Probabilistic-Cost Enforcement of Security Policies in Distributed Systems

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical & Computer Engineering

Ioannis (Yannis) Mallios

B.Sc., Informatics, Athens University of Economics & Business
M.Sc.. Information Security, Carnegie Mellon University
M.Sc., Electrical & Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

December, 2016

Acknowledgments

First of all, I thank my advisor Lujo Bauer. Lujo has been a great advisor in all aspects of graduate school, always happy to listen to, answer questions, and discuss research ideas. Without his help, guidance, and patience, this dissertation would not be possible.

In addition, I would like to thank the members of my committee: Anupam Datta, Jay Ligatti, and Dilsun Kaynar. I had the pleasure to interact and work with them since the beginning of my PhD. Our collaboration on parts of this research, their insights, guidance, and their feedback on improving this thesis have been invaluable.

Next, I thank Fabio Martinelli and Charles Morisset for being great collaborators and co-authoring papers that are part of the material included in this thesis.

I would also like to thank the members of Lujo's research group. They have been great colleagues, and they have provided constructive feedback in many drafts of papers and presentations.

I thank my close friends, whom I consider family, Thanasis Avgerinos, Nektarios and Jill Leontiadis, Elli Fragkaki, Eleana Petropoulou, Vicky Theodoreli, and Ryan Shaw. They have been wonderful friends and they have made graduate life much easier.

I would like to thank John Poulakos and his family, who have been extremely supportive since the first day I came to Pittsburgh.

I thank my family: my father Tasos, my mother Maria, and my brother Sotiris. Without them, their unconditional love, support, understanding, and patience, graduate studies would not have been possible.

Last, but definitely not least, I want to thank Ashley. She has been extremely kind, understanding, supportive, and very patient, especially during the last few hectic and stressfull months. I am extremely grateful to have her in my life.

This work was partially supported by NSF grants CNS-0716343, CNS-0742736,

CCF-0917047, and CNS-0917047; by Carnegie Mellon CyLab under Army Research Office grant DAAD19-02-1-0389, by EU FP7 projects NESSoS and SESAMO, by H2020 EU NeCS, by MIUR PRIN Security Horizons, and by the Army Research Laboratory under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government.

Abstract

Computer and network security has become of paramount importance in our everyday lives. Cyber attacks can lead to a wide range of undesirable situations ranging from breaches of personal information and confidential data to loss of human lives. One way to protect computer and network systems is through the use of technical (i.e., software and hardware) security mechanisms, such as firewalls and Intrusion Detection Systems (IDSs).

Previous work has introduced formal frameworks that can be used to model such technical security mechanisms. Such formal frameworks help us: (1) understand the fundamental limitations of security mechanisms, (2) verify the correctness of the design of security mechanisms, and (3) efficiently design secure systems.

While these frameworks provided an important first step for the modeling of security mechanisms and the analysis of their enforcement capabilities, they were able to model only individual security mechanisms and they could not be used to compare the cost of different monitoring designs.

In this thesis we present formal frameworks for modeling and reasoning about a larger class of security mechanisms and enforcement scenarios than previous research. We demonstrate how our frameworks can be used to model different types and architectures of security mechanisms, both for centralized and distributed systems (e.g., IDSs and distributed IDSs). We use our frameworks to identify and prove new lower and upper bounds of the enforceable security policies by security mechanisms.These results extend the list of bounds of enforceable security policies identified by previous research and broaden our understanding of fundamental limitations of the enforcement capabilities of security mechanisms. Finally, we demonstrate how to compare the expected cost of different designs of security mechanisms.

Contents

List of Figures

Chapter 1

Introduction

Computer and network security has become of paramount importance in our everyday lives. Almost on daily basis, we witness a proliferation of attacks that result in breaches of personal and confidential information, including email addresses and passwords [2, 3, 4], tax information [5], bank and credit card accounts [6, 7], Social Security numbers [8, 9, 10], health information [9, 10, 11], and emails [12, 13, 14]. More recently, documented attacks on vehicles [15, 16, 17], SCADA systems [18, 19, 20, 21, 22, 23], and governments' servers and networks [24, 25] made it clear that computer and network attacks could result in life-threatening situations. Thus, security scientists and engineers have been trying to develop and deploy mechanisms that will protect the systems and networks that we rely on.

Whether we are trying to protect a single file on a device or the traffic on an organization's network, we need to define our security goals. A *security policy* describes what are the behaviors of the system (or the network) that are, and are not, allowed [26]. *Security mechanisms* are methods, tools, or procedures used for enforcing security policies [26]. One of the most prevalent types of mechanisms used for enforcing security policies are *technical* security mechanisms (i.e., software and hardware), as opposed to, for example, methods such as showing an identification

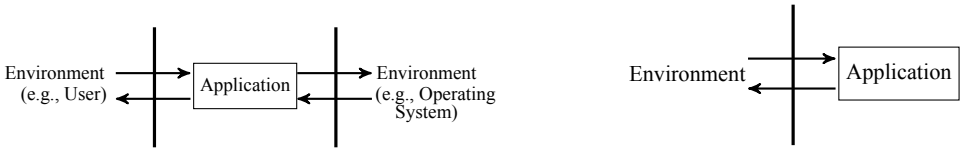
card for credential creation or using physical keys to access certain areas.

Previous work has introduced formal frameworks that can be used for modeling (technical) security mechanisms [27, 28, 29, 30]. Such formal frameworks are of great significance and benefit for the following reasons:

1. They can be used to analyze the intrinsic limits of different types of security mechanisms by characterizing the class of security policies that these mechanisms can enforce. Such a characterization can lead to a taxonomy of security policies based on mechanisms' formal semantics [27], similar to taxonomies in computability and complexity theory [31, 32].
2. They can guide the efforts and choices of systems' designers in practice: for example, a security engineer can avoid spending time to design monitors that implement security policies that are *inherently* not enforceable; instead, she can focus on either reevaluating the accepted risk of a weaker security policy, or the potential loss in the usability of the underlying monitored system.
3. They can be used to (more efficiently) verify modeled security mechanisms which can lead to systems with high assurance. For instance, when building systems, modifying, experimenting on, and verifying a model of the system (that ignores some of the details that are not important to the goals of the analysis) is much easier than experimenting on the implementation itself [33].

Most of the formal frameworks that were introduced in the past aim to characterize the security policies enforceable by a specific type of security mechanisms, namely *reference monitors*, or *run-time monitors* [27, 30, 34]. Reference monitors are programs that are interposed between an application and the (executing) environment, actively observing (i.e., mediating) the application's[1] execution, and taking remedial action when an action of the target is about to violate the

[1]We will be using the terms *application*, *target application*, *target*, *node*, and *host* to refer to the software (and potentially hardware) that the security mechanism is monitoring.



(a) Target application

(b) Target application with single environment

Figure 1.1: Target applications

predefined security policy. Reference monitors were originally designed to enforce access control policies on the system's objects by mediating all requests (i.e., references) users make [35]. Such a situation is depicted in Fig. 1.1a which for simplicity can also be modeled as in Fig. 1.1b[2]. Reference monitors are widely implemented and used in systems today. Two typical examples are depicted in Fig. 2.5: wrapping monitors (Fig. 1.2a) are models of practical implementations such as software wrappers [37], inlined reference monitors [38], and virtual machines, whereas traditional monitors (Fig. 1.2b) are models of implementations such as access control mechanisms [39], protocol scrubbers [40], and system call interposition mechanisms [41].

Since the goal of such formal frameworks was to characterize the security policies that run-time monitors can enforce, the frameworks were well-suited to model scenarios where: *individual* monitors that can *completely mediate* all security relevant actions (i.e., actions that the target application wants to execute and could potent violate the security policy) enforce security policies by *preventing* attacks, i.e., once an attempt to violate the security policy is detected the monitor either terminates the target application (modeled by security automata [27]) or suppresses the invalid behavior until the target itself decides to correct it (modeled by edit automata [30]).

Policy enforcement in practice relies on the use of multiple security mechanisms (which might

[2]The original formal frameworks of run-time monitors did not care about *whom* the application was interacting with; only the fact that it was interacting with some environment [27, 30, 34, 36]. Typically the environment was the underlying executing system, and the frameworks did not focus on the interaction between the user and the application

(a) Wrapping monitor (b) Traditional monitor

Figure 1.2: Reference monitors

cooperate) whether the policy is enforced on a single computer or a network spanning various remote physical locations. More concretely, to enforce a security policy one might typically employ firewalls [42, 43], Network Intrusion Detection Systems (NIDS) (e.g., Suricata [44]), Host-based Intrusion Detection Systems (HIDS) (e.g., tripwire [45]), Intrusion Prevention Systems (IPS) (e.g., snort [46]), audit logs, netflow data analysis [47], spam filters (e.g., SpamAssassin [48]), etc. Next, we discuss two main reasons that security analysts rely on multiple security mechanisms to enforce security policies.

The first reason that a security analyst relies on multiple security mechanisms stems from the fact that security mechanisms have different capabilities. Thus, the analyst might need to correlate information from different security mechanisms in order to enforce a security policy. For example, consider a scenario where we have two different security mechanisms: a network-based IDS, and web server access logs. The Apache chunked-encoding exploit is an attack where a successful exploitation does not leave an entry in the server logs [49]. One way to detect the attack is to analyze network packets looking for evidence of binary data that are followed by a matching entry at the server logs [49]. If none is found then an alert is raised. Such an attack could go undetected if either an IDS was not used, or server logs were not maintained. One useful way that security mechanisms can be classified is along the the following three axes [50]:

1. Vantage: the placement of monitors[3] within a network or a distributed system. Due to the

[3]For the rest of this thesis we will use the terms *security mechanism* and *monitor* interchangeably. Whether we are referring to arbitrary security mechanisms or to the specific class of reference (or run-time) monitors will be clear

(a) Input-mediating monitor (b) Output-mediating monitor

Figure 1.3: Partially-mediating monitors

geographical span of systems, and physical limitations of the underlying network architecture, typically security mechanisms are located at specific physical locations and may not have a complete view of the system. This means that they cannot offer complete coverage of the network and they can only *partially mediate* security relevant actions. For example, if the target (host) is using two different interfaces for its input and output traffic, then traffic could be routed differently for each interface. This means that a security mechanism monitoring only part of the network might not observe part of the host's interaction with the environment, as depicted in Fig. 1.3.

2. Domain: the information the security mechanism can observe and provide (e.g., network traffic, application logs).

3. Action: how security mechanisms react to traffic they intercept. Mechanisms can: (1) *report*, i.e., simply record all traffic they receive (e.g., NetFlow collectors, tcpdump, and server logs); (2) *generate events*, i.e., summarize the traffic they receive and produce events (e.g., an IDS producing an alert when it matches one of its signatures with observed traffic); and (3) *control*, i.e., mechanisms that can modify or block traffic in addition to sending events (e.g., IPSs, firewalls, and anti-spam systems). Note that whereas control monitors *must* be interposed between entities so that they can control traffic, reporting and event monitors (e.g., IDSs) may not be: they might receive information through a broadcast chan-

from the context.

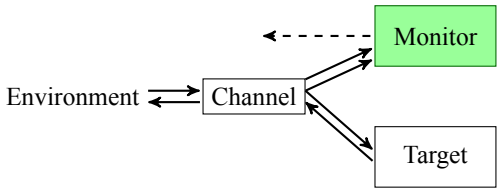


Figure 1.4: Monitor that passively mediates traffic through a broadcast channel

nel (e.g., a switch's spanning port or ethernet), as depicted in Fig. 1.4.

The second reason that multiple security mechanisms are used in practice to enforce security policies is that correlation of information from these mechanisms can help in identifying attacks with fewer false positives and negatives by identifying the temporal behavior of attackers. If an attacker chooses to randomly try exploits on a network (which some worms actually do), then the detection and defense can be easier (e.g., intercepting a Microsoft IIS exploit for an Apache web server). Thus, typically, attackers rely on several steps in order to perform their attacks and remain unnoticed, such as reconnaissance, intrusion, privilege escalation, and goal steps [51]. For example, an attacker might take the following steps to identify if a system is vulnerable to a Winnuke attack (a DoS attack against the DNS system) [52]: use *nslookup* to locate the DNS server, *ping* the server to check if the service is active, and, finally, *scan port 139* (NetBios) to learn if the Windows system is active. To detect this attack the analyst needs to correlate information from three different types of logs: DNS, NetFlow, and syslog [52]. It is typical for such correlation and analysis to happen at a central location. For example, if we are trying to monitor a distributed system as the one depicted in Fig. 1.5, then a central monitor can mediate the interaction between the distributed system and the environment (Fig. 1.6a), or even mediate all communications, including the ones of the system itself (Fig. 1.6b).

However there are two main concerns that have been raised in the past regarding monitors in central locations: a centralized monitor becomes (1) a single point-of-failure, and (2) overwhelmed with the amount of data that it needs to collect (communication load to transmit data)

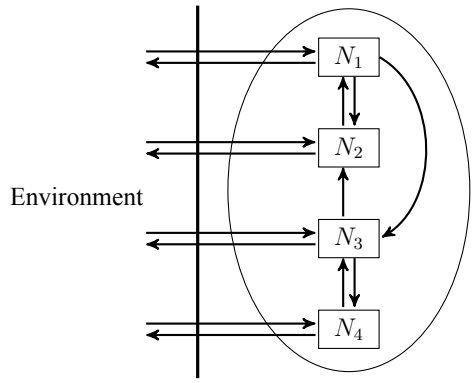
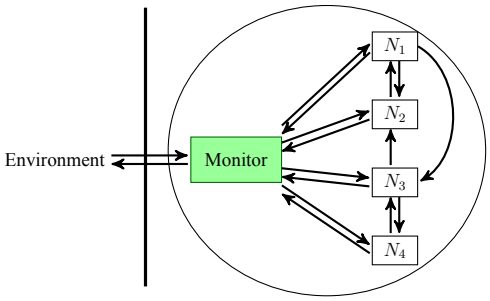
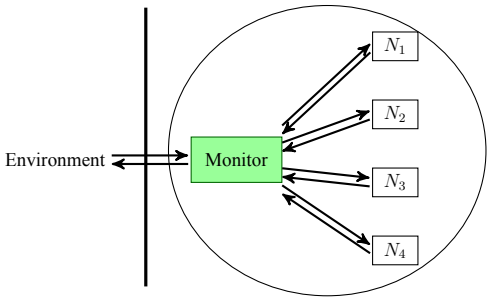


Figure 1.5: A distributed system



(a) Monitored distributed system where a central monitor actively mediates the interaction between the environment and the nodes of the system

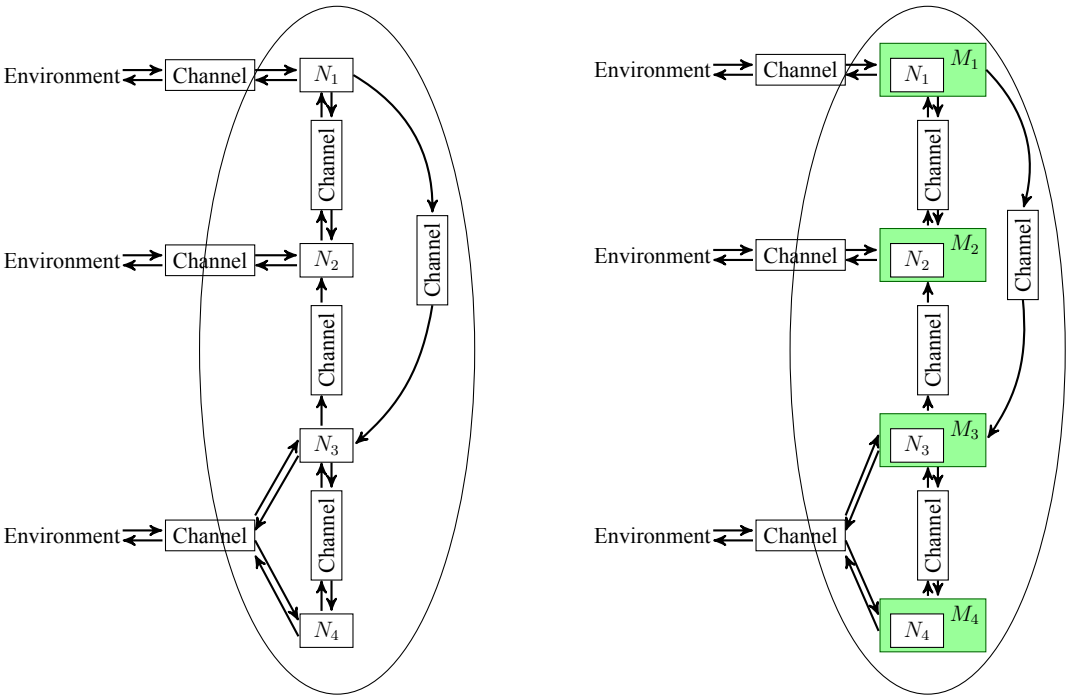


(b) Monitored distributed system where a central monitor actively mediates all interactions, including the ones among the nodes of the system

Figure 1.6: Centrally monitored distributed system

and analyze (computational load). Thus, solutions have been introduced in order to achieve better fault-tolerance, communication efficiency, and computational efficiency. Two main approaches that have been suggested are *hierarchical enforcement* and *decentralized enforcement*.

In hierarchical approaches security mechanisms are organized in a hierarchical fashion. At the lowest level of the hierarchy each mechanism is responsible for a subset of the nodes on the network. At this level mechanisms collect data, perform some analysis, and forward the results to monitors at higher levels which further analyze the data. With such approaches the goal is to minimize the communication and computation by having some of the monitors do



(a) Distributed system with channels

(b) Monitored distributed system with decentralized monitors

Figure 1.7: Decentralized monitoring

some work locally. This is a very attractive approach. If networks span over wide physical area (e.g., a subnetwork in Silicon Valley and another one in Pittsburgh), then monitors operating in a hierarchical fashion will enforce policies locally at each location, and will only exchange information if attacks are relevant to both locations.

Decentralized approaches originated from security mechanisms that required co-operation and coordination (e.g., distributed IDS and distributed firewalls). Security mechanisms are distributed over the network, operating independently of each other, and collecting and analyzing data locally. Once a node realizes that this data might be relevant to some other agents, it forwards the appropriate information to them so that they can (collaboratively) identify the attack. For instance, in Fig. 1.7a we see a distributed system with the channels that the system's node use to communicate with the environment and with each other. In Fig. 1.7b we see local monitors

attached to the nodes, using the underlying network infrastructure to communicate, and cooperate to enforce the global security policy.

Research goal: Design a formal framework for modeling and analyzing security policies that is expressive enough to model: (1) a wide variety of common types of security mechanisms, and (2) distributed systems, networks, and security mechanisms that can communicate and interact. In addition, the framework must provide the necessary tools that can be used to prove the correctness of monitoring designs and algorithms that manipulate security objects (e.g., algorithms that decompose a central security mechanism to security mechanisms that collaboratively enforce the same security policy, as the central mechanism, over a distributed system).

The more expressive a formal framework is the more valuable it can be, as security engineers and analysts can reason about a wider spectrum of practical enforcement scenarios. A formal framework that allows us to model centralized, hierarchical, and decentralized security mechanisms has a greater applicability than one that only allows us to model, and reason about, individual monitors. However, the fact that we can now model and reason about a wider range of monitoring designs for a single underlying target distributed system, naturally leads to the same questions that security engineers face in real world: *which design is better and should be implemented?*

In practice, the benefits of computer security are weighed against their total cost. Choosing the design of a security mechanism (e.g., a centralized monitor or a decentralized architecture) typically reduces to evaluating the cost of each candidate solution. To evaluate and compare each monitoring design requires to identify the potential threats against that system, the likelihood that they will occur, the loss of value that that they will incur, and finally the cost of the mechanism itself. For instance, assume that we want to enforce a security policy over an asynchronous

distributed system. A decentralized approach would require the monitors to use some expensive means of synchronization[4]. A centralized approach might be a more suitable solution, if the probability of failure of the monitor is very low.

Research goal: Design a formal framework that is expressive enough to model different designs of security mechanisms, providing the means to model and reason about their expected cost, and allowing their comparison in an unambiguous and correct manner.

Thesis statement. In this thesis we present formal frameworks for modeling and reasoning about a larger class of security mechanisms and enforcement scenarios than previous research. We demonstrate how our frameworks can be used to model different types and architectures of security mechanisms, both for centralized and distributed systems (e.g., IDSs and distributed IDSs). We provide lower bounds of enforceable policies by monitors that can be modeled in our frameworks. We characterize the constraints under which security policies are enforceable in distributed systems. Finally, we demonstrate how to compare the expected cost of different monitoring designs.

[4]Expensive in terms of communication complexity, as we discuss in more detail in Section 3.4

1.1 Dissertation Structure and Contributions

The transition from formal frameworks that can model individual (centralized) run-time monitors to formal frameworks that can model various types of security mechanisms, distributed systems, and costs of monitoring designs, introduces a number of complexities. For instance, it requires dealing with issues that are related to modeling enforcement scenarios in more detail (e.g., lifting key definitions of security policies and enforcement in order to deal with partial-mediation and asynchronous communication, e.g., passively mediating IDSs). It also requires dealing with issues that stem from the fundamental limitations of distributed systems (e.g., concurrent executions and partial ordering, lack of a global clock, and non-atomic events).

To deal with these complexities while introducing the key concepts, ideas, and results of our frameworks, we have taken an incremental approach. In particular, we start by introducing a formal framework that allows modeling one monitor and one target application that interact asynchronously (Chapter 2). This allows us to introduce the concepts and key definitions of enforcement without the (notational and conceptual) complexities of dealing with multiple targets, monitors, and communication channels. Then, we extend this basic framework to a more fine-grained framework that allows modeling distributed systems and monitors (Chapter 3). Finally, we present a more detailed framework than the one of Chapter 2 that allows modeling and reasoning about the expected cost of monitoring designs (Chapter 4).

More specifically, this thesis extends previous work in three principal ways:

1. We introduce a (basic) formal framework based on Input/Output automata that can be used to model target applications, various types of monitors (e.g., partially mediating ones), and the environment that the monitored targets operate (Chapter 2). In particular:
    - We show how different monitors (e.g., partially mediating monitors) can be modeled

in our framework.

- We extend previous definitions of security policies to support more fine-grained reasoning of policy enforcement (e.g., enforcement that requires monitors to enforce the security policy by modifying traffic).
- We identify a set of lower bounds of policies that are enforceable by monitors that can be modeled in our framework.
- We demonstrate how to use our framework to derive meta-theoretical results, such as the comparison of enforcement capabilities of monitors with different monitoring interfaces (i.e., one monitor can mediate more actions than the other), and the characterization of the class of security policies that monitors can enforce regardless of their monitoring interface. Such results are of practical importance, because they can help security engineers make design and implementation choices that lead to more efficient enforcement solutions.

2. We extend the previous basic framework, by introducing a framework that allows modeling distributed systems with arbitrary architectures, and distributed monitors (Chapter 3). In this chapter, we make the following contributions:
   - We characterize the security policies that are enforceable in asynchronous and synchronous distributed systems. This characterization is based on an analysis of which centralized monitors can be *simulated* by distributed monitors over a target (distributed) system.
   - We provide a *blueprint* for decomposing centralized monitors to monitors that are distributed over a network and enforce the same policies as the original centralized monitors. This blueprint allows us to explore in an incremental way the fundamental limitations that one has to deal with when decomposing centralized monitors and

enforcing policies over distributed systems.

- We present two different decomposition algorithms as instantiations of our blueprint.
- We discuss how our algorithms compare to previous work on decentralizing security policies, and how our results can be used to explain limitations or design choices in this work.
- We provide a characterization of the security policies that are enforceable by monitors that operate in a hierarchical manner. This is an important result, because hierarchical enforcement is often introduced as an alternative to decentralized enforcement, but the relation between the two approaches has not been yet formally analyzed.
- Finally, we identify the constraints under which monitors with simple capabilities (e.g, security automata and insertion automata) can be used in a cooperative manner to simulate monitors with seemingly more capabilities (e.g., suppression automata and edit automata)

3. Finally, we introduce a formal framework based on Probabilistic Input/Output automata that enables us to formally reason about the cost of different monitoring designs (Chapter 4). In particular:
   - We introduce the concept of *abstract schedulers* which allows fair comparison of systems, where a policy is enforced on a target by different monitors.
   - We define cost security policies and cost enforcement, richer notions of (boolean) security policies and enforcement. Cost security policies assign a cost to each trace, allowing a richer classification of traces than just good or bad. We also show how to encode boolean security policies as cost security policies.
   - Finally, we show how to use our framework to compare monitor's implementations

and we identify the sufficient conditions for constructing cost-optimal monitors.

Some of these contributions were first presented in a workshop paper written in collaboration with Lujo Bauer, Dilsun Kaynar, and Jay Ligatti [36] (Chapter 2), and in a series of workshop and journal papers written in collaboration with Lujo Bauer, Dilsun Kaynar, Fabio Martinelli, and Charles Morisset [53, 54] (Chapter 4). The author of this thesis is the primary author of all these papers.

Chapter 2

Target-aware Enforcement

2.1 Introduction

Today's computing climate is characterized by increasingly complex software systems and networks facing inventive and determined attackers. Hence, one of the major thrusts in the software industry and in computer security research is to devise ways to *provably guarantee* that software does not behave in dangerous ways or, barring that, that such misbehavior is contained and mitigated. Example guarantees for program behavior include: only accessing memory that has been allocated to them (memory safety); only jumping to and executing valid code (control-flow integrity); using no more than 10 MB of storage and 10 KB/sec network bandwidth for grid use (resource allocation); and never sending secret data over the network (a type of information flow).

As discussed in Section 1, a common mechanism for enforcing security policies on untrusted software is run-time monitoring. Run-time monitors observe the execution of untrusted applications or systems and ensure that their behavior adheres to a security policy. This type of enforcement mechanism is pervasive, and can be seen in operating systems, web browsers, firewalls, intrusion detection systems, etc. An example of monitoring is system-call interposition [37, 55, 56],

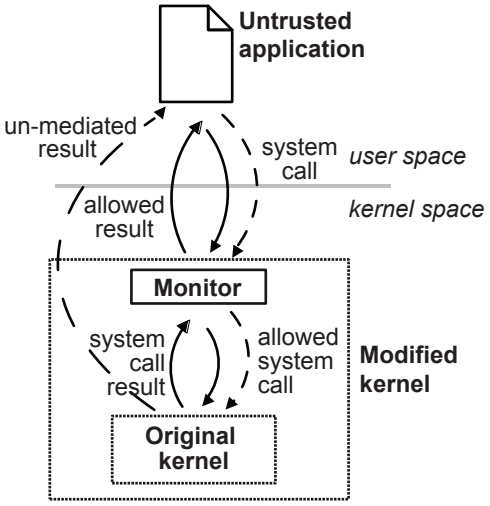


Figure 2.1: System-call interposition: dashed line shows an input-mediating monitor; solid line an input/output-mediating monitor.

depicted in Fig. 2.1: given an untrusted application and a set of security-relevant system calls, a monitor intercepts calls made by the application to the kernel, and enforces a security policy by taking corrective action when a call violates the policy.

