furthermore, since a *scc* can only be removed if it has no admissible successors (A.1.3 lines 7 and 8 and A.1.4 lines 1 and 2), c will remain accessible in $\mathcal{T}$.
    - c contains both admissible and inadmissible cycles. In such a case, we cannot construct $\mathcal{T}$ and are forced to return **error**.
    - c contains only inadmissible cycles. The *scc* is removed only if it has no successor (lines 1 and 2 of A.1.4). In this case, the execution reaching this *scc* cannot be valid. If after crossing c,the execution can reach a *scc* with admissible cycles other than H we have to abort with **error**, (line 7 in A.1.4). Otherwise, the transition t may be removed only if doing so does not remove any initial paths to H. We thus remove only the transitions that go from immediate predecessors of H to another state in the *scc* (cf the explanation of the function `TryRemove Transitions`). To sum up, a transition in c is removed only if it cannot allow the execution to reach an admissible cycle.

  - $*$ c contains no cycles. It can only be removed if it has no successors, (lines 6,7 and 8 in A.1.3). Note that if c has a successor c' with no cycle, we can show that from c' we can reach an admissible cycle, otherwise it would have been removed during some previous iteration.

- $t = (q1, a, q2)$ with $q_1 \in c_1$ and $q_2 \in c_2$, $c_1 \neq c_2$. Such a transition is only removed if its destination is a state in a *scc* that is removed.

  As discussed above, such an *scc* can never be a part of a valid path.

$\hat{\mathcal{P}}(\tau)$ follows immediately from $\sigma = \tau$ and $\hat{\mathcal{P}}(\sigma)$.

- Case 2 : $\neg\hat{\mathcal{P}}(\sigma)$.

  Since our method consists exclusively in removing, rather than adding states and transitions, it is obvious that the execution τ emitted by $\mathcal{T}$ when running σ is either equal to σ , or is a prefix of it. To show that such a sequence τ would always respect the property, we must consider two cases, namely τ is infinite and τ is finite.

  - τ is infinite. In such a case, the proof that $\hat{\mathcal{P}}(\tau)$ proceeds by contradiction. Let τ be an invalid infinite sequence, τ must enter an inadmissible cycle. Yet, all *scc* containing such cycles were removed from $\mathcal{R}^{\mathcal{T}}$ by algorithm A.1.4 line 2 or line 5. And if we were unable to remove them an error message would have been produced without generating $\mathcal{T}$.
  - τ is finite. In this case, σ has been halted in h producing τ or it reached an *end* state, after executing an a_{end} transition. We know that an execution reaching the *end* state satisfy the property, since we have kept in $\mathcal{R}^{T}$ only executions satisfying $\hat{\mathcal{P}}$. We have been careful to not remove transitions that could belong to an execution in $\mathcal{L}_{\mathcal{M}}$ without being sure that the origin of the removed transition is a state that is a predecessor of h, σ could not reach but an *end* state or h, thus we can be sure that we have $\hat{\mathcal{P}}(\tau)$.

Proof of requirement 3.3.2 The first half on the conjunction is immediate from the construction process of $\mathcal{T}$. Since we have have not added any new states or transitions, safe those needed to abort the execution when needed, it follows than any execution that remains in $\mathcal{T}$ was already present in $\mathcal{M}$. Once again we have to examine the cases of τ being finite or infinite separately.

- τ is infinite. In such a case, τ can only be invalid if it enters an inadmissible cycle. As discussed above, all such cycles were removed from $\mathcal{T}$ by algorithm A.1.4 line 2 or line 5.

- τ is finite. Likewise, we have already ascertained that any such sequence must be valid, since it necessarily ends in a safe *end* or halt state.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

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

[3] B. Alpern and F.B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:17–126, 1987.

[4] A. Bauer. Monitorability of ω-regular languages. Computing Research Repository (CoRR) abs/1006.3638, Association for Computing Machinery (ACM), June 2010.

[5] A. Bauer and J. Jürjens. Security protocols, properties, and their monitoring. In *Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems (SESS)*.