There are many ways to implement run-time monitors in practice; understanding and formally reasoning about the specifics of their design is crucial. In Section 1 we discussed two dimensions along which instantiations may differ are: (1) *vantage*, or the monitored interface: monitors may mediate different parts of the communication between the application and the kernel; e.g., an input sanitization monitor will mediate only inputs to the kernel (dashed lines in Fig. 2.1); and (2) *action*, or trace modification capabilities: monitors may have a variety of enforcement capabilities, from being restricted to just terminating the application (e.g., when the application tries to write to the password file), to being able to perform additional corrective actions (e.g., suppress a write system call and log the attempt)[1].

Given the ubiquity of run-time monitors, it is important to minimize the likelihood of subtle

[1]In this thesis we focus on mechanisms that modify traces of the target application, but not on mechanisms that directly modify the target application, such as by rewriting.

errors both in their design and implementation. Those errors can cause the enforcement mechanism to fail, leaving the target vulnerable to attacks. Several problems in the design and implementation of interposition-based mechanisms have been identified [56], including: (1) incorrectly replicating kernel functionality in the monitor, (2) race/concurrency conditions (e.g., lack of synchronization between the monitor and the kernel in copying and interpreting system call arguments [57]), and (3) modeling a subset of the system call interface [56]. Formal methods are a useful tool for identifying and preventing such design (and implementation) errors [58].

Several formal models, such as *security automata* [27] and *edit automata* [30], have been proposed to model and reason about monitors and their enforcement capabilities (e.g., whether the monitors can insert arbitrary actions into the stream of actions that the target wants to execute). These models have been used to analyze and characterize the policies that are enforceable by the various types of monitors.

However, such models do not capture many details of the monitoring process that are important in practice (as described above), including the monitored interface and concurrency, leaving scenarios that we cannot formally reason about. In the system-call interposition scenario, for example, without the ability to model the asynchronous communication among the untrusted application, the monitor, and the kernel, it would be impossible to differentiate monitors that can mediate all security-relevant communication between the application and the kernel (solid lines in Fig. 2.1) from monitors that can mediate only some of the communication (dashed lines in Fig. 2.1). Moreover, race and concurrency bugs in the design or implementation of the interposition mechanism would not be detectable since such properties fall outside the scope of the proposed formal models of monitors.

Some recent models allow for more detailed reasoning by modeling bi-directional communication between a monitor and its environment(e.g., application and kernel). However, they do not explicitly model the application or system being monitored [59, 60]. Modeling the target ap-

plication allows for formal reasoning about important practical cases such as kernel functionality replication [56]. Moreover, modeling the target and the monitor in the same formalism allows us to analyze scenarios where a monitor designer creates customized monitors for the specific components that are monitored. Such monitors that exploit knowledge about the component that they are monitoring can enforce policies that would not be possible otherwise, i.e., without access to the target application. For example, a policy that requires that every shared file that is opened must eventually be released is outside the enforcement capabilities of run-time monitors, because the monitor does not know what the untrusted application will do in the future [27]. However, if the monitored application always releases any file that it opens, then this policy can be enforceable for that particular application. Such distinctions are often relevant in practice—e.g., when implementing a patch for a specific type or version of an application—and, thus, there is a need for formal frameworks to aid in making informed and provably correct design and implementation decisions.

In this section, we introduce a framework, i.e., a set of definitions, based on I/O automata, for more detailed reasoning about policies, monitoring, and enforcement. The I/O automaton model [1, 61] is a labeled transition model for asynchronous concurrent systems. Similar to previous previous models of run-time enforcement mechanisms we use an automata-based formalism to model asynchronous systems (e.g., the communication between the application, the monitor, and the kernel). Our framework provides abstractions for reasoning about many practical details important for run-time enforcement, and, in general, yields a richer view of monitors and applications than is typical in previous analyses of run-time monitoring. For example, our framework supports modeling practical systems with security-relevant actions that the monitor cannot mediate, rather than assuming complete mediation [26, 62]. (We discuss more such examples in Section 2.3.)

In this section, we make the following specific contributions:

- We show how I/O automata can be used to faithfully model target applications, monitors, and the environments in which monitored targets operate, as well as various types of monitors and monitoring architectures (Section 2.3).
- We extend previous definitions of security policies and enforcement to support more fine-grained formal reasoning of policy enforcement (Section 2.4).
- We show our more detailed model of monitoring forces explicit reasoning about concerns that are important for designing run-time monitors in practice, but that previous models often reasoned about only informally (Section 2.5.2). We formalize these results by identifying the policies enforceable by any monitor in our framework.
- We demonstrate how to use our framework to exploit knowledge about the target application to make design and implementation choices for more efficient enforcement (Section 2.6). For example, we exhibit constraints under which monitors with different monitoring interfaces (i.e., one can mediate more actions than the other) can enforce the same class of policies.

Roadmap We start by briefly reviewing I/O automata (Section 2.2). We then informally show how to model monitors and targets in our framework and discuss some of the benefits of this approach (Section 2.3). Next, we formally define policies and enforcement (Section 2.4). Then, we show several examples of the meta-theoretical analysis that our framework enables by (a) providing some upper bounds for enforceable policies (Section 2.5), and (b) exposing constraints under which seemingly different monitoring architectures can enforce the same classes of policies (Section 2.6).

2.2 I/O Automata

I/O automata are a labeled transition model for asynchronous concurrent systems [1, 61]. In this section we review aspects of I/O automata that we build on in the rest of the thesis. We encourage readers familiar with I/O automata to skip to Section 2.3. In this thesis we follow the formal definitions and presentation of I/O automata that is typical in the literature [1, 61].

Given a function $f : X \to W$ we write $\mathsf{dom}(f)$ for X (i.e., the domain of f) and $\mathsf{range}(f)$ for W (i.e., the range of f).

I/O automata are typically used to describe the behavior of a system interacting with its environment. The interface between an automaton A and its environment is described by the *action signature* $Sig(A)$ of A. The signature $Sig(A)$ is a triple of disjoint sets—$Input(A)$, $Output(A)$, and $Internal(A)$. We write $acts(A)$ for $Input(A) \cup Output(A) \cup Internal(A)$. We sometimes refer to output and internal actions as *locally-controlled* actions and we write $Local(A)$ for $Output(A) \cup Internal(A)$.

Formally, an I/O automaton A consists of:

1. an action signature, $Sig(A)$;
2. a (possibly infinite) set of *states*, $states(A)$;
3. a nonempty set of *start states*, $start(A) \subseteq states(A)$;
4. a transition relation, $trans(A) \subseteq states(A) \times acts(A) \times states(A)$, with the property (called *input enabledness*) that for every state q and input action a there is a transition $(q, a, q') \in trans(A)$; and
5. an equivalence relation $Tasks(A)$ partitioning the set $Local(A)$ into at most a countable number of equivalence classes.

If A has a transition (q, a, q') then we say that action a is *enabled* in state q. When only input

actions are enabled in q, then q is called a *quiescent* state. The set of all quiescent states of an automaton A is denoted by *quiescent*(A). A task C is enabled in a state q if some action in C is enabled in q.

An *execution* e of A is a finite sequence, $q_0, a_1, q_1, \ldots, a_r, q_r$, or an infinite sequence $q_0, a_1, q_1, \ldots, a_r, q_r, \ldots$, of alternating states and actions such that $(q_k, a_{k+1}, q_{k+1}) \in$ *trans*(A) for $k \geq 0$, and $q_0 \in$ *start*(A). A *schedule* is an execution without states in the sequence, and a *trace* is a schedule that consists only of input and output actions. An *execution*, *trace*, or *schedule module* describes the behavior exhibited by an automaton. An execution module E consists of a set *states*(E), an action signature *Sig*(E), and a set *execs*(E) of executions. Schedule and trace modules are similar, but do not include states. The sets of executions, schedules, and traces of an I/O automaton X are denoted by *execs*(X), $scheds(X)$, and $traces(X)$. Similarly, the sets executions, schedules, and traces of an execution module X are denoted by *execs*(X), $scheds(X)$, and $traces(X)$. Sets of schedules and traces for schedule and trace modules are denoted similarly.

Given a sequence s and a set X, $s|X$ denotes the sequence which results from removing from s all elements that do not belong in X. Similarly, for a set of sequences S, $S|X = \{(s|X) \mid s \in S\}$. The symbol $\cdot$ denotes the empty sequence. We write $\sigma_1; \sigma_2$ for the concatenation of two schedules or traces,the first of which has finite length. When σ_1 is a finite prefix of σ_2, we write $\sigma_1 \preceq \sigma_2$, and in the case of a strict finite prefix $\sigma_1 \prec \sigma_2$. Given a set of actions Σ (e.g., *acts*(A), *Input*(A), etc.) we write $(\Sigma)^\star$ to denote the set of finite sequences of actions and $(\Sigma)^\omega$ to denote the set of infinite sequences of actions. The set of all finite and infinite sequences of actions is $(\Sigma)^\infty = (\Sigma)^\star \cup (\Sigma)^\omega$. Given two sequences s and t we write $s||t$ to denote the set that contains all interleavings of s and t.

The I/O automata definition also includes the equivalence relation *Tasks*(A). This is used in the definition of *fairness*. Fairness means that the automaton will give fair turns to each of its tasks while executing.

An execution *e* of an I/O automaton A is said to be *fair* if for each class C of *Tasks*(A): (1) if *e* is finite, then C is not enabled in the final state of *e*, or (2) if *e* is infinite, then *e* contains either infinitely many events from C or infinitely many occurrences of states in which C is not enabled.

Fairness eliminates the need to model a scheduler in the system. Specifically, when reasoning about practical systems, instead of explicitly modeling a scheduler one can simply reason about a fair version of the system. The type of fairness that I/O automata define is called "weak fairness", and is only one of the many different types of fairness [63].

Given an automaton or a module A we denote the sets of fair executions, schedules and traces by *fairexecs*(A), *fairscheds*(A) and *fairtraces*(A).

An automaton that models a complex system can be constructed by *composing* automata that model the system's components. When composing automata (or modules) S_i, where i belongs to some index set I, the automata's signatures are called *compatible* if their output actions are disjoint and the internal actions of each automaton are disjoint with all actions of other automata. More formally, The actions signatures $S_i : i \in I$ are called compatible if for all $i, j \in I, i \neq j$:

1. *Output*$(S_i) \cap$ *Output*$(S_j) = \emptyset$
2. *Internal*$(S_i) \cap$ *acts*$(S_j) = \emptyset$

When the signatures are compatible we say that the corresponding automata (or modules) are compatible too. The composition $A = \prod_{i \in I} A_i$ of a set of compatible automata $\{A_i : i \in I\}$ is defined as:

1. *states*$(A) = \prod_{i \in I}$ *states*(A_i)
2. *start*$(A) = \prod_{i \in I}$ *start*(A_i),
3. *Sig*$(A) = \prod_{i \in I}$ *Sig*$(A_i) =$

   $\Big($*Output*$(A) = \cup_{i \in I}$*Output*(A_i),

   Internal$(A) = \cup_{i \in I}$*Internal*(A_i),

$Input(A) = \cup_{i \in I} Input(A_i) - \cup_{j \in I} Output(A_j)\Big)$,

4. *trans*(A) is equal to the set of triples (q, a, q')[2] such that for all $i \in I$

   (a) if $a \in acts(A_i)$ then $(q_i, a, q_i') \in trans(A_i)$, and

   (b) if $a \notin acts(A_i)$ then $q_i = q_i'$

5. $Tasks(A) = \cup_{i \in I} Tasks(A_i)$

Similarly, we can define the composition of execution, schedule, and trace modules. Given a countable collection of compatible schedule[3] modules $\{S_i, i \in I\}$ we define the composed schedule module $S = \prod_{i \in I} S_i$:

1. $Sig(S) = \prod_{i \in I} Sig(S_i)$,
2. *execs*(S) is the set of schedules s such that the subsequence s' of s consisting of actions of S_i, is a schedule of S_i for every $i \in I$.

Unlike models such as CCS [64], composing two automata that share some actions (i.e., outputs of one automaton that are inputs to the other) causes those shared actions to be regarded as output actions of the composition. Those actions that are required to be internal need to be explicitly classified as such using the *hiding* operation. If S is a signature and $\Phi \subseteq Output(S)$, then $hide_\Phi(S)$ is defined to be the new signature S', where $Input(S') = Input(S)$, $Output(S') = Output(S) - \Phi$, and $Internal(S') = Internal(S) \cup \Phi$. Given an I/O automaton A and $\Phi \subseteq Output(A)$, $hide_\Phi(A)$ is the automaton A' obtained by replacing $Sig(A)$ with $Sig(A') = hide_\Phi(Sig(A))$.

The operation of *renaming*, on the other hand, changes the names of actions, but not their types (e.g., renaming does not change an input action a to an output action b; it changes an input action a to an input action b). A renaming f is a total injective mapping between sets

[2]Note that $q = \prod_{i \in I} q_i$ and $q' = \prod_{i \in I} q_i'$ where $q_i, q_i' \in states(A_i)$.

[3]In this section, for brevity, our analyses focus on schedules. Trace modules, used in later sections, have similar definitions.

of actions. f is said to be *applicable* to an automaton if the domain of f contains the actions of the automaton. If f is applicable to an automaton A, then the automaton *rename*(A) is the automaton with the states and start states of A; with the input, output and internal actions *rename*(*Input*(A)), *rename*(*Output*(A)), *rename*(*Internal*(A)) respectively; with the transition relation $\{(q, \textit{rename}(a), q') : (q, a, q') \in \textit{trans}(A)\}$; and the equivalence relation $\{(\textit{rename}(a),$ *rename*$(a')) : (a, a') \in tasksA\}$. We define a renaming function f to be applicable to schedules as follows: given a schedule $s = a_1; a_2; \ldots$ then $f(s) = f(a_1); f(a_2); \ldots$.

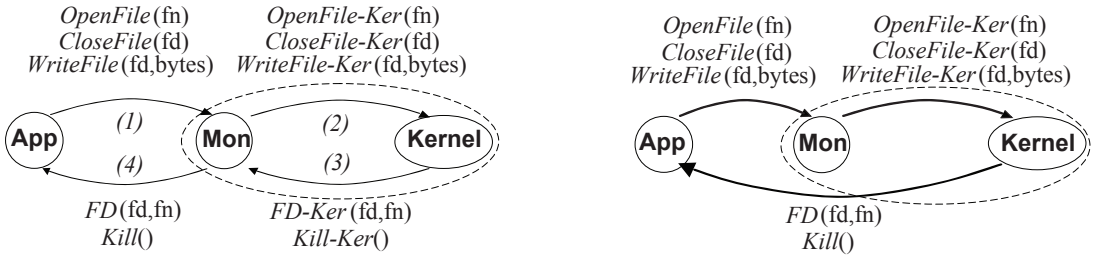
2.3 Specifying Policies, Targets, and Monitors

In this section we define security policies and show how monitors and targets can be modeled using I/O automata, building on the example of system-call interposition in Fig. 2.2. We begin by specifying the system-call interposition using I/O automata in Section 2.3.1. Then, in Section 2.3.2 we discuss the formalization choices of run-time monitor design using I/O automata, and in Section 2.3.3 we specify how to define security policies in our framework. Next, in Section 2.3.4, we shortly review two notable models of run-time enforcement mechanisms, security and edit automata, and in Section 2.3.5 we explain how to encode monitors of previous frameworks (i.e., security and edit automata) in our framework. Finally, in Section 2.3.6 we provide some concluding remarks on the advantages of using our framework for formalizing enforcement scenarios.

2.3.1 Modeling Targets and Monitors with I/O automata

In our framework we model targets (the entities to be monitored) and monitors as I/O automata. We let the metavariables T and $\mathcal{M}$ range over targets and monitors respectively. Targets composed with monitors are called *monitored targets* (e.g., the modified kernel in Fig. 2.1). A monitored target can itself be a target for another monitor.

Suppose that an application's actions are *OpenFile*, and *CloseFile* system calls; the kernel's actions are *FD* (to return a file descriptor) and the *Kill* system call. The application can make a request to open a file *fn*, and the kernel returns a file descriptor *fd* in response to the request for *fn*. The application can then read from or write to the file, and when it is done, close *fd*. Finally, a *Kill* action terminates the application and clears all requests. As noted by more recent work [59, 60], the formalization of such situations, in which the target's actions depend on results returned by



(a) Input/output-mediating monitor

(b) Input-mediating monitor

Figure 2.2: I/O automata interface diagrams of kernel, application, and monitor

the environment, was outside the scope of original models of run-time monitors (e.g., security automata [27] and edit automata [30]).

Let us assume we want to enforce a security policy[4] that restricts the resources that an application has access to–in our case file descriptors. One way to enforce this policy is to interpose a monitor between the application and the kernel that keeps track of the number of open files that an application has requested. Once that number exceeds the predefined threshold set by the policy, the monitor terminates the application with a *Kill* action.

Fig. 2.2a shows I/O automata interface diagrams of the complete monitored system consisting of the application and the monitored kernel.

The application's and kernel's interfaces differ only in that the input actions of the kernel are output actions of the application and vice versa. This models the communication between the application and the kernel when considered as a single system. Paths (2) and (3) in Fig. 2.2a represent communication between the monitor and the kernel through the renamed actions of the kernel (using the renaming operation of I/O automata, Section 2.2): e.g., *OpenFile*(x) becomes *OpenFile-Ker*(x). Renaming models changing the target's interface by adding hooks that allow the monitor to intercept the target's actions. In practice, this is often accomplished by rewriting

[4]Although the formal definition of security policies is given in Section 2.3.3, this section does not rely on this definition and can be read independently.

Signature: Input: *OpenFile*(*fn*), where *fn* is a *file_name*
(type *file_name* = nat)
CloseFile(*fd*), where *fd* is a *file_descriptor*
Kill-Ker(), *FD-Ker*(*fd*,*fn*), where *fd* is a *file_descriptor*
(type *file_descriptor* = nat)
and *fn* is the corresponding *file_name*.
Output: *Kill*(), *FD*(*fd*,*fn*), where *fd* is a *file_descriptor*
(type *file_descriptor* = nat)
and *fn* is the corresponding *file_name*
OpenFile-Ker(*fn*), where *fn* is a *file_name*
(type *file_name* = nat)
CloseFile-Ker(*fd*), where *fd* is a *file_descriptor*.

States: *req_list* : List of elements of type *file_name*
close_list : List of elements of type *file_descriptor*
resp_list : List of elements of type ⟨*file_descriptor*,*file_name*⟩
kill : flag of type *bool*

Start States: *req_list* = *nil*
close_list = *nil*
resp_list = *nil*
counter = 0
kill = *false*

Figure 2.3: Signature and states of monitor I/O automaton enforcing "no more than n files open per application"

the target. Finally, we also *hide* the communication between monitor and the kernel so that it remains concealed from the application (denoted by the dotted line around the monitored kernel automaton). This is because we model a monitoring process that is transparent to the application (i.e., the application remains unaware that the kernel is monitored).

An I/O automaton definition for the monitor is shown in Fig. 2.3 and Fig. 2.4, using the standard precondition-effect style of writing transition relations for I/O automata [1, 61].

Let us assume that we are at a point in the execution of the monitored system where the application has met its quota of requested files. Then the next time that the application requests to open a new file, e.g., named x, the monitor intercepts this request and kills the application. This scenario is modeled as follows using the above. Assuming that σ denotes the execution of

Transitions:

OpenFile(*fn*)
Effect: $req\_list = req\_list@[fn]$
$counter = counter + 1$
if $counter > n$ then $kill = true$

CloseFile(*fd*)
Effect: $close\_list = close\_list@[fd]$
$counter = counter - 1$

FD-Ker(*fd*,*fn*)
Effect: $resp\_list = resp\_list@[\langle fd, fn\rangle]$

Kill-Ker()
Effect: $kill = true$

OpenFile-Ker(*fn*)
Precondition: $\neg empty(req\_list)$ and $\exists(fn : file\_name) \in req\_list$,
where *empty* is a predicate on lists that returns *true* whenever its argument is an empty list
Effect: $req\_list = (req\_list \backslash fn)$, where $(req\_list \backslash fn)$ denotes the function that removes the element *fn* from list *req_list*

CloseFile-Ker(*fd*)
Precondition: $\neg empty(close\_list)$ and $\exists(fd : file\_descriptor) \in close\_list$
Effect: $close\_list = (close\_list \backslash fd)$, where $(close\_list \backslash fd)$ denotes the function that removes the element *fd* from list *close_list*

FD(*fd*,*fn*)
Precondition: $\neg empty(resp\_list)$ and
$\exists(\langle fd, fn\rangle : \langle file\_descriptor, file\_name\rangle) \in req\_list$
Effect: $resp\_list = (resp\_list \backslash \langle fd, fn\rangle)$, where $(resp\_list \backslash x)$ denotes the function that removes the element *x* from list *resp_list*

Kill()
Precondition: $kill = true$
Effect: $req\_list = nil$
$close\_list = nil$
$resp\_list = nil$
$counter = 0$
and $kill = false$.

Tasks: Each output action belongs to a unique task named after the action itself.

Figure 2.4: Transitions of monitor I/O automaton enforcing "no more than n files open per application"

the system so far, then the schedule s that describes this scenario is $s = \sigma$;*OpenFile*(x);*Kill*().

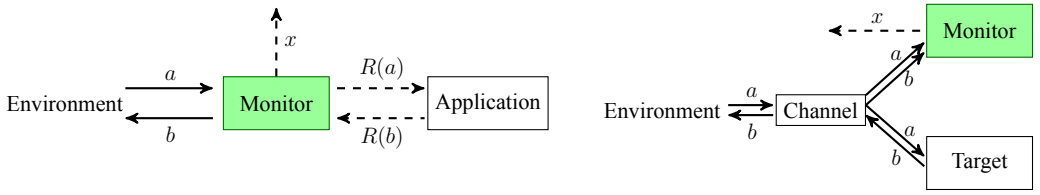
Note that s is an example of a *fair schedule*. Fairness will be important when we discuss transparency in Section 2.4.1 and Section 2.5.3. An example of an unfair schedule would be

$t = \sigma$;*OpenFile*(x). This schedule is not *fair* because, as shown in Fig. 2.4, when the monitor I/O automaton receives the input action *OpenFile*(x), it goes to a state q where the *counter* is greater than n, and thus the flag *kill* becomes *true*. This means that the automaton is at a state where the local action *Kill*() is enabled. Since each action belongs to a unique task, this means that from this state the task *Kill*() is enabled. Thus, the definition of fairness in Section 2.2 is violated, and t is not a fair schedule. Note that in the case of s, where the last action is *Kill*(), the automaton goes to a state where all lists, flag, and counter are reset. From such a state no local action, and thus task, is enabled, so s satisfies the constraints of fairless.

2.3.2 Modeling Monitoring Decisions with I/O automata

In our system-call interposition example from Section 2.1 we described some choices for the monitor designer to make, such as (1) the interface to be monitored, or vantage, (e.g., only mediate input actions), and (2) the trace modification capabilities of the monitor, or action. We next describe how to express the above choices in our model.

Modeling the Monitored Interface. By appropriately restricting the renaming function applied to the target, we can model different monitoring architectures (e.g., input sanitization, Section 2.1). For example, in Fig. 2.2b, we rename only the input actions of the kernel (i.e., *OpenFile*,*CloseFile*, and *WriteFile*). This allows us to model monitors that mediate inputs sent to the target and can prevent, for example, SQL injections attacks. Similarly, by renaming only the outputs of the target we can model monitors that only mediate output actions (and can prevent, for example, cross-site scripting attacks). In these examples, the monitors are *actively* mediating security relevant actions. This means that the monitor can choose to allow an action, block it, or replace it. However, there is a class of security mechanisms that do not *actively* interfere with observed traffic: IDSs. IDSs analyze traffic they intercept (or receive from already created logs)



(a) Actively observing IDS (b) Passively observing IDS

Figure 2.5: Designs of Intrusion Detection Systems

and try to identify instances of attacks. Although IDSs cannot prevent attacks, they are extremely useful in identifying attacks that cannot be identified in real-time or analyzing patterns of received traffic in order to create signatures of good or malicious traffic [50].

In Fig. 2.5a we see how to model an IDS that is interposed between the target application and the environment. Note that we use renaming (as we discussed previously) to model the fact that the monitor is interposed between the environment and the target application; however, we have to model in the monitor's transition relation the fact that the monitor *must* forward all actions it receives. In addition, we have a new security relevant action x which models the alerts that the IDS, typically, sends to the analyst. We are using an additional action x because typically the alert action is not something that the target application or the environment can exhibit.

However, there are situations where IDSs are not interposed between the target application and the monitor. For instance, an IDS could be placed at the spanning port of a switch where traffic between the target application and the environment is sent to the monitor. This is depicted in Fig. 2.5b. Note that no renaming is used, but the monitor has as inputs all the external actions of the environment and the target application (thus modeling the fact that it can observe all security relevant actions). As in the actively observing IDS, there is a new action x to model the alerts that the IDS is producing.

Modeling Implementation Aspects of Monitors. As we described in Section 2.3.4 monitors can have differing enforcement capabilities: if the target wants to execute some action that will violate the policy, a monitor can either halt the target [27] or take some corrective action [34]. In practice, there are additional choices that we can make regarding implementations of monitors: whether the monitor can edit input before forwarding it to the target application; the extent to which the monitor can ignore the application; and the extent to which the monitor can use the application as an oracle or simulator to discover, in a controlled way, how it would respond to different input actions. The added expressiveness of I/O automata allows us to model these different implementation choices. However, if we focus on a uni-directional communication path from the target to the monitor and then to the environment, then our framework can express types of monitors defined in previous work such as security and edit automata (which we describe in more detail in Section 2.3.4).

Returning to our example in Fig. 2.2: to model a monitor that halts the kernel once the kernel outputs an *FD-Ker*(*fd*,*fn*) with an already-assigned file descriptor *fd*, we add a transition to the monitor that, upon receipt of the "bad" action, takes the monitor to a specific halt state. Since the monitor is input enabled, that transition can be made regardless of the state of the monitor. Once the monitor goes into this halt state, the only enabled output action will be a *halt* action to kill the kernel. The kernel will need to have this *halt* action as an input action, and an appropriate transition to stop its execution. Since the monitor is input enabled, it may (even in the halt state) receive invalid actions from the kernel until the kernel is halted. In previous models, for any action that the target wanted to execute, the target would wait for the monitor to finish considering that action before trying to execute the following one; in other words, the target and the monitor were synchronized. However, in our framework the monitor does not always have such control over the target (unless we are modeling a scenario where the target and the monitor are indeed synchronized). These issues affect the policies that are enforceable by the monitor, since the

target might try to execute a series of invalid actions before the monitor gets a chance to take corrective action. We revisit this point in Section 2.3.5 where we provide more technical details on how to encode truncation and edit monitors using I/O automata.

2.3.3 Security Policies

Schneider defined a *security policy* as a predicate on (or equivalently, a set of) sets of action sequences [27]. Moreover, he identified a specific class of policies that are enforceable by run-time monitors, called *properties*. A policy $\mathcal{P}$ is a property if there exists a set of action sequences $\hat{P}$ such that for every set X, $X \in \mathcal{P}$ if and only if $X \subseteq \hat{P}$. Thus, the set $\hat{P}$ uniquely identifies the policy $\mathcal{P}$, and instead of talking about a policy $\mathcal{P}$ one can talk about the property (induced by the set) $\hat{P}$.

Following Schneider [27], we define a *policy* as a *set* of schedule[5] modules, i.e., a set of pairs where each pair's first component is a signature– a triple of input, internal, and output actions– and second component is a set of sequences of actions that belong to some set in the signature. In our framework, a property (as a set of sequences of actions) corresponds to a schedule module. We let the metavariables $\mathcal{P}$ and $\hat{P}$ range over policies and their elements (i.e., schedule modules) respectively.

The novelty of this definition of policy compared to previous ones [27] is that each element of the policy is not a set of automaton runs, but, rather, a pair of a set of runs, i.e., schedules, and a signature. The signature describes explicitly the actions that are relevant to a policy (i.e., the security relevant actions), even if they do not appear in the specific set of runs. This is useful in a number of ways. When enforcing a policy on a system composed of several previously defined components, the signatures can clarify whether a policy that is being enforced on one

[5]Our analyses equally apply to execution and trace modules, but, for brevity, in this chapter, we discuss only schedule modules.

component also reasons about (e.g., prohibits or simply does not care about) the actions of another component.

For example, let us revisit our running example from Section 2.3.1. Let S_1 be the signature that contains *Open*, *FD*, and *Close* system calls, i.e., $S_1 = \{$*Open*, *FD*, *Close*$\}$ and S_2 the signature that contains in addition the system call *SocketRead*, i.e., $S_2 = \{$*Open*, *FD*, *Close*, *SocketRead*$\}$. Moreover, let T_1 and T_2 be sets of schedules that contain the action sequence "*Open*;*FD*;*Close*", i.e., $T_1 = T_2 = \{\langle$*Open*;*FD*;*Close*$\rangle\}$. Then $M_1 = \langle S_1, T_1 \rangle$, and $M_2 = \langle S_2, T_2 \rangle$ are schedule modules. The set $P = \{M_1, M_2\}$ is a policy ta describes that every file that is opened and assigned a file descriptor must be eventually closed. Note that in the schedule module M_1 all system calls other than *Open*, *FD*, and *Close* are security irrelevant and thus permitted. Thus, if the application tries to read from a socket then this action will be allowed. On the other hand, the signature of the schedule module M_2 contains the system call *SocketRead*, but any behavior exhibiting this system call are prohibited, since there are no schedules in T_2 that contain the action *SocketRead*.

Our definition of a policy as a set of modules resembles that of a hyperproperty [65] and previous definition of policies (modulo the signature of each schedule module) and captures common types of policies such as access control, noninterference, information flow, and availability.

2.3.4 Security, Truncation, Suppression, and Edit Automata

Two models of run-time enforcement mechanisms that have been extensively analyzed in the past are security automata [27] (also referred to as truncation automata [30]) and edit automata [30]. In this section we shortly review them and provide some formal details of these models which we use in Section 2.3.5 where we discuss how to translate them in I/O automata. Note that this section assumes the definition of security policies as was given by Schneider [27]. This and the next section can be skipped without affecting the readability and understanding of the rest of this

$$\frac{\sigma = a;\sigma' \qquad \gamma(q,a) = q'}{(q,\sigma) \xrightarrow{a}_T (q',\sigma')}\ T\ step \qquad \frac{\sigma = a;\sigma' \qquad \gamma(q,a) = halt}{(q,\sigma) \xrightarrow{halt}_T (q',\ \cdot\ )}\ T\ halt$$

Figure 2.6: Operational semantics of truncation automata

$$\frac{\sigma = a;\sigma' \qquad \delta(q,a) = (q',a')}{(q,\sigma) \xrightarrow{a'}_E (q',\sigma)}\ E\ ins \qquad \frac{\sigma = a;\sigma' \qquad \delta(q,a) = (q',\ \cdot\ )}{(q,\sigma) \xrightarrow{\cdot}_E (q',\sigma')}\ E\ sup$$

Figure 2.7: Operational semantics of edit automata

chapter, especially for readers familiar with security and edit automata. However, the details will be useful in the formal statements and proofs in Chapter 3 where we discuss how distributed truncation automata may be able to simulate global suppression automata.

The two models of monitors differ on how they react to the sequence of actions that the target wants to execute. Security automata are execution recognizers: if the target wants to execute an action that is permitted by the security policy then the monitor forwards this action to the environment; otherwise the monitor halts the target. Since security automata are equivalent to truncation automata [30], we will discuss truncation automata here in order to minimize the use of different formal definitions. Given an action set Σ_{A_T}, a truncation automaton A_T is defined as a triple $A_T = \langle Q, q_0, \gamma \rangle$, where Q is the set of its states, q_0 its start state, and γ its transition function. Its operational semantics are presented in Fig. 2.6. Given a sequence of actions σ that the target wants to execute, the transition function γ of a truncation automaton specifies how the automaton moves from a state q through the first action a of σ to a state q'. The conclusions of the two rules completely describe the behavior of the automaton: it can either exhibit the action a and move on to examine the next action the target wants to execute, or halt the target.

Contrary to security automata, edit automata are execution transformers: given a sequence of actions that the target wants to execute, the monitor can either *insert* some actions to the output stream without consuming the inputs that the targets want to execute, or *suppress*, i.e., consume, the current input action and examine the next one while outputting nothing to the environment.

$$\frac{\sigma = a;\sigma' \qquad \delta(q,a) = (q',a)}{(q,\sigma) \xrightarrow{a}_E (q',\sigma)}\ S\ step \qquad \frac{\sigma = a;\sigma' \qquad \delta(q,a) = (q',\ \cdot\ )}{(q,\sigma) \dot{\rightarrow}_S (q',\sigma')}\ S\ sup$$

Figure 2.8: Operational semantics of suppression automata

Given an action set Σ_{A_E}, an edit automaton A_E is defined as a triple $A_E = \langle Q, q_0, \gamma \rangle$, where Q is the set of its states, q_0 its start state, and δ its transition function. This behavior is captured formally by the rules in Fig. 2.7.

A special type of edit automata are *suppression* automata. The difference between edit and suppression automata is that edit automata can insert arbitrary actions in the output stream, whereas suppression automata can only insert the action that the target wants to execute. Thus, suppression automata can either *allow* good actions, or *suppress* bad actions. Suppression automata are similar to truncation automata, but instead of halting the target once something bad is about to be executed, as truncation automata do, they allow the target to continue operating. This behavior is captured formally by the rules in Fig. 2.8.

Note that in these models the target is not formally specified (e.g., through specifying its transition function).

2.3.5 Translating Security and Edit Automata to I/O automata

In this section we discuss how to translate monitors expressed in previous models to our model. The translation has several steps, and care has to be taken to account for the added expressiveness of I/O automata.

First, we extend the transition function of an automaton with transitions that model the input enabledness of the I/O automata. Note that the operational semantics of truncation (Fig. 2.6, suppression (Fig. 2.8), and edit automata Fig. 2.7), assume that the actions that the target wants to execute are available to the monitor upon request.

Second, in truncation, suppression, and edit automata the actions that the target was sending to the monitor belonged to the same action set as the actions that the monitor was forwarding to the environment. However, in I/O automata, this is not possible since the input and output actions must belong to disjoint sets. Thus, we define a bijection that maps the inputs of the corresponding automata to fresh output actions.

The third step takes care of the implicit assumptions made by previous models and exposed by our framework: a run-time monitor does not control how the target produces actions. More specifically, in previous models, the target and the monitor are synchronized: after the target tries to execute an action (an action that is intercepted by the monitor), the target blocks until the monitor is ready to receive the next security relevant action. However, in our framework the monitor does not have such a control over the target. For this reason, the monitors need to have some data structure (e.g., a queue) to buffer the inputs from the target, and when given a turn to execute local actions (i.e., by a scheduler formalized through fairness assumptions), the monitors dequeue the corresponding actions and react according to their specified transition relation. Note, that another approach would be to consider a specific type of fairness (instead of the one that I/O automata use, i.e., weak fairness), that would give priority to the local actions of the monitor and would allow inputs to arrive only when the monitor does not have more actions to execute. However, in this chapter we adopt the former approach, i.e., we assume the fairness definition of I/O automata and always provide monitors with queues to buffer input actions.

The final step, is specific to truncation automata: we assume that the target has a *halt* input action that will guarantee its termination (as in our system call interposition, with the *kill* system call).

Next, we show the results of applying the above translation to truncation automata, and edit automata [30]. We will refer to the resulting I/O automata as *truncation monitors* and *edit monitors*. Translations of other types of automata can be defined similarly. For instance, the result of

translating suppression automata, i.e., *suppression monitors*, are exactly the same as edit monitors, with the only difference being that for every action α' that the edit monitor has in the transition relation, the suppression monitor has, instead, the action α.

For the translation of truncation automata to I/O automata we assume that the target can be terminated by a *stop* action. Given a truncation automaton $A_T = \langle Q, q_0, \gamma \rangle$ that is defined over some action set Σ_{A_T}, we define a set Σ_F containing fresh actions (i.e., $\Sigma_F \cap \Sigma_{A_T} = \emptyset$) such that $|\Sigma_F| = |\Sigma_{A_T}|$[6] and a truncation monitor $M_T = \langle \mathit{Sig}(M_T), \mathit{states}(M_T), \mathit{start}(M_T), R_{M_T}, \mathit{Tasks}(M_T) \rangle$, where:

1. $\mathit{Sig}(M_T) = \langle \mathit{Input}(M_T), \mathit{Internal}(M_T), \mathit{Output}(M_T) \rangle$, where:
   (i) $\mathit{Input}(M_T) = \Sigma_{A_T}$,
   (ii) $\mathit{Internal}(M_T) = \emptyset$,
   (iii) $\mathit{Output}(M_T) = \{f(x) \mid x \in \mathit{Input}(M_T)\} \cup \{\mathit{stop}\}$, where $f : \mathit{Input}(M_T) \xrightarrow[\mathit{onto}]{1-1} \Sigma_F$.
2. $\mathit{states}(M_T) = (Q \times ((\mathit{Input}(M_T)))^\star) \cup \{\langle \mathit{halt}, \cdot \rangle\}$; i.e., the state of automaton together with the queue to buffer inputs from the target, plus an additional halt state,
3. $\mathit{start}(M_T) = q_0 \times \{ \cdot \}$,
4. $R_{M_T} =$
   $\{\langle \langle q, \sigma \rangle, \iota, \langle q, \sigma; \iota \rangle \rangle \mid \langle q, \sigma \rangle \in \mathit{states}(M_T)$ and $\iota \in \mathit{Input}(M_T)\}$
   $\cup \{\langle \langle q, \alpha; \sigma \rangle, f(\alpha), \langle q', \sigma \rangle \rangle \mid \langle q, \alpha; \sigma \rangle \in \mathit{states}(M_T)$ and $\gamma(q, \alpha) = q'\}$
   $\cup \{\langle \langle q, \alpha; \sigma \rangle, \mathit{stop}, \langle \mathit{halt}, \cdot \rangle \rangle \mid \langle q, \alpha; \sigma \rangle \in \mathit{states}(M_T)$ and $\gamma(q, \alpha) = \mathit{halt}\}$,
5. Each action in $\mathit{local}(M_T)$ defines a unique equivalence class.

Next, we show how to translate an edit automaton to an I/O automaton. Given an edit automaton $A_E = \langle Q, q_0, \delta \rangle$ that is defined over some action set Σ_{A_E}, we define a set Σ_F containing

[6] $|A|$ denotes the cardinality of the set A.

fresh actions (i.e., $\Sigma_F \cap \Sigma_{A_E} = \emptyset$) such that $|\Sigma_F| = |\Sigma_{A_E}|$ and an edit monitor $M_E = \langle \mathit{Sig}(M_E), \mathit{states}(M_E), \mathit{start}(M_E), R_{M_E}, \mathit{Tasks}(M_E)\rangle$, where:

1. $\mathit{Sig}(M_E) = \langle \mathit{Input}(M_E), \mathit{Internal}(M_E), \mathit{Output}(M_E)\rangle$, where:
   (i) $\mathit{Input}(M_E) = \Sigma_{A_E}$,
   (ii) $\mathit{Internal}(M_E) = \emptyset$,
   (iii) $\mathit{Output}(M_E) = \{f(x) \mid x \in \mathit{Input}(M_E)\}$, where $f : \mathit{Input}(M_E) \xrightarrow[\mathit{onto}]{1-1} \Sigma_F$.
2. $\mathit{states}(M_E) = (Q \times ((\mathit{Input}(M_E)))^\star)$; i.e., the state of automaton together with the queue to buffer inputs from the target,
3. $\mathit{start}(M_E) = q_0 \times \{\ \cdot\ \}$,
4. $R_{M_E} =$
   $\{\langle\langle q, \sigma\rangle, \iota, \langle q, \sigma; \iota\rangle\rangle \mid \langle q, \sigma\rangle \in \mathit{states}(M_E)$ and $\iota \in \mathit{Input}(M_E)\}$
   $\cup\ \{\langle\langle q, \alpha; \sigma\rangle, f(\alpha'), \langle q', \alpha; \sigma\rangle\rangle \mid \langle q, \alpha; \sigma\rangle \in \mathit{states}(M_E)$ and $\delta(q, \alpha) = (q', \alpha')\}$
   $\cup\ \{\langle\langle q, \alpha; \sigma\rangle,\ \cdot\ , \langle q', \sigma\rangle\rangle \mid \langle q, \alpha; \sigma\rangle \in \mathit{states}(M_E)$ and $\delta(q, \alpha) = (q',\ \cdot\ )\}$,
5. Each action in $\mathit{local}(M_E)$ defines a unique equivalence class.

In both cases, a simple inductive argument suffices to show that when a truncation automaton outputs a sequence of actions $\tau = \tau_0; \tau_1; \ldots; \tau_n$ as a response to a sequence of input actions $\sigma = \sigma_0; \sigma_1; \ldots; \sigma_m$, then a truncation monitor constructed as above will output the (equivalent modulo renaming) sequence of actions $\tau' = f(\tau_0); f(\tau_1); \ldots; f(\tau_n)$ as a response to the same sequence of input actions σ. A similar argument applies to the translation of edit automata to edit monitors.

2.3.6 Discussion

In Section 2.1 we discussed the importance of formally modeling sufficiently many details of scenarios such as using interposition mechanisms for policy enforcement. In Section 2.3.2 and Section 2.3.5 we discussed how to use I/O automata to model such details (for instance the target application and the asynchronous communication between the target and the monitor) that previous models did not.

In this section we point out certain more general benefits of using expressive frameworks like I/O automata for modeling enforcement scenarios. In particular, our model exposes many details that often remain implicit or informal when reasoning about enforcement, including the following.

1. *Monitored interface/Target modification:* The way a monitor is integrated with a target is not expressed in security automata[27], edit automata [30], or Mandatory Results Automata (MRA) [59]. Our model makes this integration explicit through the renaming operation of I/O automata. Two of the benefits of this ability are: (a) acknowledging that any target (or its interface) needs to be re-written, even minimally, so that the security relevant actions are intercepted by the monitor, and (b) providing an easy syntactic check for complete mediation, e.g., if not all actions have been renamed in the transition relation of the target, then there exists a case where the execution of a security relevant action bypasses the monitor. This captures the typical monitoring approach in which a monitor intercepts a target's security-relevant actions, but does not otherwise modify a target's state and behavior. Tighter integration can be modeled by changing the target's transition relation, but we do not explore that in this thesis.

2. *Complete mediation/instrumentation:* Monitors typically assume that all security-relevant actions of a target can be mediated. Our model takes a more nuanced view, which admits

that there may exist security relevant actions that the monitor cannot observe or mediate. Such actions are either (1) labeled as internal to a target (and the I/O automata formalism prevents them from being exposed to the monitor), or (2) different from any monitor's input actions (and I/O automata composition prohibits communication between them). This way we can model scenarios where, for example, a monitor installed by a non-administrator user has only partial access to the kernel's system calls.

Reasoning explicitly about these details allows us to close the gap between monitors in practice and their theoretical abstractions while receiving a better insight about the space of policies that are enforceable by monitors. It can also shed light on issues related to the design of monitors. For example, attempting to encode a specific system in our framework can show that the monitor is not sufficiently protected from a target; conversely, a system faithfully implemented based on a model in our framework will inherit properties that are explicit in our model, including monitor integrity. This reasoning also allows us to identify practical constraints in the formal analysis of enforcement powers of monitors. For example, as we saw in the translation of the truncation automata above, and as we will further analyze in Section 2.5, we must acknowledge that there are security relevant aspects of the enforcement process that our outside the monitor's control, for example the ability to control the target by either synchronizing it with the monitor or controlling its local actions.

2.4 Policy Enforcement

In this section we introduce two definitions of enforcement. The first defines enforcement with respect to a specific target. This models scenarios in which the designer knows where the monitor is being installed (e.g., installing a system call interposition monitor to a specific version of a Linux kernel). The second one defines enforcement independently of the target. This models scenarios in which the monitor designer might not know apriori the targets to which the monitor will be applied (e.g., when designing a system call interposition that enforces policies independently of the underlying kernel).

2.4.1 Enforcement

In Section 2.3 we showed how monitoring can be modeled by renaming a target T so that its security-relevant actions can be observed by a monitor $\mathcal{M}$ and by hiding actions that represent communication unobservable outside of the monitored target. We now define enforcement formally as a relation between the behaviors, i.e., sequences of actions, allowed by the policy[7] and the behaviors exhibited by the monitored target.

Definition 1. *(Target-specific enforcement) Given a policy $\mathcal{P}$, a target T, and a monitor $\mathcal{M}$ we say that $\mathcal{P}$ is* T-enforceable$_{=}$ *on T by $\mathcal{M}$ if and only if there exists a module $\hat{P} \in \mathcal{P}$, a renaming function* rename*, and a hiding function* hide$_{\Phi}$ *for some set of actions $\Phi \subseteq$ Output$(\mathcal{M}) \cup$ Output$($rename$(T))$ such that:*

$$(\mathit{scheds}(\mathit{hide}_{\Phi}((\mathcal{M} \times \mathit{rename}(T)))) | \mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P}).$$

Here, $\mathit{hide}_{\Phi}((\mathcal{M} \times \mathit{rename}(T)))$ is the monitored target: the target T is renamed so that its security-relevant actions can be observed by the monitor $\mathcal{M}$; hide is applied to their composition

[7] In this thesis we assume that for all policies $\mathcal{P}$ and for each module $\hat{P}$ in $\mathcal{P}$, there exists an I/O automaton A such that $\mathit{Sig}(A) = \mathit{Sig}(\hat{P})$ and $\mathit{scheds}(\hat{P}) \subseteq \mathit{scheds}(A)$.

to prevent communication between the monitor and the target from leaking outside the composition[8]. If a target does not need renaming, rename can be the identity function; if we do not care about hiding all communication, the hiding function can apply to only some actions. For example, suppose the monitored target from our running example (node with dotted lines in Fig. 2.2b) is composed with an additional monitor that logs system-call requests and responses. We would then keep the actions for system-call requests and responses visible to the logging monitor by not hiding them in the initial monitored target.

Def. 1 binds the enforcement of a policy by a monitor to a specific target. We refer to this type of enforcement as *target-specific enforcement* and to the corresponding monitor as a target-specific monitor. However, some monitors may be able to enforce a property on any target. One such example is an interposition mechanism that operates independently of the target kernel's version or type (e.g., a single monitor binary that can be installed in both Windows and Linux). We call this type of enforcement *generalized enforcement*, and the corresponding monitor a generalized monitor[9]. More formally:

Definition 2. *(Generalized enforcement) Given a policy $\mathcal{P}$ and a monitor $\mathcal{M}$ we say that $\mathcal{P}$ is* enforceable$_=$ *by $\mathcal{M}$ if and only if for all targets T there exists a module $\hat{P} \in \mathcal{P}$, a renaming function* rename*, such that:*

$$(\mathit{scheds}((\mathcal{M} \times \mathit{rename}(T)))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P}).$$

Different instances of Def. 1 and Def. 2 can be obtained by replacing schedules with traces (trace enforcement), fair schedules, or fair traces (fair enforcement).

Alternative definitions of enforcement. Previous definitions of enforcement are based on the notions of *soundness* and *transparency* [29, 30]. Soundness describes the property that the mon-

[8]Since Def. 1 reasons about schedules (i.e., internal actions as well as input and output), hide$_\Phi$ is redundant. We include it in this definition to expose the re-writing process that needs to happen for run-time enforcement in practical scenarios, but we will omit it in the rest of the thesis.

[9]Monitors of previous models, such as [27] and [30], are generalized monitors.

itor will only perform valid actions, i.e., it is required that the monitor's behavior (modeled as a set of traces) is a subset of the behaviors that are allowed by the policy. Defining enforcement as soundness, i.e., as a subset relation, can lead to some trivial cases: monitors could correctly (i.e., soundly) enforce policies by simply doing nothing. Thus, to exclude such trivial cases, and capture practical monitors that do not inhibit the correct behavior of the target, the requirement of *transparency* was introduced: if the target wants to exhibit a correct sequence, then the monitor is forced to output it.

All definitions of soundness and transparency that have been introduced so far are within frameworks where policies reason only about the target's behavior [27, 30]. Thus, in these frameworks a policy $\mathfrak{P}$ is a predicate over sequences of actions that the target might exhibit (i.e., a subset of the sequences of actions that a target can exhibit). In such frameworks one explicitly states transparency in the definition of enforcement by stating that if a target's behavior s is allowed by a policy $\mathfrak{P}$ (i.e., if $s \in \mathfrak{P}$) then the monitor exhibits s.

In our framework, we take a more general view by allowing policies to describe how monitors are integrated with targets, and how monitors are allowed to react to target's requests. Thus, enforcement is now implicit in the definition of a policy (i.e., in the sequences of actions that the policy allows). The latter view has also been adopted by Mandatory Results Automata (MRA) [59]. As discussed there, in frameworks that allow more expressive policies, such as ours, transparency does not need to be stated explicitly as a requirement in the definition of enforcement [59]; transparency can be defined as a specific type of input/output relation (within the formal syntax of the framework).

Although Def. 1 and Def. 2 may seem too restrictive using an equality relation instead of a subset relation, we have found that when dealing with such more expressive policies, equality captures better the intended notion of enforcement[10]. For instance, if we encode soundness and

[10]Since in this chapter we focus only on the equality relation, we will drop the "$=$" symbol from the use of

transparency in our framework, and use a subset relation to define enforcement, then the intended semantics of enforcement are not correctly captured. Next, we illustrate how transparency can be encoded in our framework and how a subset relation can break the intended semantics of enforcement (defined through soundness and transparency).

From transparent enforcement to transparent policies. The main idea behind encoding transparency in our framework is the following: for every behavior t that is allowed by $\mathcal{P}_T$, we construct a schedule that belongs to $\mathcal{P}$ such that the renamed target exhibits (the renamed) t, and the monitor exhibits t. More specifically, given a monitor M, a renaming function *ren*, a target T, and a policy $\mathcal{P}_T$ with $\mathcal{P}_T = \mathit{Sig}(T)$, indicating the allowed behaviors of the target (i.e., policies of previous frameworks) we say that a module $\hat{P}$ of a policy $\mathcal{P}$ describing the monitored target $(M \times \mathit{ren}(T))$ is *transparent* if and only if for all $t \in \mathcal{P}_T$ there exists a schedule $s \in \hat{P}$ such that $s \in (\mathit{ren}(t) \mid\mid t)$ (note that $\mid\mid$ is the interleaving operator defined in Section 2.2).

Note, that the above construction does not tell us anything about what additional schedules $\hat{P}$ contains. This is because our framework has have multiple ways to express what it means to transparently enforce a policy. A question that illustrates this point is whether $\hat{P}$ should contain the schedule, $s' = \mathit{ren}(t)$, the schedule that the monitored target is exhibiting. Under the view of old frameworks this is fine, since $\hat{P}$ considers the target's behavior t to be valid. However, one might say that s' should not be part of $\hat{P}$ since it contradicts the intended semantics: $s' \in \hat{P}$ states that the monitor can react to $\mathit{ren}(t)$ by doing nothing, whereas $s \in \hat{P}$ states that the monitor can react by exhibiting t. However, only the latter describes transparency.

This distinction, and choice of semantics, is strongly related to the fairness assumptions that we make in our model[11]. If we assume fairness we should not include s' in $\hat{P}$. But, if we don't, then we include s' as long as s belongs in $\hat{P}$: it is ok for the monitor to do less due to external

target-specific and generalized enforcement.

[11] A concrete example of fair and unfair schedules is presented at the end of Section 2.3.1

reasons, as long as the monitor wants to do the right thing.

We formalize these two different approaches through the notions of *weak* and *strong transparency*.

Definition 3. *(Weakly transparent module) Given a monitored target* $(M \times ren(T))$*, and a schedule module* $\hat{P}_T$ *with* $Sig(\hat{P}_T) = Sig(T)$*, a schedule module* $\hat{P}$*, with* $Sig(\hat{P}) = Sig(M \times ren(T))$*, is* weakly transparent *for* $(M \times ren(T))$ *w.r.t.* $\hat{P}_T$ *if and only if*

$$\forall t \in \hat{P}_T : \exists s \in scheds(\hat{P}) : ren^{-1}(s|\mathsf{range}(ren)) = (s|\mathsf{dom}(ren)) = t.$$

Def. 3 states that a module is weakly transparent, w.r.t. some policy on the target, if and only if for every schedule that belongs to these target's behaviors there is some schedule in the module such that if we take the subsequence that consists of actions of the monitored target and we reverse the renaming then we will get another subsequence of that schedule that is exhibited by the monitor.