[6] A. Bauer and J. Jürjens. Runtime verification of cryptographic protocols. *Computers & Security*, 29(3):315–330, 2010.

[7] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. In *Proceedings of the 26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 4337 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, December 2006. Springer-Verlag.

[8] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly, but how ugly is ugly? In *Proceedings of the 7th International Workshop on Runtime Verification (RV)*, volume 4839 of *Lecture Notes in Computer Science*, pages 126–138, Berlin, Heidelberg, November 2007. Springer-Verlag.

[9] A. Bauer, M. Leucker, and C. Schallhart. The good, the bad, and the ugly—but how ugly is ugly? Technical Report TUM-I0803, Institut für Informatik, Technische Universität München, February 2008.

[10] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *Accepted for publication in ACM Transactions on Software Engineering and Methodology*, 2010.

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

[12] D. Beauquier, J. Cohen, and R. Lanotte. Security policies enforcement using finite edit automata. *Electr. Notes Theor. Comput. Sci.*, 229(3):19–35, 2009.

[13] D. Beauquier and J.-E. Pin. Languages and scanners. *Theoretical Computer Science*, 84(1):3–21, 1991.

[14] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, 2007.

[15] K. J. Biba. Integrity considerations for secure computer systems. Technical report, MITRE Corporation, 1977.

[16] N. Bielova and F. Massacci. Do you really mean what you actually enforced? edit automata revised. Technical report, Ingegneria e Scienza dell'Informazione, University of Trento, 2008.

[17] N. Bielova and F. Massacci. Do you really mean what you actually enforced? In *Formal Aspects in Security and Trust: 5th International Workshop, FAST 2008 Malaga, Spain, October 9-10, 2008 Revised Selected Papers*, pages 287–301, Berlin, Heidelberg, 2009. Springer-Verlag.

[18] N. Bielova, F. Massacci, and A. Micheletti. Towards practical enforcement theories. In *Identity and Privacy in the Internet Age, 14th Nordic Conference on Secure IT Systems, NordSec 2009, Oslo, Norway, 14-16 October 2009. Proceedings*, pages 239–254, 2009.

[19] W. E. Boebert and R. Y. Kain. A practical alternative to hierarchical integrity policies. In *Proceedings of the 8th National Computer Security Conference*, 1985.

[20] D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[21] H. Chabot. Sécurisation de code basée sur la combinaison de la vérification statique et dynamique. Master's thesis, Laval University, 2009.

[22] 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, NordSec 2009*, volume 5838 of *LNCS*, pages 287–301. Springer, oct 2009.

[23] H. Chabot, R. Khoury, and N. Tawbi. Extending the enforcement power of truncation monitors using static analysis. *Computers & Security*, Forthcoming.

[24] E. Chang, Z. Manna, and A. Pnueli. The safety-progress classification. Technical report.

[25] A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proceedings of the 23rd IEEE Computer Security Foundations Symposium, CSF 2010, Edinburgh, United Kingdom, July 17-19, 2010*, pages 200–214.

[26] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Conference record of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2000.

[27] M. d'Amorim and G. Roşu. Efficient monitoring of omega-languages. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2005.

[28] G. Edjlali, A. Acharya, and V. Chaudhary. History-based access control for mobile code. In *ACM Conference on Computer and Communications Security*, pages 38–48, 1998.

[29] S. Eilenberg. *Automata, Languages and Machines*, volume A. Academic Press, New York, 1974.

[30] U. Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004.

[31] U. Erlingsson and F.B. Schneider. Sasi enforcement of security policies: A retrospective. In *WNSP: New Security Paradigms Workshop*. ACM Press, 2000.

[32] Y. Falcone, J.-C. Fernandez, and L. Mounier. Synthesizing enforcement monitors wrt. the safety-progress classification of properties. In *Information Systems Security, 4th International Conference, ICISS 2008, Hyderabad, India, December 16-20, 2008. Proceedings*, volume 5352 of *Lecture Notes in Computer Science*, pages 41–55, 2008.