Definition 4. *(Strongly transparent module) Given a monitored target* $(M \times ren(T))$*, and a schedule module* $\hat{P}_T$ *with* $Sig(\hat{P}_T) = Sig(T)$*, a schedule module* $\hat{P}$*, with* $Sig(\hat{P}) = Sig(M \times ren(T))$*, is* strongly transparent *for* $(M \times ren(T))$ *w.r.t.* $\hat{P}_T$ *if and only if*

$$\forall t \in \hat{P}_T : \forall s \in scheds(\hat{P}) : \Big(ren^{-1}(s|\mathsf{range}(ren)) = t \Rightarrow (s|\mathsf{dom}(ren)) = t \Big).$$

Def. 4 states that a module is strongly transparent, w.r.t. some policy on the target, if and only if for every schedule t that belongs to the acceptable target's behaviors and every schedule s that the monitored target exhibits the following holds: if the sequence of actions that the renamed target being monitored tries to execute and appear in s is equal to the renamed t, then the monitor exhibits t as well. The correspondence is as the one in weak transparency: for every schedule in the module if we take the subsequence that consists of actions of the monitored target and we reverse the renaming then we will get another subsequence of that schedule that is exhibited by the monitor (which is an acceptable behavior of the target).

Def. 3 and 4 defined transparency as a specific type of policy $\mathcal{P}$ over a monitored target given

a policy $\mathcal{P}_T$ over the original target (that is now monitored). However, we did not relate $\mathcal{P}_T$ and $\mathcal{P}$ w.r.t. soundness. For example, if $\mathcal{P}_T$ does not contain a schedule s, and thus disallows it, we did not require from $\mathcal{P}$ to also exclude schedules in which the monitor exhibits s. The next definition shows how to achieve this goal, assuming monitors that completely mediate target's security relevant actions:

Definition 5. *(Sound module) Given a monitored target* $(M \times ren(T))$*, and a schedule module* $\hat{P}_T$ *with* $Sig(\hat{P}_T) = Sig(T)$*, a schedule module* $\hat{P}$*, with* $Sig(\hat{P}) = Sig(M \times ren(T))$ *and* $Sig(T) \subseteq Sig(M)$*, is* sound *for* $(M \times ren(T))$ *w.r.t.* $\hat{P}_T$ *if and only if:*

$$\forall t \notin \hat{P}_T : \nexists s \in scheds(\hat{P}) : (s|\mathsf{dom}(ren)) = t.$$

Similar definitions can be expressed for partially-mediating monitors, but we do not pursue them here further. Using Def. 5, 3 (or, 4), and the construction principles in Section 2.3.5, one can fully embed previous frameworks (e.g., definitions of enforcement and edit automata [30]) in ours.

2.4.2 Comparing Enforcement Definitions

As a first example of meta-theoretic analysis in our framework, we compare the two definitions of enforcement from Section 2.4.1, target-specific enforcement and generalized enforcement. One might expect target-specific monitors to have an advantage in enforcement. If we have a monitor that enforces a policy for any target (i.e., a generalized monitor) then that monitor also enforces the policy for some target T. However, a monitor that is "customized" for enforcing a policy on a specific target (e.g., Linux) might not be able to enforce the policy on every possible target (e.g., Windows).

Proposition 2.4.1. *Given a monitor* $\mathcal{M}$ *then:*

1. *if a policy* $\mathcal{P}$ *is* enforceable *by* $\mathcal{M}$*, then for any target* T*,* $\mathcal{P}$ *is* T-enforceable *on* T *by* $\mathcal{M}$*,*

and

2. *there exists a policy $\mathcal{P}$ and a target T such that $\mathcal{P}$ is* T-enforceable *on T by $\mathcal{M}$ and $\mathcal{P}$ is not* enforceable *by $\mathcal{M}$.*

Proof. The proof idea for (1) is to use the module $\hat{P} \in \mathcal{P}$ and renaming function with which M generally enforces $\mathcal{P}$ and we apply them to any target T that we are given. Intuitively, the universal quantification over targets in the right hand side of the implication is internalized on the left hand side in the definition of generalized enforcement.

The idea for (2) is to choose a policy $\mathcal{P}$ that contains only one module $\hat{P}$. $\hat{P}$ has as a signature all possible actions and contains only the schedules that the monitor M can exhibit. Next we choose as T the trivial automaton that exhibits no behavior. Clearly M specifically enforces $\mathcal{P}$ on T. But, for any target that exhibits some internal actions, since these actions are not part of $\hat{P}$, M cannot generally enforce $\mathcal{P}$.

More specifically, for *(1)*, we assume that we have an arbitrary $\mathcal{P}$ that is enforceable by some $\mathcal{M}$. By Def. 2, this means that:

for all targets T there exists a module $\hat{P} \in \mathcal{P}$, a renaming function rename, such that $(\mathit{scheds}((\mathcal{M} \times \mathit{rename}(T)))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$. **(A)**

We have to show that for all targets T', $\mathcal{P}$ is *T-enforceable* on T' by $\mathcal{M}$, which by Def. 1 means that we have to show that for some arbitrary T' there exists a module $\hat{P}' \in \mathcal{P}$, a renaming function rename', such that $(\mathit{scheds}((\mathcal{M} \times \mathsf{rename}'(T')))|\mathit{acts}(\hat{P}')) = \mathit{scheds}(\hat{P}')$.

By **(A)** we know that there are $\hat{P}$ and rename that correspond to any T, and thus for T'. Use the corresponding choices of $\hat{P}$ and rename for T' and our claim follows from **(A)** immediately.

For *(2)*, we must exhibit a $\mathcal{P}$ and a T such that $\mathcal{P}$ is *T-enforceable* on T by $\mathcal{M}$ and it is not the case that $\mathcal{P}$ is *enforceable* by $\mathcal{M}$.

Let $\mathcal{P} = \{\mathit{scheds}(\mathcal{M}) \cup \{\langle a \rangle \mid a \in \Sigma - \mathit{acts}(\mathcal{M})\}\}$. Also, let T be the trivial automaton, i.e.,

the I/O automaton with the empty set for actions and just a single start state. Thus, *scheds*$(T) = \{ \cdot \}$. It is easy to see that $\mathcal{P}$ is *T-enforceable* enforceable on T by $\mathcal{M}$, i.e., that there exists a module $\hat{P} \in \mathcal{P}$, a renaming function rename, such that (*scheds*$((\mathcal{M} \times$ *rename*$(T)))|$*acts*$(\hat{P})) =$ *scheds*$(\hat{P})$. $\mathcal{P}$ contains only one element, so $\hat{P} =$ *scheds*$(\mathcal{M}) \cup \{\langle a \rangle \mid a \in \Sigma -$ *acts*$(\mathcal{M})\}$ which contains all schedules that $\mathcal{M}$ can produce. Moreover, let rename be the identity function. From Thm. 8 we know that *scheds*$((\mathcal{M} \times$ *rename*$(T)))$ will be the pasting of the schedules of the two components, and since the schedules of the component *rename*(T) is just the empty sequence, *scheds*$((\mathcal{M} \times$ *rename*$(T)))$ = *scheds*$(\mathcal{M})$. So we have to show that *scheds*$(\mathcal{M})|$*acts*$(\hat{P})) =$ *scheds*$(\mathcal{M}) \cup \{\langle a \rangle \mid a \in \Sigma -$ *acts*$(\mathcal{M})\}$, which is trivially true.

To prove the second conjunct of the claim, i.e., that it is not the case that $\mathcal{P}$ is *enforceable* by $\mathcal{M}$, pick any T' that has as a signature only one output action, and produces some finite sequence of repetitions of this action of length greater than 1; i.e., *scheds*$(T') = \{(a;a)^n \mid n \geq 1$ *and* $a \in$ *Output*$(T')\}$. Note that no matter how we rename T', its renamed output actions will still be an action of $\hat{P}$, since we added all actions that are not actions of the monitor ($\Sigma -$ *acts*$(\mathcal{M})$). Using Thm. 8 again, we see that the schedules of the composition will contain schedules of the component rename(T'), which means that there is some sequence $s = (a;a) \in$ *scheds*(T'), where $a \in$ *acts*(rename(T')). But $s \notin$ *scheds*$(\hat{P})$ because $s \notin$ *scheds*$(\mathcal{M})$ (since $\mathcal{M}$ and T' have disjoint sets of output actions by definition of composition of I/O automata), and $s \notin \{\langle a \rangle \mid a \in \Sigma -$ *acts*$(\mathcal{M})\}$ since s has length > 1. This concludes the proof of our claim. □

Prop. 2.4.1 compares the two definitions of enforcement (Def. 1 and Def. 2) with respect to the same monitor and shows that our definitions capture the intuitive notions of enforcement; i.e., a monitor that enforces a policy without being tailored for a specific target should enforce the policy on any target, while the inverse should not be true in general.

However, we can get a deeper insight when trying to characterize the two definitions of en-

forcement in general independently of a specific monitor. Surprisingly, in such a comparison the two definitions turn out to be equivalent.

Theorem 2.4.1. *Given a policy $\mathcal{P}$ and a target T, there exist monitors $\mathcal{M}$ and $\mathcal{M}'$ such that $\mathcal{P}$ is* T-enforceable *on T by $\mathcal{M}$ if and only if $\mathcal{P}$ is* enforceable *by $\mathcal{M}'$.*

Proof. The proof idea is the following. For the left direction use M' as M. For the right direction, use as M' the monitored target $M \times \mathsf{rename}(T)$ (which we know that exhibits behaviors that belong to $\mathcal{P}$) and $\alpha-$rename any target that we try to enforce the policy on. More specifically:

($\Rightarrow$ *direction*) We assume that we are given a policy $\mathcal{P}$ and a target T such that $\mathcal{P}$ is *T-enforceable* on T by some monitor $\mathcal{M}$. That is, we assume that there exists a module $\hat{P} \in \mathcal{P}$, a renaming function rename, and a hiding function hide for some set of actions Φ such that $(\mathit{scheds}(\mathit{hide}_\Phi((\mathcal{M} \times \mathit{rename}(T))))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$.

We have to show that $\mathcal{P}$ is enforceable by some monitor $\mathcal{M}'$, or, by definition, that there exists monitor $\mathcal{M}'$ such that for all targets T' there exists a module $\hat{P}' \in \mathcal{P}$, a renaming function rename', and a hiding function hide' such that $(\mathit{scheds}(\mathit{hide}_\Phi((\mathcal{M}' \times \mathit{rename}(T'))))|\mathit{acts}(\hat{P}')) = \mathit{scheds}(\hat{P}')$.

Let:

1. $\mathcal{M}' = \mathit{hide}_\Phi((\mathcal{M} \times \mathit{rename}(T)))$,
2. $\hat{P}' = \hat{P}$,
3. rename' be a function that maps a to a' where $a \in \mathit{acts}(T'), a' \notin \mathit{acts}(\hat{P})$,
4. $\mathsf{hide}' = \mathsf{hide}_\emptyset$.

Now it is easy to see that:

$(\mathit{scheds}(\mathit{hide}_\Phi((\mathcal{M}' \times \mathit{rename}(T'))))|\mathit{acts}(\hat{P}')) = \mathit{scheds}(\hat{P}')$

$\Leftrightarrow (\mathit{scheds}(\mathit{hide}_\emptyset((\mathit{hide}_\Phi((\mathcal{M} \times \mathit{rename}(T))) \times \mathit{rename}(T'))))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$ (by substitution)

$\Leftrightarrow$ ($\mathit{scheds}((\mathit{hide}_\Phi((\mathcal{M} \times \mathit{rename}(T))) \times \mathit{rename}(T')))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$ (by definition of hiding and the fact that $\Phi = \emptyset$)

$\Leftrightarrow$ ($\mathit{scheds}(\mathit{hide}_\Phi((\mathcal{M} \times \mathit{rename}(T))))|act\hat{sP}) \times (\mathit{scheds}(\mathit{rename}(T'))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$ (by Theorems 5 and 7 in App. A)

$\Leftrightarrow$ ($\mathit{scheds}(\mathit{hide}_\Phi((\mathcal{M} \times \mathit{rename}(T))))|act\hat{sP}) \times \cdot = \mathit{scheds}(\hat{P})$ (by definition of rename' and operator |)

$\Leftrightarrow$ ($\mathit{scheds}(\mathit{hide}_\Phi((\mathcal{M} \times \mathit{rename}(T))))|act\hat{sP}) = \mathit{scheds}(\hat{P})$ (by Theorem 7 in App. A)

Note that the last line is true from our assumption, so we are done.

($\Leftarrow$ *direction*) We assume that we are given a policy $\mathcal{P}$ and a target T. Moreover we assume that $\mathcal{P}$ is enforceable by some monitor $\mathcal{M}'$. That is, by definition, we assume that for all targets T there exists a module $\hat{P} \in \mathcal{P}$, a renaming function rename, and a hiding function hide such that ($\mathit{scheds}(\mathit{hide}_\Phi((\mathcal{M} \times \mathit{rename}(T))))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$

We have to show that $\mathcal{P}$ is *T-enforceable* on T by some monitor $\mathcal{M}$. That is, we have to show that there exists a module $\hat{P}' \in \mathcal{P}$, a renaming function rename, and a hiding function hide for some set of actions Φ such that ($\mathit{scheds}(\mathit{hide}_\Phi((\mathcal{M} \times \mathit{rename}(T))))|\mathit{acts}(\hat{P}')) = \mathit{scheds}(\hat{P}')$.

This is trivially true, since we can use the module, renaming function, hiding function, and monitor from our assumptions. Since the subset relationship is satisfied for every target, it is also trivially satisfied by T.

□

The left direction of the theorem is straightforward: any generalized monitor can be used as a target-specific monitor. The right direction is more interesting since it suggests, perhaps surprisingly, that it is possible to construct a generalized monitor from a target-specific one. More specifically, once we have a monitor that enforces a policy on a specific target, we can use this *monitored target* as the basis for a monitor on any other target. In that case, the only security-relevant behavior of the system would be exhibited by the monitor (formally, every action in

every other target would be renamed to become security irrelevant). For example, suppose we have different versions of a specific application installed on each of our machines. If we find a patch (i.e., monitor) for one version, then Thm. 2.4.1 implies that instead of finding patches for all other versions, we can simply distribute the patched version (i.e., monitored target) to all machines and modify the existing applications on those machines so that their behavior is ignored. This approach might be relevant when reinstalling the patched version of the application on top of other versions is more cost-efficient than finding patches for every other version. Thm. 2.4.1 also implies that if we can include enough of a target's functionality in the monitor so that the policy is enforced, then this monitor suffices to enforce the policy on any possible target.

Thm. 2.4.1 holds because Def. 1 and 2 place no restrictions on the renaming functions or how a monitor is integrated with a target. In practice, the interactions between the monitor and the target may, or should, be more constrained. For instance, in Section 2.1 we discussed that in interposition-based mechanisms care must be taken when replicating kernel functionality to the monitor (in fact it better be avoided). Thus, one might argue that it would be more natural to have the only-if direction of the theorem fail, since it erases the distinction between target-specific and generalized enforcement.

To erase the distinction between these two types of enforcement we introduce the notion of *maximal enforcement* which restricts some elements in the definitions of target-specific and generalized enforcement. Intuitively, when talking about a monitor *maximally enforcing* a policy on a target, we require the following restriction: when picking a module of the policy to compare our monitored system with, we must pick one that maximally matches the signature of the monitor, the target, and the range of the renaming function. The module of the policy that we pick has to reason about how the monitor and the target are integrated together. Thus, we make explicit the semantics of $\alpha-$renaming: we cannot make security-relevant actions security-irrelevant by $\alpha-$renaming them to actions that our outside the signature of the module. This is to ensure that

we do not choose modules that do not care about the behavior of the original target, and thus allow everything the target wants to do. For example, assume we have a policy that reasons about file operations and networking events. Moreover, assume the policy has two modules: one reasons about file operations and the other reasons about networking events. Both modules allow the empty sequence of actions; i.e., they allow scenarios in which the target does nothing. With maximal enforcement, to check whether the policy is enforced by a firewall on the networking interface of a target, we compare the behavior of the monitored target with the schedules of the network module, and not the file module. Thus, if the target tries to send a packet to a blacklisted address, and the firewall does not mediate that communication, for example due to insufficient rights, then the firewall cannot enforce that policy. However, had we used the basic definition of enforcement, then one could argue that the policy is enforceable since the firewall trivially enforces it using the file module: no file operations are exhibited by the network interface and thus nothing bad can happen.

Def. 6 and 7 formally express the above constraints.

Definition 6. *(Target-specific Maximal enforcement) Given a policy $\mathcal{P}$, a monitor M, a target T, and a set of renaming functions $\mathcal{R}$ we say that $\mathcal{P}$ is* T-max-enforceable *on T by M using $\mathcal{R}$ if and only if there exist $ren \in \mathcal{R}$ and $\hat{P} \in \mathcal{P}$ such that:*

1. $Sig(\hat{P}) = Sig(M) \cup Sig(T) \cup range(ren)$, *and*
2. $\big(scheds(M \times ren(T)) \mid acts(\hat{P})\big) = scheds(\hat{P})$.

Definition 7. *(Generalized Maximal enforcement) Given a policy $\mathcal{P}$, a monitor M, a set of targets $\mathcal{T}$, and a set of renaming functions $\mathcal{R}$ we say that $\mathcal{P}$ is* max-enforceable *on $\mathcal{T}$ by M using $\mathcal{R}$ if and only if $\forall T \in \mathcal{T}$ there exist $ren \in \mathcal{R}$ and $\hat{P} \in \mathcal{P}$ such that:*

1. $Sig(\hat{P}) = Sig(M) \cup Sig(T) \cup range(ren)$, *and*
2. $\big(scheds(M \times ren(T)) \mid acts(\hat{P})\big) = scheds(\hat{P})$.

Using the definition of maximal enforcement we can show that the equivalence between general and specific enforcement is no longer true[12]:

Theorem 2.4.2. *There exist a policy $\mathcal{P}$, a set of renaming functions $\mathcal{R}$, a set of targets $\mathcal{T}$, and a target $T \in \mathcal{T}$ such that $\mathcal{P}$ is* T-max-enforceable *on T by some monitor $\mathcal{M}$ using $\mathcal{R}$, but $\mathcal{P}$ is not* max-enforceable *on $\mathcal{T}$ by any monitor M' using $\mathcal{R}$.*

Proof. The idea of the proof is to construct a policy $\mathcal{P}$ with two elements $\hat{P}_1$ and $\hat{P}_2$ that describe two different targets, T_1 and T_2. But although $\hat{P}_1$ is enforceable, $\hat{P}_2$ is not (we construct a non-enforceable policy by applying Thm. 2.5.1). Thus, by definition of maximal enforcement when trying to enforce $\mathcal{P}$ on T_1 we have to pick $\hat{P}_1$ which is enforceable, and thus specifically maximally soundly enforceable. Dually, when trying to enforce $\mathcal{P}$ on T_2 we have to pick $\hat{P}_2$ which is not enforceable, and thus $\mathcal{P}$ is not generally maximally soundly enforceable.

More specifically, given an I/O automaton A, let $\mathcal{X}(A) = \{s \mid s \text{ contains only internal actions of } A\}$.

Let T_1 and T_2 be targets with $\mathit{Sig}(T_1) \neq \mathit{Sig}(T_2)$, $\mathit{scheds}(T_1) \neq \emptyset$, $\mathit{scheds}(T_2) \neq \emptyset$, and $\mathcal{X}(T_2) \neq \emptyset$.

Let $\mathcal{T} = \{T_1, T_2\}$, $\mathcal{R} = \{\mathit{id}\}$, i.e., the only renaming function allowed is the identity function.

Now we construct the policy $\mathcal{P} = \{\hat{P}_1, \hat{P}_2\}$, where $\hat{P}_1 = \langle \mathit{Sig}(T_1), \mathit{scheds}(T_1)\rangle$, and $\hat{P}_2 = \langle \mathit{Sig}(T_2), \mathit{scheds}(T_2) - \mathcal{X}(T_2)\rangle$, i.e., $\hat{P}_2$ disallows any schedule of T_2 that contains only internal actions of T_2.

Now, pick $T_1 \in \mathcal{T}$. It is easy to see that there exists a monitor M, the trivial monitor with the empty signature that does nothing, such that (1) $\mathit{Sig}(\hat{P}) = \mathit{Sig}(T_1) \cup \mathit{Sig}(M) \cup \mathit{range}(\mathit{id})$, and (2) $\mathit{scheds}(M \times \mathit{id}(T_1)) = \mathit{scheds}(\hat{P}_1)$. By simple syntactic manipulations we get (1) $\mathit{Sig}(T_1) = \mathit{Sig}(T_1)$, and (2) $\mathit{scheds}(T_1) = \mathit{scheds}(T_1)$, which is trivially true.

[12]Note that Thm. 2.4.1 is implicitly universally quantified over all (sets of) renaming functions and targets. Thus, Thm. 2.4.2 is indeed the negation of Thm. 2.4.1.

Now, it suffices to show that there is no monitor M' such that $\mathcal{P}$ is generally maximally soundly enforceable by M'. When trying to see whether $\mathcal{P}$ is enforceable for T_2, the first constraint of the definition of general maximal enforcement forces us to choose $\hat{P}_2$: it is the only module that matches the signature or T_2.

But, by construction, $\hat{P}_2$ disallows any schedule that contains internal actions of T_2, and T_2 can exhibit such schedules. Moreover, since our only renaming function available is the identity, there is no monitor M' that can prohibit these internal actions from happening: I/O automata composition do not allow for one component automaton to control the local actions of any other component. Thus, for any M', the monitored target $M' \times id(T_2)$ will exhibit schedules that belong to $\mathcal{X}(T_2)$ and thus the subset relation $scheds(M' \times id(T_2)) = scheds(\hat{P}_2)$ does not hold. Thus $\mathcal{P}$ is not generally maximally soundly enforceable, and this completes the proof of the theorem.

□

Thm. 2.4.2 does not allow for general and specific enforcement to imply each other. Thus, the definition of maximal enforcement restricts the generality of the framework as was introduced in the previous subsections. Maximal enforcement can be useful in two types of scenarios: (1) a designer wants to reason about how the monitor and the target are integrated and thus the policy needs to reason about the renaming function (e.g., $\alpha-$rename is not allowed), and (2) scenarios in which the monitor cannot implement (i.e., substitute for) every behavior of the target. For example, if we have a policy that reasons about cryptographic operations and file operations, we can build a monitor that faithfully reproduces the file operations that the target can perform, but not a monitor that reproduces the cryptographic operations of the target (e.g., due to lack of access to private cryptographic keys). In this case maximal enforcement can help us to reason about the practical limitation of monitors when discussing about the enforceability of policies. On the other hand, the patching example after Thm. 2.4.1 illustrates a practical scenario where the most

general framework is appropriate for modeling and reasoning about enforcement. Since our goal is to introduce a framework that is general enough to accommodate as many practical scenarios as possible (even seemingly degenerate ones), we rely on the monitor designer to impose appropriate restrictions on monitors to better reflect on monitors under scrutiny.

2.5 Generally Enforceable Policies

The definitions and abstractions described thus far enable rigorous, detailed analyses of practical monitored systems. They also allow meta-theoretic reasoning that furthers our understanding of general limitations of practical monitors that fit this model. In this section we begin our analysis by focusing on general enforcement and derive several such meta-theoretic results. In the next section, we further our analysis by focusing on target-specific enforcement.

2.5.1 Auxiliary Definitions

I/O automata are input enabled, which as discussed in Section 2.2 means that all input actions of an automaton A are enabled at all states of A. Several arguments can be made in favor of or against input-enabledness. For example, one might argue that input-enabledness may lead to better design of systems because one has to consider all inputs that may be received from the environment [1]. On the other hand, this constraint might be too restrictive for practical systems [66].

In our context, we believe that input-enabledness is a useful characteristic, since run-time monitors are by nature input-enabled systems: the monitor may receive input at any time both from the target and from the environment (e.g., keyboard or network). However, a monitor modeled as an input-enabled automaton can enforce only those policies that allow the arrival of inputs at any point during execution. This is reasonable; a policy that prohibits certain inputs cannot be enforced by a monitor that cannot control those inputs. We later combine this and other constraints to describe the upper bound of enforceability in our setting.

We say that a module, or policy, is *input forgiving* (respectively, *internal* and *output* forgiving) if and only if it allows the empty sequence and allows each valid sequence to be extended

to another valid sequence by appending any, possibly infinite, sequence of input actions (respectively, internal and output actions).

Definition 8. *A schedule module $\hat{P}$ is* input forgiving *if and only if:*

(1) $\cdot \in \mathit{scheds}(\hat{P})$*; and*

(2) $\forall s_1 \in \mathit{scheds}(\hat{P}) : \forall s_2 \preceq s_1 : \forall s_3 \in ((\mathit{Input}(\hat{P})))^{\infty} : (s_2; s_3) \in \mathit{scheds}(\hat{P})$.

I/O automata's definition of executions allows computation to stop at any point. Thus, the behavior of an I/O automaton is *safe*; any prefix of a schedule exhibited by an automaton is also a schedule of that automaton, and all successive extension of schedules are limit closed [1]:

Definition 9. *A schedule module $\hat{P}$ is a* safety module *if and only if:*

1. $\mathit{scheds}(\hat{P}) \neq \emptyset$*,*
2. $\mathit{scheds}(\hat{P})$ *is* prefix closed*; i.e., if* $s \in \mathit{scheds}(\hat{P})$ *and* $s' \preceq s$*, then* $s' \in \mathit{scheds}(\hat{P})$*, and*
3. $\mathit{scheds}(\hat{P})$ *is* limit closed*; i.e., if* s_1*,* s_2*, ... is an infinite sequence of finite sequences in* $\mathit{scheds}(\hat{P})$*, and for each* i*,* s_i *is a prefix of* s_{i+1}*, then the unique schedule* s *that is the limit of* s_i *under the successive extension ordering is also in* $\mathit{scheds}(\hat{P})$.

These characteristics are unsurprising from the standpoint of models for distributed computation, but describe practically relevant details that are typically absent from models of run-time enforcement. Our model, instead of making assumptions that might not hold in every practical scenario (e.g., that all actions can be mediated) takes a more nuanced view, which admits that there are aspects of enforcement outside the monitor's control, such as security-relevant actions that the monitor cannot observe or mediate, or the existence of scheduling strategies that might not favor the monitor. When formally analyzing the policies enforceable by monitors, as in the next section, the above definitions help us make precise these assumptions.

2.5.2 Upper Bounds of Enforceable Policies

Another constraint that affects the upper bounds of enforceability specific to monitoring is that a monitored system cannot always ignore *all* behaviors of the target application. Some realistic monitors decide what input actions the application sees, but otherwise do not interfere with the application's behavior—firewalls belong to this class of monitors. In such cases, a monitor can soundly enforce a policy only if the policy allows all the behaviors that the target can exhibit even if it receives no input. We call these policies *quiescent forgiving* (recall the definition of a quiescent state from Section 2.2). Modules contained in such policies are also called quiescent forgiving. This definition captures one type of limitation that was understood to be present in run-time monitoring, but that was not formally expressed. Quiescent forgiving modules can be defined more formally as follows:

Definition 10. *A schedule module* $\hat{P}$ *is* quiescent forgiving *for some* T *if and only if:*

$\forall e \in \mathit{execs}(T)$ *such that* $e = q_0, a_1, \ldots, q_n$:

$$\Big(q_n \in \mathit{quiescent}(T) \wedge \big(\forall i \in \mathbf{N} : 0 \leq i < n : q_i \notin \mathit{quiescent}(T)\big)\Big) \Rightarrow$$

$$\big(\mathit{sched}(e)|\mathit{acts}(\hat{P})\big) \in \mathit{scheds}(\hat{P}) \wedge (\forall i \in \mathbf{N} : 0 \leq i < n : (\mathit{sched}(q_0, \ldots, q_i) \mid \mathit{acts}(\hat{P})) \in \mathit{scheds}(\hat{P})).$$

The following theorem formalizes an upper bound: a policy that is not quiescent forgiving, input forgiving, and prefix closed cannot be (precisely) enforced by any monitor.

Theorem 2.5.1. *Given a policy* $\mathcal{P}$*, a schedule module* $\hat{P} \in \mathcal{P}$*, a target* T*, and a renaming function* rename*, if there exists a monitor* $\mathcal{M}$ *such that*

$$(\mathit{scheds}(\mathcal{M} \times \mathit{rename}(T))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P}),$$

then $\hat{P}$ *is input forgiving, safety, and quiescent forgiving for* $\mathit{rename}(T)$.

Proof. The proof is by contradiction. The main ideas are the following: if $\hat{P}$ does not allow certain inputs then it is not enforceable because the monitored target is an I/O automaton, and I/O

automata cannot block inputs: they always appear in the schedules of the monitored target. The same argument holds for safety: I/O automata's schedules are prefix closed (and limit closed), thus $\hat{P}$ has to be as well. Finally, since the monitor does not control the local actions that the target will execute at the beginning of its execution, if the property forbids some of these schedules then the monitor cannot enforce the policy.

More specifically, we fix a policy $\mathcal{P}$, a module $\hat{P}$, a hiding function $hide_{\Phi}()$, and a renaming function $rename()$, and we assume that there exists a monitor $\mathcal{M}$ such that $\Big(scheds(hide_{\Phi}(\mathcal{M} \times rename(T)))|acts(\hat{P}) \subseteq scheds(\hat{P})\Big)$. We have to show that $\hat{P}$ is input forgiving, prefix closed, and quiescent forgiving for $rename(T)$.

For the sake of contradiction, assume that $\hat{P}$ is not input forgiving, or not prefix closed, or not quiescent forgiving for $rename(T)$.

Case: $\hat{P}$ is not input forgiving:

Since $\hat{P}$ is not input forgiving, then either $\cdot \notin scheds(\hat{P})$ or there exists an $s_1 \in scheds(\hat{P})$, a finite prefix s_2 of s_1, and some sequence of input actions s_3 such that $(s_2; s_3) \notin scheds(\hat{P})$. If we assume the first case of $\cdot \notin scheds(\hat{P})$ we derive a contradiction since the empty sequence belongs to the schedules of any I/O automaton by definition of executions and schedules of I/O automata. If we assume the latter case, then we know that $s_2 \in scheds(hide_{\Phi}(\mathcal{M} \times rename(T)))$ by assumption. Let q_n be the state that the monitored target is at after executing the last action of s_2. By definition, every state of an I/O automaton is input enabled. Thus q_n is input enabled, which means that $\forall s' \in ((Input(hide_{\Phi}(\mathcal{M} \times rename(T)))))^{\infty}$: $(s_2; s') \in scheds(hide_{\Phi}(\mathcal{M} \times rename(T)))$ (remember we assume no fairness thus it does not matter whether q_n is quiescent or not). But for $s' = s_3$ we get that $(s_2; s_3) \in scheds(hide_{\Phi}(\mathcal{M} \times rename(T)))$ and that $(s_2; s_3) \notin scheds(\hat{P})$ which contradicts our assumption. Thus, in both cases we derived a contradiction, and thus $\hat{P}$ must be input forgiving.

Case: $\hat{P}$ is not safety:

Since $\hat{P}$ is not safety, then there exists some schedule s_1 that belongs to the schedules of $\hat{P}$, but there exists some prefix s_2 of s_1 that does not belong to the schedule of $\hat{P}$, or more formally: $\exists s_1 \in (\Sigma)^\infty : \big(s_1 \in \mathit{scheds}(\hat{P})\big) \wedge \big(\exists s_2 \in (\Sigma)^\star : s_2 \preceq s_1 : s_2 \notin \mathit{scheds}(\hat{P})\big)$.

Without loss of generality, assume s_2 is the longest strict prefix of s_1, i.e., it is the longest prefix of s_1 that does not belong to the schedules of $\hat{P}$, and that all prefixes of s_2 belong to the schedules of $\hat{P}$. If $s_2 = a_1, \ldots, a_{n-1}, a_n$ then let $s_2^- = a_1, \ldots, a_{n-1}$ and $s_2^+ = a_1, \ldots, a_{n-1}, a_n, a_{n+1} \preceq s_1$. We know that $s_2^- \in \mathit{scheds}(\hat{P})$ by assumption, $s_2^+ \in \mathit{scheds}(\hat{P})$ because if it was not in the schedules of the property this would be the longest invalid prefix of s_1 which contradicts our choice of s_2, and thus by assumption they both also belong to the schedules of the monitored target $\mathit{hide}_\Phi(\mathcal{M} \times \mathit{rename}(T))$. Let q_n be the state that the monitored target is after executing a_{n-1}, and q_{n+1} be the state before executing a_{n+1}. In order for the automaton to transition from q_n to q_{n+1} it must execute some a_n. But then $s_2^-; a_n = s_2 \in \mathit{scheds}(\mathit{hide}_\Phi(\mathcal{M} \times \mathit{rename}(T)))$, while we assumed that $s_2 \notin \mathit{scheds}(\hat{P})$. This contradicts our original assumption, and thus $\hat{P}$ must be prefix closed.

Case: $\hat{P}$ is not quiescent forgiving:

Since $\hat{P}$ is not quiescent forgiving for $\mathit{rename}(T)$, then there exists some execution $e = q_0, a_1, \ldots, q_n$ of $\mathit{rename}(T)$ with $q_n \in \mathit{quiescent}(T)$ and $q_i \notin \mathit{quiescent}(T)$ for $0 \leq i < n$ such that either $(\mathit{sched}(e)|\mathit{acts}(\hat{P})) \notin \mathit{scheds}(\hat{P})$ or some prefix t of $(\mathit{sched}(e)|\mathit{acts}(\hat{P}))$ does not belong to the schedules of $\hat{P}$.

By Theorem 7 we know that if $\mathit{sched}(e) \in \mathit{scheds}(\mathit{rename}(T))$, then $\mathit{sched}(e) \in \mathit{scheds}(\mathit{hide}_\Phi(\mathcal{M} \times \mathit{rename}(T)))$. Thus, $(\mathit{sched}(e)|acts\hat{P}) \in \mathit{scheds}(\mathit{hide}_\Phi(\mathcal{M} \times \mathit{rename}(T)))$. But the fact that $(\mathit{sched}(e)|\mathit{acts}(\hat{P})) \notin \mathit{scheds}(\hat{P})$ contradicts our assumption. With the same argument we can show that even if we assume some prefix t of $\mathit{sched}(e)$ we also derive a contradiction. Thus $\hat{P}$ must be quiescent forgiving.

□

Thm. 2.5.1 reveals that monitors, regardless of their editing power, can enforce only safety properties. Thus, in our context, even the equivalent of an edit monitor cannot enforce renewal properties (as opposed to [30]), since when renewal properties are constrained by prefix closure they collapse to safety properties. This is because, as mentioned above, our model of executions allows computation to stop at any point. This is another helpful characteristic of our model; it highlights that on systems in which execution may cease at any moment (e.g., due to a power outage), only safety properties can be enforced.

In practice, monitors typically reproduce at most a subset of a target's functionality. Hence, if a monitor composed with an application is to exhibit the same range of behaviors as the unmonitored application, it will have to consult the target application in order to generate these behaviors. In the system-call interposition example, for instance, the monitor cannot return correct file descriptors without consulting the kernel. Such monitors that regularly consult an application, cannot precisely enforce (with respect to schedules) arbitrary policies even if they are quiescent forgiving, input forgiving, and prefix-closed. This is because an input forwarded by the monitor to an application might cause the application to execute internal or output actions (e.g., a buffer overflow) that are not allowed by the policy and that the monitor cannot prevent, since these are outside of the interface between the monitor and the target.

On the other hand, it is also common for the monitor (or system designer) to have some knowledge about the target, even if it does not have access to its state. This knowledge can be exploited to use simpler-than-expected monitors to enforce of (seemingly) complex policies. Although similar observations have been made before (e.g., program re-writing [29], non-uniformity [30], use of static analysis [67]), our framework can be used to formally extend them, as we demonstrate in Section 2.6.

2.5.3 Lower Bounds of Transparently Enforceable Policies

In Section 2.4.1 we discussed how to encode the notion of transparent enforcement as a specific type, or instantiation, of policy in our framework. Here we take a closer look at the constraints under which such policies are enforceable.

As discussed in the previous section, in our basic framework monitors can only enforce safety policies. Thus, if a monitor needs to exhibit more than one action to (strongly) transparently enforce a policy, there is no guarantee that it will be able to do so. In contrast, previous models (e.g., [30]) assumed that enabled actions of a monitor would always be performed. In our framework, to achieve equivalent results we can either explicitly add similar guarantees about the runs of the system through appropriate fairness constraints [63], or relax the requirements in the definition of transparent enforcement (cf. weak and strong transparency). This is another instance of our framework making explicit the (practical) assumptions and constraints that affect the enforcement of policies.

First, we provide definitions of target-specific and general *fair enforcement*. The following definitions of enforcement compare the schedules of the policy with the fair schedules of the monitored target; i.e., we only care about the final behaviors that the monitored target wants to exhibit and not all steps that it has to take to reach them.

Definition 11. *(Fair target-specific enforcement) Given a policy $\mathcal{P}$, a target T, and a monitor $\mathcal{M}$ we say that $\mathcal{P}$ is* fairly T-enforceable *on T by $\mathcal{M}$ if and only if there exists a module $\hat{P} \in \mathcal{P}$, a renaming function* rename*, and a hiding function* hide *for some set of actions Φ such that* $(\mathit{fairscheds}(\mathit{hide}_{\Phi}((\mathcal{M} \times \mathit{rename}(T))))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$.

Definition 12. *(Fair generalized enforcement) Given a policy $\mathcal{P}$ and a monitor $\mathcal{M}$ we say that $\mathcal{P}$ is* fairly enforceable *by $\mathcal{M}$ if and only if for all targets T there exists a module $\hat{P} \in \mathcal{P}$, a renaming function* rename*, such that* $(\mathit{fairscheds}((\mathcal{M} \times \mathit{rename}(T)))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$.

Using these definitions we can now characterize the policies that are transparently enforceable by monitors, or equivalently, under which constraints monitors can enforce policies that encode transparency.

If a policy $\mathcal{P}$ contains a module $\hat{P}$ that is strongly transparent, then there is no monitor that can precisely enforce $\mathcal{P}$ using $\hat{P}$ without taking into account fairness: the schedules of the monitored target will be prefix closed, whereas the schedules of the policy won't; i.e., the equality relation won't hold. On the other hand, if a monitor fairly T-enforces $policy$ using $\hat{P}$ then the $\hat{P}$ must be be input and quiescent forgiving. It is easy to show that the last statement is non-trivial; i.e., there exists some strongly transparent $\hat{P}$ that is fairly T-enforceable by some monitor M. These observations are formalized in Thm. 2.5.2.

Theorem 2.5.2. *Given a policy $\mathcal{P}$, a schedule module $\hat{P} \in \mathcal{P}$, a target T, a monitor $\mathcal{M}$, a schedule module $\hat{P}_T$, and a renaming function* rename*, such that $\hat{P}$ is strongly transparent for $M \times \mathsf{rename}(T)$ w.r.t. $\hat{P}_T$ then:*

1. *there is no monitor M that T-enforces $\mathcal{P}$ on T using $\hat{P}$, and*
2. *if there exists M that fairly T-enforces $\mathcal{P}$ on T using $\hat{P}$ then $\hat{P}$ is input forgiving, and quiescent forgiving for rename(T).*

Proof. The idea of the proof is as follows: since $\hat{P}$ is strongly transparent then every schedule that belongs to $\hat{P}$ must be of even length (containing the behavior that the target wants to execute and the behavior that the monitor forwards to the environment). But, the monitored target is an I/O automaton which means that its schedules are prefix closed and thus it contains schedules of odd length. Thus, there is no way for the set of schedules of the monitored target to be equal to set of schedules of a strongly transparent $\hat{P}$. On the other hand, the fair schedules of the monitored target can be of even length, and thus the only constraints that $\hat{P}$ must adhere to is input and quiescence forgiveness (for reasons described in Thm. 2.5.1).

More specifically, the proof is by contradiction. First, pick a policy $\mathcal{P}$ with only one non-trivial element $\hat{P}$ and a non-trivial module $\hat{P}_T$, i.e., $\mathit{scheds}(\hat{P}) \neq \emptyset$ and $\mathit{scheds}(\hat{P}_T) \neq \emptyset$. Now, let $\hat{P}$ contain only transparent schedules, i.e., $\forall t \in \hat{P}_T : \exists s \in \mathit{scheds}(\hat{P}) : \mathit{ren}^{-1}(s|\mathsf{range}(\mathit{ren})) = (s|\mathsf{dom}(\mathit{ren})) = t$, and $\hat{P}$ does not contain any other schedules besides the ones specified above.

Proof of (1): Assume there exists monitor M that specifically precisely enforces $\mathcal{P}$ on T using $\hat{P}$. By Thm. 2.5.1, $\hat{P}$ must be prefix closed. But also, by construction, $\forall s \in \mathit{scheds}(\hat{P}) : \exists t \in \hat{P}_T : \mathit{ren}^{-1}(s|\mathsf{range}(\mathit{ren})) = (s|\mathsf{dom}(\mathit{ren})) = t$.

Since $\mathit{scheds}(\hat{P}_T)$ and $\mathit{scheds}(\hat{P})$ are non-empty pick $s \in \mathit{scheds}(\hat{P})$ such that $\exists t \in \hat{P}_T$ such that $\mathit{ren}^{-1}(s|\mathsf{range}(\mathit{ren})) = (s|\mathsf{dom}(\mathit{ren})) = t$. Now since we assumed that $\hat{P}$ is prefix closed, then if $s = a_1; a_2; \ldots; a_n; a_{n+1}$, then $s' = a_1; a_2; \ldots; a_n$ must also belong to $\hat{P}$. But then it is easy to see by a simple counting argument that there is no $t' \in \hat{P}_T$ such that $\mathit{ren}^{-1}(s'|\mathsf{range}(\mathit{ren})) = (s'|\mathsf{dom}(\mathit{ren})) = t'$: s' contains one less action than s, whereas it should contain two less actions for the above equality to hold.

Thus we have a contradiction, and no such M can exist that specifically precisely enforce $\mathcal{P}$ on T using $\hat{P}$.

Proof of (2): Similar to the proof of Thm. 2.5.1 for the cases of input-forgiving and quiescent forgiving, but substituting occurrences of $\mathit{scheds}()$ with $\mathit{fairscheds}()$.

□

The requirement for fairness in order to enforce strongly transparent modules (and policies) is not tight to the definition of precise enforcement (i.e., equality relation between sets of schedules). It is required even when talking about sound enforcement (i.e., subset relation). However, there is a corner case where we can soundly enforce a strongly transparent module without the use of fairness; the monitored target exhibits no behavior at all and the module allows it. This is an important point for the specification of security policies since there might be cases where

transparency is not correctly captured, for example, if a target is not violating the policy but is blocking. In this situation the target does exhibit good behaviors but only if the monitor initiates the computation, then one might argue that we are not transparent since we are prohibiting the target from performing good actions. But, it depends on the specific scenario and semantics of transparency and policies that the designer wants to capture to decide whether this corner case should or should not be allowed. The following proposition formalizes this idea:

Proposition 2.5.1. *Given a policy $\mathcal{P}$, a schedule module $\hat{P} \in \mathcal{P}$ such that $\cdot \notin scheds(\hat{P})$, a target T, a monitor $\mathcal{M}$, a schedule module $\hat{P}_T$ with $scheds(\hat{P}_T) \neq \emptyset$, and a renaming function* rename*, then if $\hat{P}$ is strongly transparent for $M \times \mathsf{rename}(T)$ w.r.t. $\hat{P}_T$ then there is no monitor M that specifically soundly enforces $\mathcal{P}$ on T using $\hat{P}$.*

Proof. As explained in the proof idea of Thm. 2.5.2, a monitored target may exhibit schedules of odd length that can not belong in a strongly transparent module. The only corner case is when the monitored target exhibits no schedules at all, which means that it exhibits the empty schedule (which is of even length). Since the strongly transparent modules of the theorem do not contain the empty schedule, then the policy is not even soundly enforceable.

More specifically, by Def. 4, of strong transparency, it is easy to see that every schedule that belongs to $\hat{P}$ and contains some behavior of the target that belongs to $\hat{P}_T$ must have even length. But as described in the proof of Thm. 2.5.2 since the schedules of the monitored target are prefix closed (i.e., safety), the monitored target will exhibit schedules (of odd length) that do not belong to $\hat{P}$. In Thm. 2.5.2 that was enough to contradict the theorem statement because we were proving precise enforcement, i.e., equality. Here, there is one case that the monitor can soundly enforce the policy (i.e., subset relation): the monitored target does nothing, i.e., it exhibits a schedule

with no actions (which has an even length, i.e., 0). But, by assumption $\cdot \notin \hat{P}$. Thus, this corner case cannot happen under the assumptions of the theorem statement and we conclude that there is no monitor that specifically soundly enforces $\mathcal{P}$ on T using $\hat{P}$.

□

As discussed above, we can transparently enforce a larger class of policies if we allow for a weaker definition of transparency (i.e., weak transparency). A weakly transparent module may range between the two extremes: strongly transparent modules and safety modules. These two bounds are found when we take into account fairness, or the lack of it:

Theorem 2.5.3. *Given a policy $\mathcal{P}$, a schedule module $\hat{P} \in \mathcal{P}$, a target T, a monitor $\mathcal{M}$, a schedule module $\hat{P}_T$, and a renaming function* rename*, such that $\hat{P}$ is strongly transparent for $M \times \mathsf{rename}(T)$ w.r.t. $\hat{P}_T$ then:*

1. *if M T-enforces $\mathcal{P}$ on T using $\hat{P}$, then $\hat{P}$ is input forgiving, safety, and quiescent forgiving for rename(T), and*
2. *if M fairly T-enforces $\mathcal{P}$ on T using $\hat{P}$ then $\hat{P}$ is input forgiving, and quiescent forgiving for rename(T).*

Proof. Derived easily from Thm. 2.5.1, for (1), and 2.5.2, for (2).

□

2.6 Target-specifically Enforceable Policies

As discussed in Section 2.3, the expressiveness of our model allows multiple ways to define monitors (e.g., a truncation monitor) that had a single natural definition in previous models. Due to space limitations, rather than comprehensively analyzing the policies enforceable by specific monitors, as done in previous work [27, 29, 30, 34, 59], we demonstrate how our framework enables formal results that can be exploited by designers of run-time monitors who have knowledge about the target application by using a novel analysis of how some knowledge of the target can compensate (in terms of enforceability) for a narrower monitoring interface (i.e., incomplete mediation).

We begin by showing how partially mediating monitors can be formally defined in our framework. We will first define what it means for a monitored target to be input/output mediating and input mediating. The definitions formalize the constraints on the renaming functions of the monitored target, as they were described in Section 2.3.

Definition 13. *A monitor $\mathcal{M}$ is input/output mediating if and only if for any target T there exists a renaming function* rename *such that:*

1. *Output*$(\mathsf{rename}(T)) \subseteq$ *Input*$(\mathcal{M})$*,*
2. *Input*$(\mathsf{rename}(T)) \subseteq$ *Output*$(\mathcal{M})$*,*
3. *Internal*$(\mathsf{rename}(T)) =$ *Internal*(T)*,*
4. *Output*$(T) \subseteq$ *Output*$(\mathcal{M})$*, and*
5. *Input*$(T) \subseteq$ *Input*$(\mathcal{M})$*.*

Constraints (1-3) force the renaming function to match the interfaces of the target and the monitor (i.e., it does not allow to arbitrarily rename the target interface) and ensure that all security relevant input/output behavior of the target is completely mediated by the monitor. In particular,

constraint (1) ensures that all security relevant outputs will be received by the monitor, while constraint (2) ensures that all the security relevant inputs to the target will come from the monitor. Constraint (3) ensures that the security relevant actions of the target are not renamed so that we can capture the fact that there are actions that are outside the monitor's control. If we could rename them to some internal actions of the monitor, then it would be possible to not exhibit invalid internal actions because the monitor controls its own internal actions. Finally, constraints (4) and (5) ensure that the monitor has the ability to input and output the actions that the original target could.

Similarly to Definition 13 we can define a monitored target to be *input mediating*:

Definition 14. *A monitor $\mathcal{M}$ is input mediating if and only if for any target T there exists a renaming function* rename *such that:*

1. $\mathit{Input}(\mathsf{rename}(T)) \subseteq \mathit{Output}(\mathcal{M})$,
2. $\mathit{Internal}(\mathsf{rename}(T)) = \mathit{Internal}(T)$,
3. $\mathit{Output}(\mathsf{rename}(T)) = \mathit{Output}(T)$, *and*
4. $\mathit{Input}(T) \subseteq \mathit{Input}(\mathcal{M})$.

Constraint (1) ensures that all the security relevant inputs to the target will come from the monitor. Constraints (2) and (3) ensure that the security relevant actions of the target are not renamed so that we can capture the fact that there are actions that are outside the monitor's control, this time including the output actions of the target. Finally, constraint (4) ensures that the monitor has the ability to input all the actions that the original target could.

In Section 2.3 we described two monitoring architectures: one in which the monitor mediates the inputs and the outputs of the target, and another in which it mediates just the inputs. Intuitively, an input/output-mediating monitor should be able to enforce a larger class of policies than an input-mediating one, since the former is able to control (potentially) more security-relevant

actions than the latter (i.e., the outputs of the target). In other words, there exist policies that are enforceable by input/output mediating monitors, but not by input mediating monitors. This can be expressed as follows:

Theorem 2.6.1. *There exists a policy $\mathcal{P}$ that is enforceable by some input/output-mediating $\mathcal{M}_1$ and not enforceable by some input-mediating $\mathcal{M}_2$.*

Proof. The idea of the proof is to construct a policy that prohibits certain (targets') output actions. It is easy to see that there exists a target (i.e., an I/O automaton) that exhibits exactly the actions that the policy disallows. An input/output-mediating monitor can enforce that policy since it mediates the output actions that the (renamed) target wants to execute and if they violate the policy does not forward them to the environment. But an input-mediating monitor cannot (by definition) prohibit the bad output actions from happening, and thus it cannot precisely enforce the policy.

More specifically, take $\mathcal{P} = \{\hat{P}\}$, where $\mathit{acts}(\hat{P}) = \mathit{Output}(\hat{P}) = \bigcup_{i \in I} \mathit{Output}(T_i) \cup \bigcup_{i \in I} \mathsf{rename}_{\mathsf{j} \in \mathsf{J}}(\mathit{Output}(T_i))$, $\mathit{scheds}(\hat{P}) = \{\ \cdot\ \} \cup \{\langle a \rangle \mid a \in \mathit{acts}(\hat{P})\}$, and $\hat{P}$ does not reason about the communication between the monitor and the target where I is the set of all targets, and J the set of all renaming functions (note that in the rest of the proof, for purposes of brevity of presentation, we are assuming that the universes of Input, Output, Internal actions are disjoint, and that renaming functions always map actions to fresh actions that are distinct from the Input, Output, and Internal actions of the targets).

For proving the left conjunct of the theorem statement, we have to prove that there exists an input/output-mediating $\mathcal{M}_1$ such that for all targets T there exists a module $\hat{P} \in \mathcal{P}$, a renaming function rename, such that $(\mathit{scheds}((\mathcal{M}_1 \times \mathit{rename}(T)))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$.

Let $\mathcal{M}_1$ be the input/output-mediating monitor that has as elements of its signature the following sets: $\mathit{Input}(\mathcal{M}_1) = \{\bigcup_{i \in I} \mathit{Input}(T_i)\} \cup \{\bigcup_{i \in I} \mathsf{rename}_{\mathsf{j} \in \mathsf{J}}\ (\mathit{Output}(T_i))\}$, $\mathit{Output}(\mathcal{M}_1) =$

$\{\bigcup_{i\in I} \mathit{Output}(T_i)\} \cup \{\bigcup_{i\in I} \mathsf{rename}_{\mathsf{j}\in\mathsf{J}}(\mathit{Input}(T_i))\}$, $\mathit{Internal}(\mathcal{M}_1) = \emptyset$, where I is the set of all targets, and J the set of all renaming functions.

Moreover, let $\mathit{scheds}(\mathcal{M}_1)$ contain no schedules that include more than one output actions from the subset $\{\bigcup_{i\in I} \mathit{Output}(T_i)\}$, i.e., the monitor does not exhibit any output behavior to the environment that contains more than one action. This is easy to do: just exhibit the first valid output action that the target wants to execute, and suppress all future attempts. Now it is easy to see that for all targets T there exists a module $\hat{P} \in \mathcal{P}$, a renaming function rename, such that $(\mathit{scheds}((\mathcal{M}_1 \times \mathit{rename}(T)))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$: assume otherwise, i.e., there exists a schedule $s \in \mathit{scheds}((\mathcal{M}_1 \times \mathit{rename}(T)))$ that is not an element of $\mathit{scheds}(\hat{P})$. Since $\mathit{scheds}(\hat{P})$ contains all possible sequences of length one that contain the output actions of all targets (and all their possible renamings), the only way for $(s \mid \mathit{acts}(\hat{P}))$ not to be an element of the schedules of $\hat{P}$ is to contain output actions and have length larger than 1. However, this is impossible by (1) construction of the monitor, and (2) assumption that the policy does not reason about the communication between the monitor and the target (by Def. 13, all output actions of the target are mediated by the monitor and thus considered part of their communication).