[33] Y. Falcone, J.-C. Fernandez, and L. Mounier. Synthesizing enforcement monitors wrt. the safety-progress classification of properties. Technical Report TR-2008-7, Verimag Research Report, 2008.

[34] Y. Falcone, J.-C. Fernandez, and L. Mounier. Enforcement monitoring wrt. the safety-progress classification of properties. In *SAC '09: Proceedings of the 2009*

ACM symposium on Applied Computing, pages 593–600, New York, NY, USA, 2009. ACM.

[35] Y. Falcone, J.-C. Fernandez, and L. Mounier. Runtime verification of safety-progress properties. In *Runtime Verification: 9th International Workshop (RV 2009), Grenoble, France, Selected Papers*, pages 40–59, Berlin, Heidelberg, 2009. Springer-Verlag.

[36] P. Fong. Access control by tracking shallow execution history. In *In Proceedings of the 2004 IEEE Symposium on Security and Privacy, Oakland, California, USA, May 2004.*, 2004.

[37] G. Le Guernic. Automaton-based Confidentiality Monitoring of Concurrent Programs. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSFS20)*, pages 218–232. IEEE Computer Society, July 6–8 2007.

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

[39] P. Jun, C. Xingyuan, W. Bei, D. Xiangdong, and W. Yongliang. Policy monitoring and a finite state automata model. In *CSSE '08: Proceedings of the 2008 International Conference on Computer Science and Software Engineering*, pages 646–649, Washington, DC, USA, 2008. IEEE Computer Society.

[40] G. Morrisett K. W. Hamlen and F.B. Schneider. Computability classes for enforcement mechanisms. Technical Report TR2003-1908, Cornell University, 2003.

[41] R. Khoury and N. Tawbi. Corrective enforcement of security policies. In *the 7th International Workshop on Formal Aspects of Security & Trust (FAST2010)*, 2010.

[42] R. Khoury and N. Tawbi. Using equivalence relations for corrective enforcement of security policies. In *the Fifth International Conference Mathematical Methods, Models, and Architectures for Computer Networks Security*, 2010.

[43] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Computational analysis of run-time monitoring - fundamentals of java-mac. *Electr. Notes Theor. Comput. Sci.*, 70(4), 2002.

[44] S. M. Kim and R. McNaughton. Computing the order of a locally testable automaton. *SIAM J. Comput.*, 23(6):1193–1215, 1994.

[45] S. M. Kim, R. McNaughton, and R. McCloskey. A polynomial time algorithm for the local testability problem of deterministic finite automata. *IEEE Trans. Comput.*, 40(10):1087–1093, 1991.

[46] O. Kupferman, Y. Lustig, and M.Y. Vardi. On locally checkable properties. In *Proccedings of the 13th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[47] O. Kupferman and M. Y. Vardi. Model checking of safety properties. *ormal Methods in System Design*, 19(3):291–314, 2001.

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

[49] M. Langar and M. Mejri. Formal and efficient enforcement of security policies. In *Proceedings of The 2005 International Conference on Foundations of Computer Science, (FCS 2005)*, pages 143–149, 2005.

[50] M. Langar and M. Mejri. Optimizing enforcement of security policies. In *proceedings of the Foundations of Computer Security Workshop (FCS'05) affiliated with LICS 2005 (Logics in Computer Science)*, June-July 2005.

[51] M. Langar, M. Mejri, and K. Adi. Formal monitor for concurrent programs. In *Workshop on Practice and Theory of IT Security*, 2006.

[52] M. Langar, M. Mejri, and K. Adi. A formal approach for security policy enforcement in concurrent programs. In *Proceedings of the 2007 International Conference on Security & Management*, pages 165–171, 2007.

[53] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 2004.

[54] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *10th European Symposium on Research in Computer Security (ESORICS)*, Milan, September 2005.

[55] 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.

[56] J. Ligatti and S. Reddy. A theory of runtime enforcement, with results. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, September 2010.