For proving the right conjunct of the theorem statement, we have to prove that it is not the case that there exists an input-mediating $\mathcal{M}_2$ such that for all targets T there exists a module $\hat{P} \in \mathcal{P}$, a renaming function rename, such that $(\mathit{scheds}((\mathcal{M}_2 \times \mathit{rename}(T)))|\mathit{acts}(\hat{P})) = \mathit{scheds}(\hat{P})$. In other words, we have to prove that for all input-mediating $\mathcal{M}_2$, there exists a target T, such that for all modules $\hat{P} \in \mathcal{P}$, and for all renaming functions rename: $(\mathit{scheds}((\mathcal{M}_2 \times \mathit{rename}(T)))|\mathit{acts}(\hat{P})) \neq \mathit{scheds}(\hat{P})$.

To prove the claim, take any target T such that $\exists s \in \mathit{scheds}(T)$, and s contains more than two output actions. Let s' be the schedule of the renamed target $\mathsf{rename}(T)$ that corresponds to s. Then, by Thm. 8 s' is contained in $(\mathit{scheds}((\mathcal{M}_2 \times \mathit{rename}(T)))$ (since the output actions of the target and the monitor are disjoint). Also, s, and thus s' contain more than two output actions.

Moreover, by definition of $\hat{P}_i$, $\mathit{acts}(\hat{P}) = \mathit{Output}(\hat{P}_i) \supseteq \mathit{Output}(\mathsf{rename}(T))$, for any rename function. And since every element $\hat{P}_i$ of $\mathcal{P}$ does not contain any schedules with more than two output actions, $(s' \mid \mathit{acts}(\hat{P})) \notin \mathit{scheds}(\hat{P})$. This concludes the proof of the claim.

□

It follows from Thm. 2.6.1 that an input/output-mediating monitor enforces strictly more policies than an input-mediating monitor.

Corollary 2.6.1. $\{\mathcal{P} \mid \mathcal{P}$ *is enforceable by some input-mediating* $\mathcal{M}_1\} \subsetneq \{\mathcal{P} \mid \mathcal{P}$ *is enforceable by some input/output-mediating* $\mathcal{M}_2\}$.

The subset direction is straightforward: for any input-mediating monitor M_1 we can construct an input-output mediating monitor M_2 that echoes to the environment every output action that it intercepts. It is clear that for any target T, M_1 and M_2 will have equivalent behaviors. The not-equal direction follows from Thm. 2.6.1.

If we instantiate in Cor. 2.6.1 monitor M_1 with a truncation monitor T_{M_1} and monitor M_2 with a truncation monitor T_{M_2}, we get the following result.

Corollary 2.6.2. *Given an input-mediating truncation monitor* T_{M_1} *and input/output-mediating truncation monitor* T_{M_2}, $\{\mathcal{P} \mid \mathcal{P}$ *is enforceable by* $T_{M_1}\} \subsetneq \{\mathcal{P} \mid \mathcal{P}$ *is enforceable by* $T_{M_2}\}$.

Cor. 2.6.2 illustrates how differences in the implementation of monitors with seemingly equal power, or capabilities, (according to certain previous models [30]), affect the enforceability of security policies.

In fact, when taking into consideration implementation details of monitors we have to revisit previous results on enforceability of policies by monitors with different operational semantics. For example, it has been proven that edit monitors can enforce a strict superset of policies that truncation monitors enforce [30]. However, this relation does not hold if we consider how the monitors are implemented (i.e., what interface of the target they monitor):

Corollary 2.6.3. *There exist a safety policy $\mathcal{P}$, an input/output mediating truncation monitor M_T, and an input-mediating edit monitor M_E such that $\mathcal{P}$ is enforceable by M_T but not enforceable by M_E.*

Thm. 2.6.1 and Corollaries 2.6.1- 2.6.3 establish that some policies are generally enforceable by input/output mediating monitors but not by input mediating monitors. However, for some targets and policies the two architectures are equivalent in enforcement power. The following theorem characterizes the targets and policies for which this equivalence holds.

Theorem 2.6.2. *Given a policy $\mathcal{P}$, a schedule module $\hat{P} \in \mathcal{P}$, a target T, an input/output-mediating monitor $\mathcal{M}_1$ with renaming function* rename_1*, and an input-mediating monitor $\mathcal{M}_2$ with renaming function* rename_2*, such that:*

(C1) *Let* $X = \mathit{input}(\mathsf{rename}_1(T)) \cup \mathit{output}(\mathsf{rename}_1(T)) \cup \mathit{input}(\mathsf{rename}_2(T))$ *in*

$$[X \cap \mathit{acts}(\hat{P}) = \emptyset] \vee [\forall s \notin \mathit{scheds}(\hat{P}) : (s = \sigma; a) \Rightarrow (a \notin X)],$$

(C2) $\mathit{scheds}(\hat{P}) \subseteq \mathit{scheds}(T)$*, and*

(C3) $\forall s \in \mathit{scheds}(T) : [(s = \sigma; a; b) \wedge (a, b \in \mathit{Local}(T))] \Rightarrow$

$$[s \in \mathit{scheds}(\hat{P})] \vee [(s \notin \mathit{scheds}(\hat{P})) \Rightarrow (\sigma; a) \notin \mathit{scheds}(\hat{P})],$$

then $\mathcal{M}_1$ T-enforces $\mathcal{P}$ on T using $\hat{P}$ if and only if $\mathcal{M}_2$ T-enforces $\mathcal{P}$ on T using $\hat{P}$.

Proof. The constraints of the theorem are explained below. The main proof of the theorem is by construction. The proof idea is as follows: for the right direction we construct M_2 by removing from M_1 any transitions that deal with outputs that the target wants to execute, and connect together the transition graph, if disconnected components were created from the removal of the transitions. For the left direction, we construct M_1 by using M_2 as a basis and adding transitions that (1) intercept output actions that the target wants to execute, and (2) forward these intercepted

actions to the environment. More specifically:

($\Rightarrow$ direction) We assume that we have some arbitrary policy $\mathcal{P}$ and target T, and an input/output-mediating $\mathcal{M}_1$ that specifically precisely enforces $\mathcal{P}$ on T. We need to show that there exists an input-mediating $\mathcal{M}_2$ that specifically precisely enforces $\mathcal{P}$ on T.

We construct $\mathcal{M}_2$ by (a) taking the restriction of $\mathcal{M}_1$ that deals only with inputs from the environment, i.e., we ignore the part of $\mathcal{M}_1$ that receives input actions from T and outputs output actions to environment, and (b) for every input i that belongs to the set of inputs that invalidate extensions we simply remove the corresponding transitions: in other words, for every i that invalidates executions, and for every transition of the form $\langle q, i, q' \rangle$, we remove all transitions of the form $\langle q', a, q'' \rangle$, where a is a local action (we keep input actions because of input-enabledness). We will show that if $\mathcal{M}_1$ specifically precisely enforces $\mathcal{P}$ on T under the above constraints, then $\mathcal{M}_1$ specifically precisely enforces $\mathcal{P}$ on T also.

First we will show that the part of $\mathcal{M}_1$ that receives inputs from T and outputs actions to the environment does not do anything non-trivial (under the given constraints), i.e., it either outputs nothing or it simply forwards valid actions that T wants to execute. We do a case analysis on the actions that T might execute and prove that the output-mediating part of $\mathcal{M}_1$ is trivial. First, observe that by (**C3**), $\mathcal{M}_1$ cannot arbitrarily add actions that T might not execute. Second, if T wants to output some action a that obeys the policy, then because of the precise enforcement constraint, $\mathcal{M}_1$ will have to (eventually) output it. Thus in the case of $\mathcal{M}_1$, a is eventually exhibited by itself, whereas in $\mathcal{M}_2$, a will be exhibited directly by T. Finally,if T wants to output some action a that disobeys the policy, then a can either be preceded by some input or not. If it is not preceded by some input, then it must be part of quiescent behavior. But since $\mathcal{P}$ is enforceable, then by Thm. 2.5.1 it must be quiescent forgiving, i.e., it must be valid – contradiction. So a must be preceded by some input. But, by (**C2**), there is some input i that precedes a after which all extensions are invalid. Thus, i will be the last action appearing on the schedule. This means

that since T can still communicate with $\mathcal{M}_1$, $\mathcal{M}_1$ will suppress all the security relevant behavior following i (i.e., it will trivially output nothing).

Note that the latter is equivalent to not forwarding i to T (or any future inputs) and just continue execution by receiving inputs from the environment. This is exactly the construction that corresponds to (b). So under the given constraints the two monitors will both precisely enforce $\mathcal{P}$ on T.

($\Leftarrow$ direction)

We assume that we have some arbitrary policy $\mathcal{P}$ and target T, and an input-mediating $\mathcal{M}_2$ that specifically precisely enforces $\mathcal{P}$ on T. We need to construct an input/output-mediating $\mathcal{M}_1$ that specifically precisely enforces $\mathcal{P}$ on T.

We use $\mathcal{M}_2$ to construct $\mathcal{M}_1$. Specifically, we use the same transition relation as $\mathcal{M}_2$ which we extend in a manner similar to the construction of a truncation monitor from a truncation automaton (Section 2.3), i.e., we add a special state and a queue that buffers the inputs that the $\mathcal{M}_1$ receives from T. Specifically, once $\mathcal{M}_1$ receives some output $\mathsf{rename}(a)$ from T, it records the state it was before starting to receive inputs. If more inputs follow, it adds them to the queue. Once it has finished forwarding to the environment all the outputs that T wanted to execute (i.e., forward a for each $\mathsf{rename}(a)$ received), it returns to the original state to continue execution (as $\mathcal{M}_2$).

Since $\mathcal{P}$ does not reason about the communication between the monitor and the target, it is easy to see that $(\mathit{scheds}((\mathcal{M}_2 \times \mathit{rename}(T)))|\mathit{acts}(\hat{P})) = (\mathit{scheds}((\mathcal{M}_1 \times \mathit{rename}(T)))|\mathit{acts}(\hat{P}))$. Every schedule that $(\mathcal{M}_2 \times \mathit{rename}(T))$ produces is a schedule of $(\mathcal{M}_1 \times \mathit{rename}(T))$, since by construction $\mathcal{M}_2$ does not add any new schedules, and dually, every schedule that $(\mathcal{M}_1 \times \mathit{rename}(T))$ produces is a schedule of $(\mathcal{M}_2 \times \mathit{rename}(T))$, since by construction $\mathcal{M}_2$ does not remove any schedules.

□

Constraint **C1** ensures that the policy does not reason about the communication between the monitor and the target. If the policy prohibited some communication for one monitor but not the other, then the first monitor would have an unfair advantage over the second. For instance, if the policy prohibited the input mediating monitor to forward certain inputs to the target, then the target would not exhibit valid behaviors that depend on these blocked inputs. Constraint **C2** ensures that the input/output-mediating monitor does not have an unfair advantage over the input mediating one, just because the policy requires from the monitor to output actions, even if the target would never perform them. Constraint **C3** ensures that every behavior that the target exhibits and violates the policy must be preceded by an input action. This way, an input mediating monitor can enforce the policy by not forwarding the actions that will cause the bad behaviors. Note that by Thm. 2.5.1 the schedule module must be safety, and thus we are guaranteed that once a behavior of the target becomes invalid, it stays so for all possible extensions.

Thm. 2.6.2 is an illustration of how our framework can help in making sound decisions for designing and implementing run-time monitors in practice. For example, suppose we have a Unix kernel and want to enforce the policy that a secret file cannot be (a) deleted or (b) displayed to guest users. A monitor designer who wants to *precisely* enforce that policy cannot in general use an input-mediating monitor. Although it can enforce (a) by not forwarding commands like "rm secret-file", it cannot enforce (b), because it does not know whether the kernel can, for example, correctly identify guest users and not display secret files to them. However, the designer can check if the specific kernel meets the constraints of Thm. 2.6.2. If it does, e.g., the kernel does not display any secret files while booting (i.e., quiescence forgiving), and does not display secret files to guest users, e.g., through a correct access-control mechanism (i.e., **C3**), then an input-mediating monitor suffices to enforce the policy. The correctness of such design choices might not always be obvious, and the above example demonstrates how our framework can aid in making more informed decisions. Moreover, such decisions can have benefits both in efficiency (by

not monitoring the kernel's output sequence at run time), and in security (since the TCB/attack surface of the monitor is smaller).

Precise enforcement is useful in situations where we want to guarantee that the monitor exhibits certain behaviors that may not be exhibited by the monitored target; e.g., always append to a file the date of modification before closing it (where append and close actions belong to the signature of the target), or log certain security relevant events (where logging actions do not belong to the signature of the target). In the first case, if we allow for transparent and sound enforcement, a monitor could simply buffer the open and writes to the file, and once a close action was issued it could simply discard the particular sequence of actions.

2.7 Related Work

The first model of run-time monitors, *security automata*, was based on Büchi Automata and introduced by Schneider [27]. Since then, several similar models have been proposed that extend or refine the class of enforceable policies based on the enforcement capabilities (i.e., operational semantics) [30], computational powers (e.g., finite or infinite state) of monitors [68], or refinements of the notion of enforcement [69]. In Section 2.3.5 we showed how to encode truncation (i.e., security) and edit automata in our framework. Moreover, in Section 2.4.1 we explained why our definition of policies is more expressive than previous ones (since it allows to define how a monitor responds to the inputs from a target or the environment). As an instance of this expressiveness we showed how to encode the definition of sound and transparent enforcement (often referred to as *effective$_=$ enforcement* [30, 69]) as a special class of policies in our framework. This process can be extended to allow for encoding additional definitions of enforcement, such as that of *late effective$_=$ enforcement* which extends soundness and transparency with the requirement that the output of the monitor is always some prefix of the input that it received [69].

Another track of extensions of Schneider's work includes frameworks that model additional aspects of the monitoring and enforcement process, as opposed to the abilities and powers of the monitors themselves. These frameworks are orthogonal to the models of computational extensions. We can classify the majority of these frameworks into three categories: (1) Static Information, (2) Interaction, and (3) Incomplete Mediation.

Static information. Some frameworks extend Schneider's work to account for information that is not available to the monitor at run-time, e.g., information about the target obtained by, for example, static analysis (also identified as *non-uniformity* [30]). Hamlen et al. described a model based on Turing Machines [29], with which they compared the classes of policies enforceable by

several types of enforcement mechanisms, such as static analysis and inlined monitors. Chabot et al. used Rabin automata to derive in-lined monitors that enforce policies on specific targets, and showed that non-uniform truncation monitors (i.e., monitors that consider only a subset of all possible executions that a target might exhibit) are strictly more powerful than uniform truncation monitors [67]. Our framework is more general than the above as it allows to formally reason about the communication between the target and the monitor.

Interaction. Another line of frameworks focuses on modeling the interaction and communication interface between the target and the monitor. Such frameworks, either revise the computational models, or adopt alternate ones, such as the Calculus of Communicating Systems (CCS) [64] and Communicating Sequential Processes (CSP) [70], to more conveniently reason about applications, the interaction between applications and monitors, and enforcement in distributed systems. An example of revising existing models is Mandatory Results Automata (MRA) which model the (synchronous) communication between the monitor and the target [59, 71]. MRA's, however, do not model the target explicitly, and thus results about enforceable policies in target-specific environments might be difficult to derive. Among the works building on CCS or CSP is Martinelli and Matteucci's model of run-time monitors based on CCS [72]. Like ours, their model captures the communication between the monitor and the target, but their main focus is on synthesizing run-time monitors from policies. In contrast, we focus on a meta-theoretical analysis of enforcement in a more expressive framework. Basin et al. proposed a practical language, based on CSP and Object-Z (OZ), for specifying security automata [73]. This work focuses on the synchronization between a single monitor and target application, although the language is expressive enough to capture many other enforcement scenarios. In our work we focus more on showing how such a more expressive framework can be used to derive meta-theoretical results on enforceable policies in different scenarios, instead of focusing on the (complementary aspect) of

showing how to faithfully translate and model practical scenarios in such frameworks. Gay et al. introduced *service automata*, a framework based on CSP for enforcing security requirements in distributed systems at run time [60]. Although CSP provides the abstractions necessary to reason about specific targets and the communication with the monitor, such investigation and analysis is not the focus of that work.

Incomplete mediation. Concurrently with our work, Basin et al. introduced a model to reason about actions that (truncation) monitors cannot modify [74]. For instance, a monitor cannot control time, and thus if we model time as a specific type of actions (e.g., clock tick actions) a monitor can observe but not suppress them. In our context, an instance of uncontrollable actions are input actions to an I/O automaton: since I/O automata are input enabled they cannot prevent inputs from arriving. Both Basin et al. and we have derived to similar theoretical results, namely that a policy is enforceable only if it does not prohibit sequences of actions that terminate in uncontrollable actions (cf. input forgiveness, Def. 8). Uncontrollable actions (e.g., clock ticks) can be encoded in our framework as input actions to the monitored target from the environment, and using a fairness definition that interleaves clock ticks appropriately with other actions. In addition, our framework allows for: (1) reasoning about (partial) knowledge about the target's behavior, (2) encoding more powerful monitors than truncation automata (e.g., edit automata), and (3) modeling actions that the monitor cannot even observe (either through incomplete rewriting/mediation or internal actions of the target). Although there are scenarios where encoding unobservable actions as uncontrollable ones suffices for certain formal analyses, faithful formal modeling of practical scenarios may require both types of actions explicitly.

Previous work on run-time monitoring has focused individually on just one of the above categories (i.e., static information, interaction, or incomplete mediation). Our work uses I/O automata to establish an automata-based framework that allows reasoning about all these three

categories, and presents several results that belong to more than one of the above categories, thus extending each of the above previous work individually.

Another area of research that is related to our work is *assume-guarantee reasoning* [75, 76, 77, 78]. A major problem in proving through model checking that a distributed system satisfies a given specification is the *state explosion problem*: the number of reachable states to be explored is exponential in the number of concurrent tasks in the system [79]. Assume-guarantee reasoning is a compositional reasoning technique that can be used to deal with the state explosion problem: the user first proves the correctness of individual components of the system, and then, by using appropriate proof rules, she can combine the individual proofs to derive a proof that the system as a whole satisfies the required property. For instance, a proof rule might say that if component X guarantees property B assuming property A (written as $\langle A \rangle\ X\ \langle B \rangle$), and component Y guarantees property C assuming property B (written as $\langle B \rangle\ Y\ \langle C \rangle$), then we can conclude that the parallel composition of X and Y (written as $X\ ||\ Y$) satisfies property C assuming property A (i.e., $\langle A \rangle\ X\ ||\ Y \langle C \rangle$). Of course, in order to use such a proof rule, one must first show that the particular rule is sound (and complete).

As we described in Section 2.3.1, one way that our work differs from previous work in run-time monitoring is the ability to (explicitly) model the target application. Specifically, we defined enforcement (Section 2.4.1) as a set relation between the traces of a property and the traces of a monitored target (i.e., a target composed with a monitor). Thus, given a monitor, a target, and a security policy, one could use assume-guarantee reasoning to prove that the specific monitored target (i.e., the two components interacting) satisfies the given property.

The focus of this chapter was on comparing classes of monitors and classes of policies (Section 2.5 and Section 2.6)—as opposed to showing how to verify concrete instances of enforcement scenarios. Thus, the work in this chapter is closer to work where proof rules of assume-guarantee

reasoning are proven sound or complete (e.g., [80])—as opposed to work that shows how to use such proof rules in practice (e.g., [76]). Assume-guarantee reasoning could be used as an alternative approach to prove results similar to ours, but some of the insight of the results, e.g., the constraints under which two types of monitors with different enforcement capabilities are equivalent (Theorem 2.6.2), might not be obvious from the rules themselves.

Finally, another area of research that is related to our work focuses on designing sound proof rules for assume-guarantee reasoning in the presence of adversaries whose programs (or models) are not available for analysis [77, 78]. Protocol composition logic (PCL) considers network adversaries that interact with cryptographic protocols and can modify messages on the network subject to certain constraints imposed by security properties of cryptographic primitives [77]. System M considers richer adversaries who can supply code over higher-order interfaces exposed by a trusted system [78]. In these settings, the protocol or trusted system acts like a monitor to enforce specific kinds of safety policies in the presence of powerful adversaries. The main difference from our work is that these papers do not provide a characterization of bounds of classes of enforceable security policies by classes of monitors under assumptions about targets (as we did in this chapter); instead, they develop deductive systems to support assume-guarantee reasoning for a class of systems (monitors) in the presence of a class of adversaries (targets).

2.8 Conclusion

Formal models for run-time monitors have helped improve our understanding of the powers and limitations of enforcement mechanisms [27, 30], and aided in their design and implementation [81, 82]. However, these models often fail to capture many details and complexity relevant to real-world run-time monitors, such as how monitors integrate with targets, and the extent to which monitors can control targets and their environment.

In this chapter, we introduced a general framework, based on I/O automata, for reasoning about policies, monitoring, and enforcement. This framework provides abstractions for reasoning about many practically relevant details important for run-time enforcement, and yields a richer view of monitors and applications than is typical in previous analyses of run-time monitoring. For example, our framework supports modeling practical systems with security-relevant actions that the monitor cannot mediate, rather than assuming complete mediation. Moreover, we show how this framework can be used for meta-theoretic analysis of enforceable security policies. In particular, we derived results that describe upper bounds on enforceable policies that are independent of the particular choice of monitor (Thm. 2.5.1). We also identified constraints under which monitors with different monitoring and enforcement capabilities (i.e., monitors that see only a subset of the target's actions; and monitors that have more or less ability to correct a target's invalid behavior) can enforce the same classes of policies (Thm. 2.6.2).

Chapter 3

Distributed Enforcement

3.1 Introduction

As we discussed in Section 1, policy enforcement in practice relies on the use of multiple security mechanisms for two reasons. First, mechanisms may have different capabilities, for instance different *domain* (e.g., network traffic, application logs) or *vantage*, and thus employing multiple security mechanisms allows us to get a more complete view of the network activity and identify more attacks (e.g., a successful remote exploit). Second, typically, attackers rely on several steps in order to perform their attacks, and these steps might be identified by different mechanisms due to, for instance, different domain (e.g. detect a SQL-based worm that scans for port 1433 and then tries to login in on a SQL server). The attacker's temporal behavior can only be identified by correlating information from all these security mechanisms (in the previous SQL-based worm example we will need to temporally order and correlate information from a NIDS and SQL server logs). Although many existing approaches perform such a correlation and analysis at a central location (e.g., [52, 83, 84, 85, 86]), they face two potential limitations: (1) they become a single point of failure, and (2) they can get overwhelmed with the amount of data that they need to collect

and analyze. Thus, in order to achieve better fault tolerance, communication efficiency, and computational efficiency, two types of alternative solutions have been introduced: (1) hierarchical enforcement [87, 88, 89], and (2) decentralized enforcement [90, 91, 92, 93, 94, 95].

In hierarchical approaches, security mechanisms are organized in a hierarchical fashion. At the lowest level of the hierarchy, each mechanism is responsible for a subset of the nodes on the network. At this level mechanisms collect data, perform some analysis, and forward the results to monitors at higher levels, which further analyze the data [87, 88, 89]. For instance, let us assume a monitored network where monitors are organized hierarchically into two levels. At the lowest level of the hierarchy, each security mechanism is responsible for monitoring only a small subset of the nodes in the network. A port scanning attack can be detected efficiently as follows: once a monitor at the lower level detects that a particular source IP address has scanned the ports of some of the nodes it is monitoring, it forwards this information (the source IP, the scanned ports, and the number of hosts that were scanned) to the monitor at the top level. The monitor at the top level collects information from the monitors at the lower level. Once that monitor detects that a particular source IP address exceeds a predefined threshold of scanned hosts, it notifies all other monitors that an attack is in progress. Note that if a monitor at the lower level detects that the predefined threshold has been exceeded (e.g., because all the nodes that it monitors were attacked) it can immediately notify all other nodes for the attack, thus achieving an even more efficient detection process.

Decentralized approaches originated from security mechanisms that required co-operation and coordination (e.g., distributed IDS [90, 91, 92, 93, 94, 95] and distributed firewalls [96, 97]). Security mechanisms are distributed over the network, and each part of the mechanism is operating independently of the other, collecting and analyzing data locally. Once a node realizes that this data might be relevant to some other agents, it forwards the appropriate information to them, so that they can (collaboratively) identify the attack. For instance, let us assume that in a

decentralized monitoring architecture a monitor M installed at a node of the network realizes that a particular IP address is actively scanning for port 1433. Then, M can directly notify a monitor installed at the SQL server to block all traffic from that particular source IP address.

Both hierarchical and decentralized approaches require communication and coordination among the monitoring entities, and thus they must implement certain distributed algorithms in order to achieve their goals. As explained in Section 1, it is of both practical and theoretical importance to characterize the policies that these mechanisms can enforce. This is the goal of this chapter.

We start in Section 3.2 by discussing some key differences between centralized and distributed systems. These differences provide the intuition behind the constraints that we formalize in the characterization of enforceable policies in distributed systems. Next, in Section 3.3 we present some motivating examples of multi-step and distributed attacks that will be used throughout the chapter to illustrate key concepts and ideas. In addition we discuss how multi-step attacks can naturally be modeled as state transition systems, e.g., I/O automata.

In Section 3.4 we explore and present the main characterization of enforceable policies in asynchronous distributed systems. First, in Section 3.4.1 we present our formal framework, i.e., we show how to model distributed systems and monitors using I/O automata, security policies, and what it means to enforce a security policy over a distributed system. Then, in Section 3.4.2 we explain how to reduce the question of which policies are enforceable in distributed systems to the question of which *global monitors*[1] can be decentralized over a given distributed system. In Section 3.4.3 we present some basic decentralization algorithms that work under some very strong assumptions about communication among nodes. Section 3.4.4 presents one of our key

[1] We will use the terms *global monitor* and *centrally specified monitor* to refer to a (possibly hypothetical) monitor that can monitor directly all traffic in a distributed system. Of course in practice such a monitor might not exist (e.g., due to physical limitations and constraints), but as we explain in Section 3.4.2, it provides a nice abstraction for specifying the intended monitoring behavior without having to worry about details of the underlying communication network. We call such a monitor *global* because often in the literature of distributed enforcement, a *central monitor* or *centralized monitor* is a monitor that, in a network of multiple communicating monitors, takes all enforcement decisions.

contributions, namely a blueprint for designing algorithms that can decentralize monitors over distributed systems. The details of the steps of the blueprint are analyzed in Section 3.4.5, Section 3.4.6, and Section 3.4.7. Specifically, in Section 3.4.6 we present two novel decomposition algorithms that transform a global monitor into two important types of distributed enforcement architectures: centralized and decentralized.

Next, in Section 3.5.1, we discuss how our results of decomposing a global monitor into centralized and decentralized architectures over asynchronous distributed systems relate to the enforcement of security policies in synchronous systems. In Section 3.6 we discuss approaches for hierarchical enforcement of security policies in synchronous and asynchronous networks, and we give a first characterization of the policies that are (efficiently) enforceable in a hierarchical manner. Next, in Section 3.7 we present another important contribution of this work, namely that in distributed systems, and under certain constraints, it is possible to simulate the global behavior of powerful monitors using weaker monitors that interact with each other. Finally, in Section 3.8 we present related work on decentralized and hierarchical enforcement of security policies, and we conclude in Section 3.9.

3.2 Differences Between Centralized and Distributed Systems

In order to understand decentralized algorithms and how monitors can distributedly enforce security policies, it will be constructive to explore first what are the key differences between distributed systems and centralized systems[2]. Centralized systems have *full knowledge* and *immediate access* to all information and context that is relevant to the computation: input is known, intermediate steps of the computation are known, and in general the state of the computation (e.g., contents of CPU, memory, and disk) is known and available. Of course access to the state is not *instantaneous*, i.e., accessing the disk is *less instant* than accessing the memory. But all the information that centralized systems need to perform their computations, is available to them: their computation does not depend on information that is not present within the system, i.e., information that is not *immediately* accessible. Moreover, centralized systems execute *at most* one *atomic* (i.e., indivisible) activity at a time. At the most granular level of a centralized system, that atomic activity is usually a CPU instruction, but by adopting a more coarse-grained view (e.g., in single-tasking operating systems) atomic activities could also include system calls or the execution of a single program.

On the other hand, distributed systems are composed from multiple atomic entities. These entities attempt to compute a global function by executing (in parallel) their own atomic activities, while coordinating through some (shared) underlying communication medium. The entities may be located on different physical locations and have different characteristics (e.g., CPU speed, memory size, and disk size). In addition, the communication medium that entities use to coordinate might be different depending on the scenario at hand. For example, the communication medium could be fast and reliable (e.g., shared memory systems) or slow and unreliable (e.g.,

[2]The presentation and the material in this section follow corresponding presentation and material from [1] and [98].

multi-hop asynchronous networks).

This nature of distributed systems has several important consequences [98]:

1. Global knowledge: Information that is necessary by each entity to make progress towards the common goal (i.e., computing the global function) may not be immediately available: inputs might arrive on different nodes and local processors might take steps that are not known to the rest of the system.
2. Global ordering: Steps by different local processors might be taken at the same time. This means that computation steps can no longer be totally ordered, but instead must be partially ordered.
3. Global time: In centralized systems there exists a global clock and all steps the system takes can be totally ordered. However, in distributed systems such a global clock might not exist. Each node has its own clock, and these local clocks might proceed at different speeds[3]. If we combine this with the difference in processors' speeds and communication latency, the system might not be able to order events happening on two different nodes, even if these events happen at different (real and absolute) times.
4. Failures: In an event of a failure in a centralized system, e.g., power failure, the system stops its operation. In a distributed system, because nodes are in different physical locations, if one entity fails the rest of the distributed system may continue its operation[4].

These characteristics of and differences between centralized and distributed systems guide the formal models that have been introduced for each system respectively. Typically, models for centralized systems (e.g., Turing Machines [31]) consider them to be single, atomic, controlling units. On the other hand, due to the large design space of distributed systems (nature of commu-

[3]Even if nodes try to synchronize using common global clocks (e.g., atomic clocks), or try to synchronize their clocks using some protocol (e.g., NTP), there is still some margin of error. Although for some applications this might be an acceptable error for synchronization, in general, it might not be.

[4]In this thesis we don't deal with failures, and we leave it as a topic for future research.

nication medium, existence of global clock, types of failures, etc.), several models for distributed systems have been introduced in the literature [1]. Two of the most general models for distributed systems are *message passing* and *shared memory* models [1, 98, 99].

Message-passing model. In the message-passing model communication is modeled explicitly: whenever two nodes want to communicate, they need to send messages through certain communication channels. The message-passing model is useful for modeling communication networks and has been commonly used to study issues that depend on communication, e.g., what algorithms and protocols can we use to simulate a broadcast service if the underlying network offers only point-to-point communication[5]. Message passing models can be further classified to two models: the *synchronous model* and the *asynchronous model*.

Synchronous model. In the synchronous model all nodes are operating in synchrony: first they all send messages to their neighbors; then they receive messages from their neighbors; and finally they perform some local computation. All these steps are considered to be done in a single round (i.e., instance). Message transmission and local computation are assumed to take negligible time.

Asynchronous model. In the asynchronous model there is no global clock in order to achieve synchronization. Nodes need to exchange messages in order to make decisions, and they do not know what the state of a message is until they have received it. Thus, they cannot infer using, e.g., timeouts if a message was sent by some other node, or if a message was lost. Notice that in the synchronous model all nodes in the network can exchange complete information within a bounded time – $O(diam)$ rounds (where *diam* is the diameter of the network), whereas in the asynchronous model such a communication might require unbounded time.

[5]In this thesis we focus on models where the communication model is assumed to be point-to-point.

Non-determinism. Non-determinism is a feature that is common to both models [1, 98]. However, it needs to be emphasized that it is being used to model different *realities*. In the synchronous model steps might be taken by different nodes at the same round and thus we must use a partial order in order to model the occurrence of events. Here, non-determinism captures this within-round partial order. Rounds, on the other hand, are totally ordered. In the asynchronous model steps are not assumed to be taken at the same time-slot, i.e., round; this is due to differences in speeds of local computations and message transmission delays. These differences cannot be predicted. Thus, even though nodes might be executing steps deterministically, the occurrence of these events might differ with different runs of the system. As a result, the executions of a system in the asynchronous model is non-deterministic: unless events are *causally related* (i.e., one event triggers another one) then all their interleavings are possible and need to be considered.

Shared-memory model. In the shared memory model communication amongst nodes (typically called processes [1]) happens implicitly through writing and reading to a memory that is shared (i.e., accesible) by all nodes. The shared memory model is useful for modeling parallel architectures, where communication is not the issue and the focus is on how nodes can synchronize in order to solve a problem collaboratively, e.g., algorithms and protocols to simulate having access to a read-modify-write shared variables when the underlying hardware provides only read-write shared variables.

Thus, one benefit of the shared memory model is that it abstracts away from communication issues and, compared to the message passing model, it is much closer to the centralized model. This means that solutions written in the shared memory model might be simpler than the ones written in the message passing model, since there is one less complexity to consider – communication.

The message passing model and shared memory model have been used to model and solve

different problems. Even though the shared memory model might seem *less expressive*, since it deals with less complexity (i.e., communication) than the message passing model, they are *essentially* the same: there are algorithms that can transform solutions from one model to the other (and vice versa) [1]. In this thesis we take advantage of this fact. Although we use message passing model to express distributed enforcement scenarios, we use the shared memory model to either directly express centralized monitors or transform centralized monitors to an intermediate representation before de-centralizing them. Monitors are simpler to specify this way, because they are written as if there was a shared (i.e., centralized) memory. Then, we use algorithms to decompose[6] these centralized monitors to distributed ones, that behave in exactly the same way (in a way that will be made more formal next). Of course, since these monitors are assumed to be centralized, the issues that we discussed previously (transmission delays, global time, immediate global information knowledge, and faults) will limit the extent to which such transformations are possible.

[6]We will be using the terms *decompose* and *decentralize* interchangeably.

3.3 Multi-step and Distributed Attacks

One of the motivations for decentralized policy enforcement is to better identify attackers by correlating information about their activities. Attackers rarely choose to perform random (or uncorrelated) attacks because they can be detected easily [50]; it is much more difficult to identify determined attackers that employ intelligent strategies towards achieving their goals [50]. Typically, intelligent attackers rely on several stages in order to perform their attacks and remain unnoticed, e.g., reconnaissance, intrusion, privilege escalation, and goal steps [51]. In order for the attacker to move to the next stage of a multi-step attack, she must complete successfully the previous ones; e.g., an intrusion attack step will rely on the information received from a completed reconnaissance attack step.

Next we describe a few simple motivating example attack scenarios that have been discussed in previous work of distributed and decentralized security policy enforcement [83, 87, 94, 95].

Attack #1**.** A *doorknob attack* is an attack where the intruder tries to check for vulnerable hosts by trying a small number of user accounts and passwords on a large number of computers (different combination of username/password on each host) [83, 87, 95]. Since the attacker attempts only a few logins on each host the attack might not be detected by each host individually; information needs to be correlated and either the attempts must be traced back to the same source, or some patterns in the behavior of the attacker must be identified.

Doorknob attacks belong to a larger class of attacks, called *coordinated attacks*. Coordinated attacks are multi-step attacks, where the attacker is distributing the steps of the attack over multiple sessions in order to speed up the attack or avoid detection [87].

Attack #2**.** *Network browsing* is an attack in which the attacker is accessing a number of files on many different computers within a short period of time [95]. Similarly to doorknob attacks, because the activity on individual hosts is small, an alarm might not be raised, unless information is correlated. Of course, some pre-processing can happen locally at the nodes themselves before information is correlated. This might minimize the amount of data transfered, e.g., in the case that the analysis happens at a central location.

Attack #3**.** *Worms* are programs that use resources on one machine to attack other machines, and propagate themselves across a network [87]. Due to the way that worms propagate, the traffic they generate forms a tree-like pattern of similar activities [87]. This pattern can be used to detect such attacks, but it requires a global view of the network or correlating information from multiple nodes.

Attack #4**.** This example is a variation of the Apache chunked-encoding exploit attack described in Section 1 [94]: the attacker first scans each host on a subnet for an open port 80 and 8080; then, when a webserver has been identified, the attacker attempts to send some malicious HTTP traffic. Note that in attacks #1, #2, and #3, similar events from several hosts were communicated and aggregated to *detect* suspicious patterns and raise alarms. Attack #4 can be used to illustrate how communication among hosts can also help in the *prevention* of attacks. Specifically, one way to prevent the multi-step attack #4 is through communication and synchronization among the monitoring components: once a monitor on a host (other than the webserver) detects the port scanning activity, it notifies the webserver with the suspicious IP. When the monitor at the webserver later receives traffic exactly from that source, it can take appropriate remedial actions, such as raising alerts, or responding to the HTTP requests in a different way than it would normally do for legitimate traffic.

Attack #5**.** A variation of *attack* #4 is a hypothetical variant of the CodeRed worm [94]. In this attack the attacker not only scans for open ports 80 and 8080, but also sends a HINFO request to the DNS server in order to identify the type of operating system the web server is running (so that Microsoft machines are targeted more accurately). In this attack, the (monitor at the) web server raises an alarm *only if* it receives HTTP traffic from an IP address that performed *both* a port scan *and* contacted the DNS server; otherwise it processes the received HTTP traffic as normal.

Each of the aforementioned examples describes the sequence of steps that the attacker takes towards achieving her goal. In addition, some of the examples, e.g., attacks #4 and #5, specify the steps that the monitoring agents should take in order to prevent the attacker's end goal. In the other examples, the steps that the monitors take are implicit: agents allow all traffic, including malicious, and when they have enough information to determine that an attack has happened an alarm is raised.

Note that the above examples specify *what* constitutes an attack and *what* the monitors should do, but they do not describe *how* the monitoring agents will achieve their goals (i.e., meet their specifications). For instance, in attack #5, it is assumed that the webserver has *immediate* access to all the information necessary to make its decision. However, as we discussed in Section 3.2, immediate access to global knowledge is a property of centralized systems, that distributed systems may not have. In particular, if we wanted to design a decentralized solution for enforcing this policy, it is not clear whether this task would be achievable, and how (e.g., which mechanisms should be used, how should they communicate, etc). For instance, in order for the webserver to know whether to block or allow traffic from a specific IP address it must receive information from other monitoring agents, including the one at the DNS server. But, the DNS server might not be available to provide the relevant information to the webserver: the attacker could attack the DNS server itself to render it inaccessible after she has received the appropriate information.

Thus, the webserver needs to decide whether to allow an HTTP request without having access to all necessary information. If the webserver waits until the DNS server is available again, then a legitimate request might not be serviced on time, leading, effectively, to a denial-of-service attack. Essentially, the legitimate request is treated as if it was an attack (i.e., a false positive). On the other hand, if the webserver allows the request, then an attacker could complete her attack, thus the request is treated as a legitimate one (i.e., a false negative). So it seems, that a decentralized architecture cannot avoid having false positives or false negatives, even though in a (hypothetical) centralized system, this policy would be enforceable since the monitor would have immediate access to the global knowledge.

In this particular example there is a solution: the DNS server will not respond to the attacker until after the webserver has been notified. But this solution requires a monitoring architecture where the monitors can delay events, i.e., they can actively mediate events in the network. This means that an IDS would be unable to enforce[7] this policy; at best it could detect the attack, but only *after* the attack has happened.

As we mentioned, the above example attacks, and the policies they implicitly describe, *assume* that global knowledge is immediately available. This was a property of centralized systems, that distributed systems may not have. Thus, for the rest of this chapter we will refer to such attacks and policies as *centrally specified*.

Centrally specified policies are simpler to describe because (1) we do not have to reason about *how* the policies are enforced, and (2) we can make the (strong) assumption that global knowledge is immediately available. However, as we discussed in Section 3.2, this assumption may not always hold in distributed systems. Thus, a goal is to keep the convenience and simplicity

[7]Remember that whether a mechanism *detects* or *prevents* an attack is specified in the policy itself. Thus, when we say that an IDS cannot enforce a policy, this could mean two things: (1) the IDS cannot make the correct detection decision, or (2) the IDS cannot prevent or deter an attack. Even though the second case might seem straightforward, it emphasizes the point that terms like enforcement, policies, etc., must be carefully defined because often they might be used in ambiguous ways, e.g., when theory is used to analyze what practical mechanisms can do.

of specifying policies (and attacks) in a centralized manner, while knowing whether these policies are enforceable in a decentralized manner, and with what types of mechanisms. In this chapter we provide a principled way to provide answers to questions such as: Are all policies that are enforceable in centralized systems also enforceable in a decentralized manner? If not, what is the characterization of policies that are enforceable? What mechanisms are required in order for centrally specified policies to be enforceable in a decentralized manner? Does knowing the attacker's strategy (i.e., the multi-step specification of attacks) help us enforce more policies?

3.3.1 Multi-step Attack Specification Using Preconditions and Postconditions

As the above examples illustrated, multi-step attacks are sequences of actions (which may be attacks themselves) that take the system from an initial state to a final compromised state [100]. These sequences of *security relevant actions* include the actions that the attacker takes and potentially the actions that the system takes to respond to the attacker. Thus, these sequences describe policies as they were defined in Section 2.3.3, even though sometimes the steps that the monitored systems take are implicit in the informal description of the policy. For instance, in attack #5, the security relevant actions that the attacker takes are the $scan(IP, port)$, *DNS_query*, and *HTTP_request* actions. The actions that the (monitored) system can take are the responses to the attacker's actions, namely $response(IP, port)$, $response(DNS\_query)$, and *HTTP_response*. Finally, the (monitored) system has an additional *raise_alarm* action which is exhibited in the case of an attack. So, an attack scenario could be modeled with the following sequence of actions: $\langle scan(IP, port)$, $response(IP, port)$, *DNS_query*, $response(DNS\_query)$, *HTTP_request*, *raise_alarm*$\rangle$.

Multistep attacks can be specified using preconditions and postconditions, i.e., state transi-

tions. A precondition describes the necessary conditions for the step to be taken and a postcondition describes the effect, or the conditions that hold, after the step is taken. A key idea of approaches that are based on preconditions and postconditions is that postconditions of certain attacks can be used as preconditions for other attacks, and thus larger scenarios can be built from the specification of individual steps [95].

An interesting characteristic of multi-step attacks and policies, compared to arbitrary sequences of steps, is that the attacker and the monitored system interleave steps: the attacker block after taking a step because she needs the returned result, before executing her next attack – and dually for the monitored system. In other words, the attacker and the monitored system take turns executing actions, and until one of them is done processing, the other one is blocked. This observation motivates the following assumptions.

Assumptions in modeling multi-step attacks and policies. For the rest of this chapter we will assume that the attackers and the monitored system, i.e., the implementation of the security policy, are *causal*: after the attacker sends a (single) input to a node, she blocks until the monitored system produces a response[8]. After the monitored system produces a response, it goes back to the state of receiving input (on that particular node). Note that we allow the attacker to send multiple (single) inputs to multiple nodes. Thus, we allow concurrent behavior among different nodes, but we assume sequential behavior (i.e., sequences of inputs and outputs) on single nodes. Example attacks $\#1 - \#5$ meet this assumption.

Preconditions and postconditions (and state transitions) have been used to describe sequential specifications in many areas of computer science including concurrent object specification [101,

[8]Even though in most of our examples attackers block after sending single inputs, it might be useful for some scenarios to allow blocking after receiving (finite) sequences of inputs. Although it is not hard to extend the theory presented in this chapter to account for such cases, we chose to avoid the added complexity to present the key ideas and results in a reasonable and simple way.

102], security policy specification [103, 104, 105], and multi-step and distributed attacks signature specification [49, 95, 100, 106, 107, 108, 109].

In fact, preconditions and postconditions can be used not only for the specification of misuse detection systems, but also for the description of policies in specification-based detection systems. Misuse detection systems, attempt to identify attacks by detecting pre-specified patterns of *bad behaviors*, as the examples described above. Specification based detection systems rely on pre-determined specifications of *valid behaviors*, e.g., specifications derived from RFCs or other descriptions of protocols such as the IP, ARP, TCP and UDP [110]. Any deviation from these behaviors constitutes an attack, and thus attacks can be detected even if they have not been previously known [111].

In this thesis, we use I/O automata (and PIOA) to model targets, monitors, and security policies. Our definition of security policies (Section 2.3.3) allows us to describe both specification-based approaches (through transparency) and misuse-based approaches (through response actions, such as raising alarms). For instance, Fig. 3.1 depicts the transitions of an I/O automaton specification for attack #5.

There is a straightforward translation from the semantic framework of I/O automata to the description of distributed algorithms using preconditions and effects (i.e, postconditions). In fact, it is typical to use precondition and effects to describe I/O automata without distinguishing between the two levels of description (i.e, semantic and syntactic) [1, 112]. We too follow this approach in this thesis. Due to this fact, our framework can serve as the basis for an attack specification language with formally specified semantics. Although specification languages for I/O automata have already been introduced in the literature [112, 113, 114], we leave the translation of our framework to a concrete implementation of a specification language as future work.

In conclusion, state transitions using preconditions and postconditions can be used:

1. to represent multistep attacks, either explicitly, by defining specific sequences of transi-

Transitions: *scan(IP,port)*
// $T1$ is a set containing the source IP addresses
// that have been identified scanning
Effect: *T1.append*(*source_address*)
DNS_query
// $T2$ is a set containing the source IP addresses that
// have been identified contacting the DNS server
Effect: *T2.append*(*source_address*)
HTTP_request
// *Req_list* is a set containing pending HTTP requests
Effect: *Req_list.append*(*source_address*)
response(IP,port)
Precondition: *T1.contains*(*source_address*)
Effect: None
response(DNS_query)
Precondition: *T1.contains*(*source_address*)
Effect: None
raise_alarm
Precondition: *T1.contains*(*Req_list.head*())
&& *T2.contains*(*Req_list.head*())
Effect: None
HTTP_response
Precondition: !*T1.contains*(*Req_list.head*())
|| !*T2.contains*(*Req_list.head*())
Effect: *req_list.removeHead*()

Figure 3.1: I/O automaton transitions for attack #5.

tions as attacks [100], or implicitly, by defining a set of atomic bad transitions that can be combined in valid ways (i.e., matching preconditions and postconditions) to create arbitrary sequences [95];

2. to implement detection tools, where the monitor simulates the transition system that defines the attack or the valid behavior [100]; or
3. to verify or prioritize alerts by recognizing pre-specified patterns that are known to be irrelevant, e.g., detecting Apache exploits even though the network has only Microsoft web servers [95].

3.3.2 Theoretical and Practical Limitations of Attack Detection Using State-transition-based Signatures

Accurate detection of attacks using signature specification is not trivial in practice for several reasons, including the following:

1. Misuse signatures need to characterize exactly the steps that attackers will take. This not only requires expert knowledge, but also gives the opportunity to attackers to evade detection by (perhaps even slightly) changing the steps they take [92, 115].
2. Some security relevant actions that are important in the detection process might be missed, e.g., because the monitor is overwhelmed with incoming traffic resulting to dropped packets [50, 92]).
3. In specification-based detection approaches, differences in implementations of the same protocol might lead to false positives or false negatives in the detection process [110, 116, 117, 118].
4. Specification of distributed attacks might require reasoning about concurrent events and race conditions which are notoriously hard to do in practice [92, 119].

It is well known to partitioners of computer and network security that detecting attacks with misuse and specification-based approaches without false positives or false negatives is almost impossible in practice [50]. It is important to emphasize that these false positives and false negatives are due to the aforementioned limitations of specifying attacks, and, potentially, due to bugs in the *implementation* of underlying detection mechanisms. This means that *if* signatures were accurate and complete, then by *definition* (or by *design*) misuse and specification-based detection systems would provide zero false positives and false negatives [26, 110, 116, 117, 118]. However, in this thesis we show that this claim holds, in general, only for centrally specified signatures: even if

we assume correct and complete signatures, it might be *impossible* to design (and thus implement) a decentralized monitoring architecture that detects attacks described by these signatures without false positives and false negatives. More specifically, if we consider all the attacks that can be specified and detected in centralized systems without any false positives and negatives, then the attacks that can be detected without any false positives and negatives in a decentralized manner form a strict subset. This result extends the existing list of impossibility results for security mechanisms: e.g., it is impossible, in general, to construct a security mechanism that through static analysis will identify all viruses without false positives or false negatives [29, 120]; it is impossible to construct an *execution monitor*, or run-time monitor, that will enforce policies that do not belong to the class of computable safety properties [27, 29, 30]; and it is impossible, in general, to construct decentralized monitors that can enforce centrally specified policies without any false positives and false negatives.

3.4 Asynchronous Enforceability

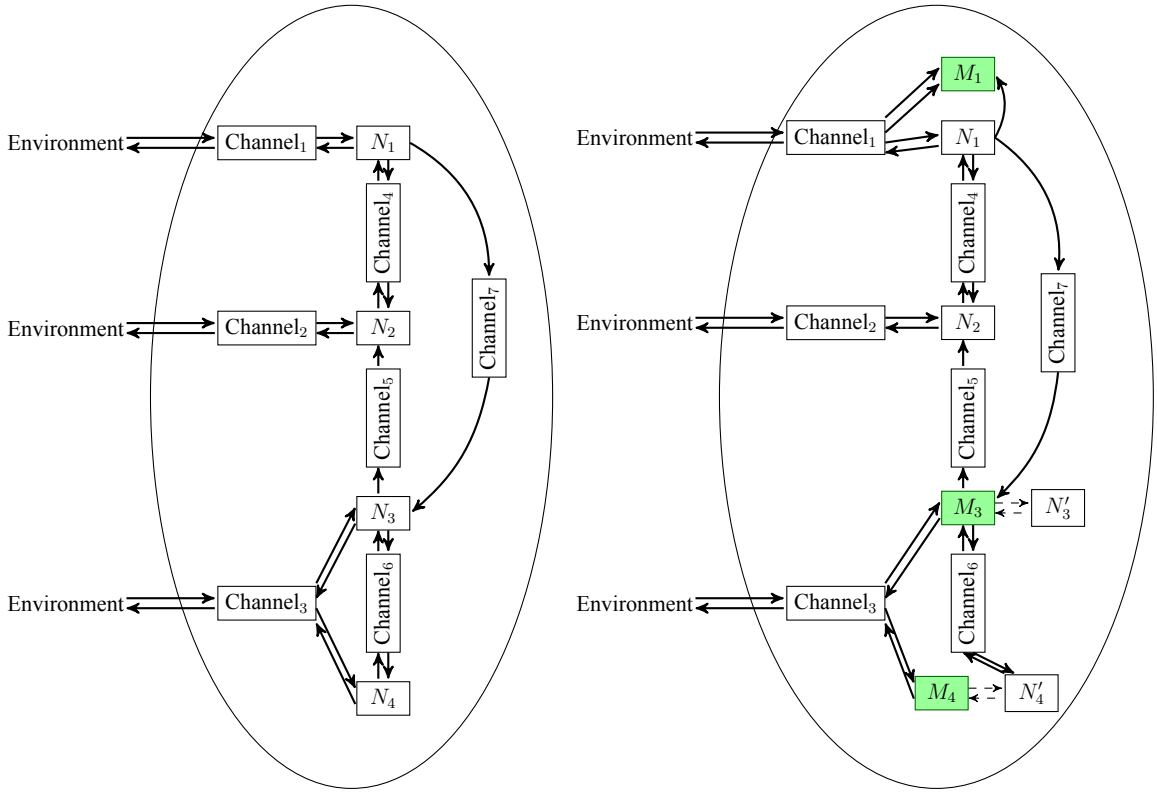
3.4.1 Definitions

3.4.1.1 Modeling Distributed Systems

An asynchronous distributed system can be thought as a directed graph $G = (V, E)$, where V are the vertices of the graph and correspond to the nodes of the system, and E are the edges of the graph and correspond to the communication channels between pairs of nodes. We are assuming systems that correspond to arbitrary connected graphs, i.e., we do not assume that our systems are necessarily complete (i.e., every pair of distinct vertices is connected by a unique edge). *Nodes* of G communicate over *channels* associated with directed edges. To model the asynchrony of the system we model both nodes and channels as I/O automata. We assume a fixed message alphabet M.

Node Automata. Each node i in the system is modeled as a (deterministic) I/O automaton N_i. The set of all node automata is denoted as $\hat{N}$. Each N_i has output actions of the form $send(m,j)_i$, where j is an outgoing neighbor of i, and m is a message (i.e., $m \in M$); and input actions of the form $receive(m,j)_i$, where j is an incoming neighbor of i. N_i does not have any other external actions.

Channel Automata. The channel associated with each directed edge (i, j) of G is modeled as an I/O automaton $C_{i,j}$. The input actions of $C_{i,j}$ are $\{send(m,j)_i \mid m \in M \wedge i \neq j\}$. The output actions of $C_{i,j}$ are $\{receive(m, i)_j \mid m \in M \wedge i \neq j\}$. Although channels could be modeled as arbitrary I/O automata, which would allow arbitrary behaviors (e.g., reordering or lossy channels), here we assume that channels are *reliable FIFO channels* [1].