[57] T. Y. Lin. Chinese wall security policy — an aggressive model. In *Proceedings of the Fifth Aerospace Computer Security Application Conference*, 1989.

[58] A. Magnaghi and H. Tanaka. An efficient algorithm for order evaluation of strict locally testable languages. In *International Conference on Combinatorics, Graph Theroy and Computing (Florida, USA)*.

[59] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[60] T. Mechri, M. Langar, M. Mejri, H. Fujita, and Y. Funyu. Automatic enforcement of security in computer networks. In *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Sixth SoMeT 2007*, pages 200–222, 2007.

[61] D. E. Muller. Infinite sequences and finite machines. *Switching Circuit Theory and Logical Design*, 0:3–16, 1963.

[62] H. Ould-Slimane and M. Mejri. Enforcing security policies by rewriting programs using automata. In *Proceedings of the Workshop on Practice and Theory of IT Security (PTITS)*, pages 195–207, 2006.

[63] H. Ould-Slimane, M. Mejri, and K. Adi. Enforcing security policies on programs. In *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Fifth SoMeT 2006, October 25-27, 2006, Quebec, Canada*, pages 195–207, 2006.

[64] H. Ould-Slimane, M. Mejri, and K. Adi. Using edit automata for rewriting-based security enforcement. In *Data and Applications Security XXIII, 23rd Annual IFIP WG 11.3 Working Conference, Montreal, Canada, July 12-15, 2009. Proceedings*, pages 175–190, 2009.

[65] D. Perrin and J. E. Pin. *Infinite Words*. Pure and Applied Mathematics Vol 141. Elsevier, 2004.

[66] A. Pnueli. The temporal logic of programs. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:46–57, 1977.

[67] P. J. Ramadge and W. M Wonham. The control of discrete event systems. *IEEE Proceedings: Special issue on Discrete Event Systems*, 77(1):81–97, January 1989.

[68] J.-F. Raskin. *Logics, Automata and Classical Theories for Deciding Real-Time*. PhD thesis, Université de Namur, 1999.

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

[70] K. Sen, , A. Vardhan, G. Agha, and G. Roşu. Efficient decentralized monitoring of safety in distributed systems. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 418–427, Washington, DC, USA, 2004. IEEE Computer Society.

[71] C. Small. A tool for constructing safe extensible C++ systems. In *In Proceedings of the Third Usenix Conference on Object-Oriented Technologies*, pages 175–184, 1997.

[72] A. E. K. Sobel and J. Alves-Foss. A trace-based model of the chinese wall security policy. In *In Proceedings of the 22nd National Information Systems Security Conference*, 1999.

[73] A. Syropoulos. Mathematics of multisets. In *Proceedings of the Workshop on Multiset Processing*, pages 347–358. Springer-Verlag, 2001.

[74] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement for limited-memory systems. In *proceedings of the PST06 Conference (Privacy, Security, Trust)*, October 2006.

[75] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. In *proceedings of FCS-ARSPA-06 (Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis) associated with FLOC 2006 (Federated Logic Conference)*, August 2006.

[76] C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitations constraints. *Information and Computation*, 206(1):158–184, 2008.

[77] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[78] A. N. Trahtman. An algorithm to verify local threshold testability of deterministic finite automata. In *Revised Papers from the 4th International Workshop on Automata Implementation*, WIA '99, pages 164–173, London, UK, 2001. Springer-Verlag.

[79] M. Viswanathan. *Foundations for the run-time analysis of software systems*. PhD thesis, University of Pennsylvania, 2000.

[80] F. Yan and P. W. L. Fong. Efficient irm enforcement of history-based access control policies. In *Proceedings of the 2009 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2009, Sydney, Australia*, pages 35–46, 2009.

[81] W.D. Young, P.A. Telega, and W.E. Boebert. A verified labeler for the secure ada target. In *Proceedings of the 9th National Computer Security Conference*, September 1986.

[82] G. Zhu and A. Tyagi. Stream automata as run-time monitors for open system security policies, 2008.