(a) Distributed system with channels (b) Monitored distributed system

Figure 3.2: Decentralized monitoring

Environment Automaton. Sometimes we will model the environment as an I/O automaton, denoted $\mathcal{E}$, which models the external world with which the distributed system interacts. This can be useful, for example, to formally state assumptions about the environment, such as the one from Section 3.3 where the attacker is assumed to be causal.

In addition, we use I/O automata $C_{E,j}$ and $C_{i,E}$ to model (as channels) the communication of the system with the outside world (i.e., the environment E). This is depicted in Fig. 3.2a, where each box (channel or N_i is an I/O automaton). The set of all channel automata is denoted as $\hat{C}$.

Distributed System. The distributed system is modeled as the composition of the above automata:

$$DS = \prod_{N_i \in \hat{N}} N_i \times \prod_{C_{i,j} \in \hat{C}} C_{i,j} \quad (1)$$

3.4.1.2 Modeling Monitored Distributed Systems

A monitored system is a distributed system *MS*, where there is an underlying system *DS* of n nodes, and a set of m monitors, $m \leq n$, that are attached to the nodes of *DS*. A monitor can be attached to the underlying node either by completely mediating, partially mediating, or passively observing security relevant actions.

As we discussed in Section 3.2, the design space for distributed systems and distributed monitors is very large. Thus, in our definitions, analyses, and results we will make some assumptions that, we believe, are relevant in practice. For instance, we assume that monitoring cannot change the architecture of the underlying network. Even though in some cases changing the network architecture is desirable (or even necessary if we want to enforce a specific policy that is not enforceable in the existing architecture), in general, such a task is undesirable (or even impossible), especially for distributed systems where the nodes are located in geographically separate locations. In this thesis we will assume that when we are trying to enforce a policy on a distributed system, we cannot add, remove, or change communication channels in any way, i.e., the network infrastructure will remain the same regardless of the type of monitoring we do on the network. Since will not be further concerned with communication channels, for notational convenience, in equation (1), we can consider the communication channels as a single I/O automaton C which is the result of composing all communication channels C_i:

$$C = \prod_{C_i \in \hat{C}} C_i. \quad (2)$$

Using (2), equation (1) becomes:

$$DS = \prod_{N_i \in \hat{N}} N_i \times C. \quad (3)$$

Each monitor i is modeled as an I/O automaton M_i. The set of monitors is denoted as $\hat{M} =$

$\{M_i \mid 1 \leq i \leq m, m \leq n\}$. Since some nodes of *DS* might not be monitored, we denote the set of monitored nodes as $\widehat{MN}$, and the set of unmonitored nodes as $\widehat{UN}$. To model the attachment of a monitor to a node automaton N_i, we will use the model of Section 2.3.1, where a monitored node is modeled through appropriate renaming and hiding functions, as $MN_i = \mathit{hide}_{i_{\Phi_i}} M_i \times \mathit{rename}_i(N_i)$. Notice that since the only external actions of N_i are actions to send and receive messages from channels, the same holds for MN_i. This models the fact that the attached monitors must respect the underlying network architecture of *DS*.

The monitored system *MS* is described by the following equation, which builds on equation (3):

$$MS = \left(\prod_{N_i \in \widehat{MN}, M_i \in \hat{M}} \mathit{hide}_{i_{\Phi_i}} M_i \times \mathit{rename}_i(N_i) \right) \times \left(\prod_{N_i \in \widehat{UN}} N_i \right) \times (C). \tag{4}$$

The first component of the composition *MS* is the composition of the monitored nodes $\prod_{N_i \in \widehat{MN}, M_i \in \hat{M}} \mathit{hide}_{i_{\Phi_i}} M_i \times \mathit{rename}_i(N_i)$, the second component is the composition of the unmonitored nodes $\prod_{N_i \in \widehat{UN}} N_i$, and finally the last component is the composition of the communication channels C.

In Fig. 3.2b we show the model of a monitored version of the distributed system depicted in Fig. 3.2a:

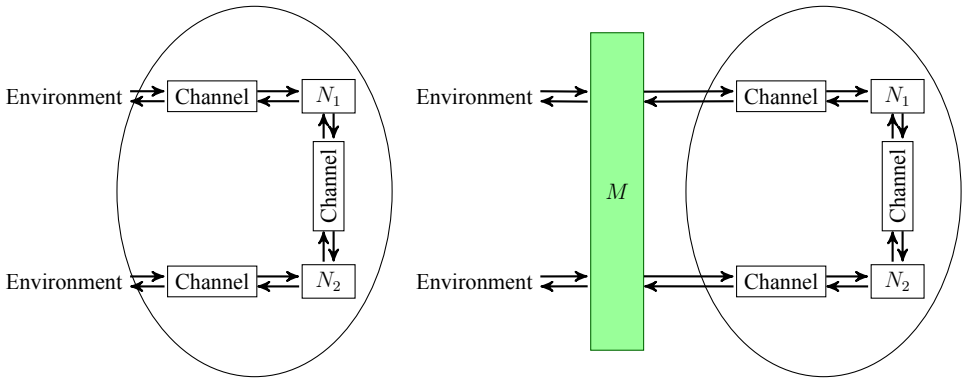
- M_1 models an IDS that is partially and passively monitoring the communication of N_1 (through a spanning port on Channel_1 and a network tap on Channel_7);
- N_2 is an unmonitored node;
- M_3 models an IPS that is actively monitoring N_3 by complete mediation. The external actions of M_3 contain all the external actions of N_3 (now labeled N_3'), and all the $\alpha-$renamed actions of $\mathit{rename}(N_3')$.
- M_4 is another IPS that partially and actively monitors the communication of N_4 (now labeled N_4'), through Channel_3.

3.4.1.3 Policies and Enforcement

To define policies and enforcement in distributed systems we build on the definitions of security policies and enforcement from Section 2.3. However, even from the simple enforcement scenarios that the framework in Section 2.3 could model it was clear that the definitions of policies and enforcement can get quite complex. One reason for this complexity is the many design choices available, e.g., target-specific vs generalized enforcement, different fairness assumptions. Distributed systems and distributed enforcement add additional complexity to the analysis of enforcement scenarios. Thus, in order to simplify the presentation of the results in this section we are going to make the following assumptions:

1. We focus on *traces* of distributed systems and not schedules, i.e., we will not deal with enforcing policies on the internal communication of the system.
2. We focus on *unfair traces*, i.e., we will not make any fairness assumptions. However, in some places we will discuss on the effect that a fairness assumption would have in the presented content or results.
3. We focus on policies that consist only of a single set of trace modules, i.e., we are restricting our attention to *properties* [27, 30].
4. We focus on target-specific enforcement, i.e., on enforcing policies on particular target distributed systems.

The main results of this section will still hold if we remove some of the above constraints. For instance, if we want to reason about the internal communication of the distributed system (i.e., reason about schedules and not just traces), we can temporarily un-hide internal communication by making all internal actions output actions. Then, we can apply the results and algorithms of this section (since we are dealing only with input and output actions, i.e., traces), and finally, re-hide the output actions that were original internal.



(a) Distributed system (b) Global monitor

Figure 3.3: Global enforcement

Typically, designing central algorithms is easier than distributed algorithms [1, 104, 119]. This is because central algorithms view the system as a single atomic object, and thus do not have to deal with complexities of interaction and concurrency. In our case, as we will see in the next section, central algorithms (i.e., monitors in our case) are also useful for analyzing the constraints under which policies are enforceable in distributed systems.

Our first definition of enforcement will consider the distributed system to be a single entity. To differentiate the terminology between this case and the case where multiple monitors enforce a policy on a distributed system but only one makes the enforcement decisions, typically called *centralized enforcement*, we will refer to the former enforcement as *global enforcement*.

Remember that in the following definitions policies are schedule modules and we focus on traces (not schedules).

Definition 15. *(Global Enforcement) Given a policy $\hat{P}$, a distributed system DS, and a monitor M, we say that $\hat{P}$ is globally enforceable on DS by M if and only if there exists a renaming function rename and a hiding function $hide_\Phi$ for some set of actions $\Phi \subseteq \Big(Output(M) \cup Output(rename(DS))\Big)$ such that:*

$$\Big(traces(hide_\Phi(M \times rename(DS))) \mid acts(\hat{P})\Big) = traces(\hat{P})$$

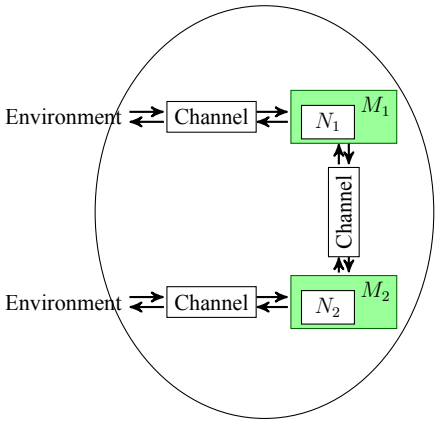


Figure 3.4: Distributed message-passing monitors

Next, we define what it means for a set of monitors to enforce a policy on a distributed system. Note that this definition covers both *centralized* and *decentralized* enforcement. In both cases, monitors are needed to be attached to nodes so that security-relevant actions are reported appropriately; in the first case, all but one nodes are *dummy* forwarding nodes that send all security relevant actions to a *central* monitor (i.e., location) in order to make enforcement decisions, whereas in the second case every monitor is capable of making enforcement decisions by itself, thus enforcing the policy in a *decentralized* manner.

Definition 16. *(Distributed Enforcement) Given a policy $\hat{P}$, a distributed system DS, and a set of monitors $\hat{M}$ such that the monitors are less or equal to the number of nodes in DS, we say that $\hat{P}$ is distributedly enforceable on DS by $\hat{M}$ if and only if there exist renaming functions $rename_i$, and hiding functions $hide_{i_{\Phi_i}}$ for some set of actions $\Phi_i \subseteq \Big(Output(M_i) \cup Output(rename(N_i))\Big)$, with $1 \leq i \leq |\hat{M}|$ such that:*

$$\Big(traces(MS) \mid acts(\hat{P})\Big) = traces(\hat{P}), \text{ where}$$

$$MS = \left(\prod_{N_i \in \widehat{MN}, M_i \in \hat{M}} hide_{i_{\Phi_i}} M_i \times rename_i(N_i)\right) \times \left(\prod_{N_i \in \widehat{UN}} N_i\right) \times (C).$$

We will refer to the monitor in Def. 15 as *global monitor*, and to the monitors in Def. 16 as *distributed monitors*. Fig. 3.3a depicts a distributed system, Fig. 3.3b depicts an instance of a

global monitor for that system, and Fig. 3.4 depicts an instance of distributed monitors.

3.4.2 Reduction to Decomposability

We mentioned in Section 3.3 that a goal of this chapter is to analyze the policies that are enforceable in distributed systems. Previous work that characterized the enforceable security policies by specific types of (centralized) monitors [27, 29, 30], typically followed the following four steps: (1) formally define security policies (typically as sets of traces), (2) define a formal model of monitors (e.g., security automata [27] or edit automata [30]), (3) define what it means for a monitor to enforce a security policy (e.g., soundness and transparency [27, 29, 30]), and (4) identify the constraints that a set of traces (i.e., a policy) must satisfy in order to be enforceable by the particular model of monitors.

This thesis follows a similar process. Previous sections have already covered the first three steps. However, for the last step we are taking a different approach: instead of *directly* characterizing the policies enforceable by a specific model of monitors, we characterize the global monitors (as defined in Def. 15) that can be decomposed to a set of monitors that can enforce (as defined in Def. 16) the same policies as the given global monitors. Pictorially, we are characterizing the monitors depicted in Fig. 3.3b that can be transformed into the monitors depicted in Fig. 3.4. We follow this approach for two main reasons:

1. Any characterization of enforceable policies by monitors in centralized systems (e.g., Sections 2.5 and 2.6, and [27, 29, 30]) is orthogonal to the characterization of enforceable policies in distributed systems. Intuitively, since centralized systems can simulate distributed systems, if something is not enforceable in centralized systems, then it is also not enforceable in distributed systems (we formalize this argument later in this section); on the other hand, there are policies that are enforceable in centralized systems that are not enforceable in distributed systems, since distributed systems might not be able to precisely simulate centralized systems (Section 3.1). Our approach to characterizing enforceable policies

through decomposing global monitors allows us to focus on exactly the details that make the second simulation impossible; thus, we avoid reformulating and reproving enforceability limits of centralized systems (e.g., computability [29, 30], and input-forgiveness – see Section 2.5.1), that, due to the first simulation, would also apply to distributed systems.

2. The constructive proofs of the decomposition theorems are essentially decomposition algorithms, which can be used to distribute a centrally specified policy over a distributed system. Previous work has introduced a large number of specification languages that can be used to express (centralized) security policies as centralized automata (i.e., global monitors) [103, 104, 105, 108, 121]. As we discussed, such work provides a simpler means to specifying security policies [1, 104, 119]. Our formal characterization of the global monitors that can be decomposed to distributed monitors, provides (through the proofs of the theorems) *provably correct* algorithms to decentralize policies expressible in the above languages. To the best of our knowledge, we are the first to provide formal proofs of correctness for decentralizing algorithms of security policies that can be specified in such expressive languages (i.e., arbitrary centralized state machines).

The following theorem formally expresses the correctness (soundness and completeness) of our approach.

Theorem 3.4.1. *A policy $\hat{P}$ is distributedly enforceable on a distributed system DS if and only if there exists a global monitor M such that:*

1. *$\hat{P}$ is globally enforceable on DS by M*
2. *there exist monitors $\{M\}_i = \{M_1, \ldots, M_n\}$, one for each node of the system, such that traces(M) = traces$(\Pi_i M_i)$*

Proof. Case: $\Rightarrow$ If a policy $\hat{P}$ is distributedly enforceable then by Def. 16 there exist monitors $M_1, \ldots, M_n$, renaming functions *rename*$_i$, and hiding functions *hide*$_{i_{\Phi_i}}$ for some set of actions

$\Phi_i \subseteq$ *Output*$(M_i) \cup$ *Output*(*rename*(N_i)), with $1 \leq i \leq |\hat{M}|$ such that:

$\Big($*traces*(*MS*) | *acts*$(\hat{P})\Big) =$ *traces*$(\hat{P})$, where *MS* is defined in equation (4).

Then, we can construct the global monitor by taking the composition of the monitors, and through associativity and commutativity of renaming and hiding [1], construct $M = \Pi_i M_i$. This construction meets both requirements.

Case: $\Leftarrow$ If there exists a monitor M that globally enforces $\hat{P}$ on *DS* then there exists a renaming function *rename* and a hiding function *hide*$_\Phi$ for some set of actions $\Phi \subseteq$ *Output*$(M) \cup$ *Output*(*rename*(*DS*)) such that:

$\Big($*traces*(*hide*$_\Phi$($M \times$ *rename*(*DS*))) | *acts*$(\hat{P})\Big) =$ *traces*$(\hat{P})$.

In addition we know that there exist monitors $\{M\}_i = \{M_1, \ldots, M_n\}$, one for each node of the system, such that *traces*$(M) =$ *traces*$(\Pi_i M_i)$. We can place $M_1, \ldots, M_n$ over $N_1, \ldots, N_n$, and partition the renaming function *rename* and hiding function *hide*$_\Phi$ to n *sub-functions*, one for each node N_i, based on the signature of the node. Then we can apply these sub-functions on N_i's to complete the proof.

□

The proof of Theorem 3.4.1 formalizes appropriate transformations between global and distributed monitors through algebraic manipulations of I/O automata.

3.4.3 Basic Decomposition

In Section 3.1 we mentioned that one approach to enforce security policies in distributed systems is to decentralize the enforcement decisions to individual nodes. This approach can help us achieve better fault-tolerance, communication efficiency, and computational efficiency. In decentralized architectures, security mechanisms are distributed over the network, collecting and analyzing data locally, independently of each other. Only when a node realizes that some of its data might be relevant to some other agents, it forwards the appropriate information to them, so that they can collaboratively identify the attack. These approaches are different from centralized approaches, where a single (centralized) node is identified as the enforcement decision point, and all other nodes blindly send data to the central node and wait to be told (by the central node) what action to take.

In this section we present two algorithms that decompose global monitors to a set of decentralized monitors that can be placed on the nodes of a distributed system. However, we make a very strong assumption: the distributed system has no underlying network architecture. This means that nodes communicate through shared actions: every time a nodes takes a step, i.e., it executes an action, this is *immediately* seen by all other nodes. This assumption can be though of as implying that nodes communicate through an *instant broadcast* communication medium. This assumption allows us to achieve two goals. First, we present decomposition algorithms and prove them correct without having to deal with complexities such as asynchronicity and communication channels. These algorithms can be seen as intermediate steps in a hierarchical step-wise process of decomposing a global monitor: we first decompose a global monitor to a set of distributed monitors that meet the interfaces of the nodes of a given network. Then, in later steps (i.e., more concrete implementations) we can deal with additional complexity (e.g., asynchronous networks in Section 3.4.4, synchronous networks in Section 3.5, etc.). Second, we identify certain fun-

damental conditions that decomposition algorithms must meet. Specifically, we show that the instant broadcast assumption is both necessary and sufficient for decomposing global monitors to distributed monitors that are placed on different physical locations. This means that when, in later sections, we consider more practical networks (e.g., asynchronous point-to-point), the algorithms that decompose global monitors *must* simulate the instant broadcast; otherwise the distributed monitors will not be able to enforce the same policy as the global monitor.

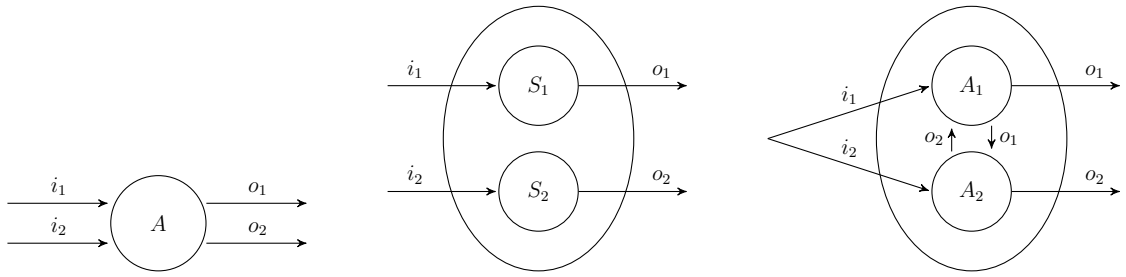
We assume that we are given an arbitrary I/O automaton A which represents the global monitor. An example automaton is depicted in Fig. 3.5a. We are also given a set of disjoint external[9] signatures S_i, one for each node i in the network, i.e., for all i and j such that $i \neq j$, $S_i \neq S_j$. The external signature S_i models the external interface of node i: if node i has input events I and output events O, then $S_i = I \cup O$. We also assume that the union of all the external signatures S_i is the complete signature of the I/O automaton A, i.e., $\mathit{Sig}(A) = \bigcup S_i$. Fig. 3.5b shows such an example of two signatures. The goal is to decompose A into a set of automata A_i, one for each signature S_i, such that the traces of the composition of A_i are the same as the traces of A. As we said in the introduction of this section, we assume that no renaming is allowed and there are no communication channels: automata communicate instantly through shared actions. The interface of such a goal decomposition is shown in Fig. 3.5c.

3.4.3.1 Deterministic Automata

In this section we assume that A is a deterministic automaton. This means that A has only one start state, and for every action π there is at most one π-transition from each state.

Note that non-determinism typically refers to state non-determinism: from a given state and for a given action there is more than one end state. Thus, our assumption that A is a deterministic

[9]The algorithms presented in this section work for internal actions as well: we can temporarily change internal actions to output actions, apply the transformations, and finally hide them to make them internal again.



(a) Centralized automaton (b) Signatures (c) Decomposed automaton

Figure 3.5: Decomposing centralized automaton given a signature

automaton refers to state determinism. However, in I/O automata there is another form of non-determinism: more than one local action can be enabled from a given state. Of course, in practice, the automaton can execute only one action of the possible ones. This non-determinism is useful when I/O automata are used as specifications (rather than implementations), describing what allowed behaviors a more refined automaton must meet. In a centralized automaton A this non-determinism is resolved by a scheduler (i.e., an external entity) that decides the step to be taken. The idea in the following decomposition algorithm is to *lift* the choices that a scheduler takes, from actions, to components. Thus, if one component is scheduled to take a local step, then that step is broadcasted and the rest of the components *skip* the actions they wanted to take, and move to the next state. Note that because of the assumption of determinism this state is unique for all automata, and thus no divergence is possible.

***DetDecomp* Algorithm:** We assume that we are given a (centralized) I/O automaton A and a set of signatures S_i that meet the constraints we described previously. The decomposition algorithm works by following the next steps:

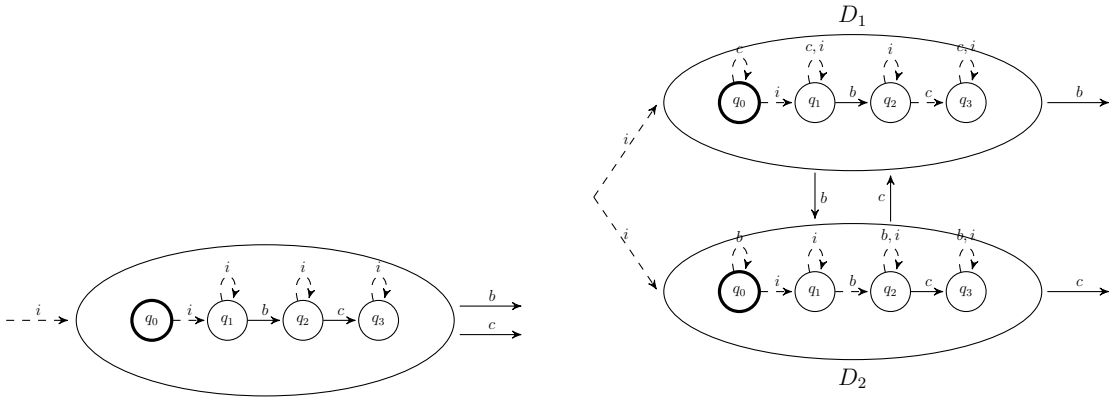
1. Extend each S_i by adding to the set of its input actions the output actions of every other node S_j, i.e., $\mathit{Input}(S_i) = \bigcup \mathit{Output}(S_j)$, $j \neq i$.
2. Extend each S_i by adding to the set of its input actions the input actions of every other node

S_j, i.e., *Input*$(S_i) = \bigcup$ *Input*(S_j), $j \neq i$.

3. Make n copies of A one for each node i. Consider the set $X_i =$ *Output*$(A) -$ *Output*(S_i). X_i contains the output actions of all nodes other than i, which after step 1 are input actions to node i. At each copy A_i, at each state where an action of X_i is not enabled, add a transition to the same state, i.e., a self-loop.

Step one, intuitively, ensures that every time that a node outputs something to the environment, it broadcasts it to the rest of the nodes as well. Notice that the broadcast, at this level of abstraction (i.e., under our assumptions), is instant. Step two ensures that the input from the environment is broadcast to all nodes. Thus, all nodes receive the same input sequence from the environment, and this happens at the same time at all nodes. Finally, the third step is a technical fix that ensures that the automata at each node are input-enabled: since we extended A's input actions at the first step, we have to ensure that these new input actions are enabled from each state of the automaton. Fig. 3.6 depicts the result of applying this transformation to a centralized automaton that outputs first a b and then a c, after receiving at-least one input i.

Essentially, the above decomposition algorithm allows all automata to take transitions in synchrony. By step two, inputs from the environment are received by all automata at the same time. By steps one and three, whenever an automaton takes a transition with an output action, that action is sent to the environment and also broadcasted at the same time to all other automata, which transition on the received action. The synchrony of stepping by decomposed automata is illustrated in Fig. 3.7 for the trace ib. In the beginning, automata are in the same state, as depicted in Fig. 3.6b. First, i is received by both automata and they move to state q_1 (depicted by bold arrows) in Fig. 3.7a. From there, the top automaton is given a chance to take a local action: it outputs b while it moves to state q_2. But this action is seen by both the environment and the bottom automaton, which also moves to state q_2, as depicted in Fig. 3.7b. Thus, the two automata remain synchronized: they are always in the same state that the original global automaton would



(a) Centralized automaton C (b) Decomposed automata D_1 and D_2

Dashed arrows: transitions with input actions
Solid arrows: transitions with output actions
Ellipse: automaton's interface
Circles: states
Bold circles: start states

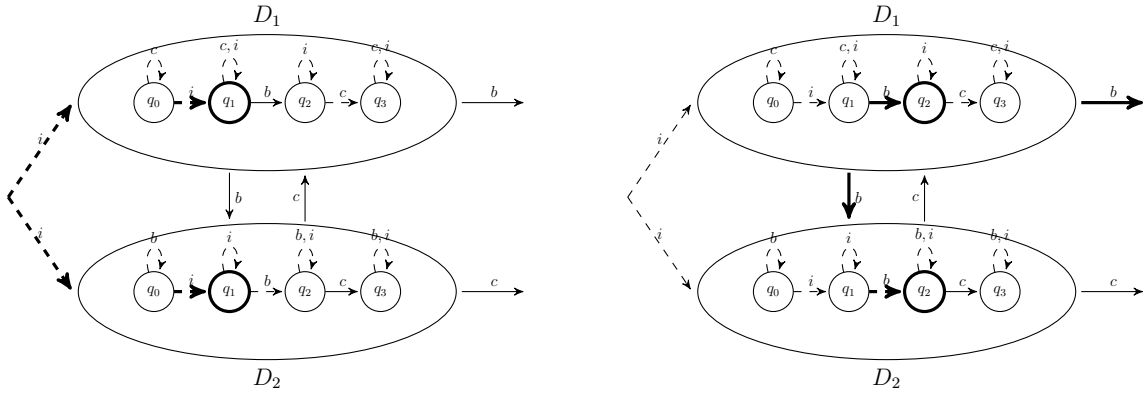
Figure 3.6: Applying *DetComp* algorithm to decompose centralized automaton C to automata D_1 and D_2

be when exhibiting the same trace ib (as shown in Fig. 3.6a).

Theorem 3.4.2. *Given an automaton A the* DetDecomp *algorithm produces a set of automata A_i, such that Sig$(A_i) = S_i$ and traces$(A) =$ traces(ΠA_i).*

Proof. The proof is by simulation. In the first case we give a simulation relation f from ΠA_i to A, and in the second case a simulation relation g from A to ΠA_i. Then the results follow from *Theorem 8.12* in [1] which says that if there is a simulation relation from an automaton A to an automaton B, then *traces*$(A) \subseteq$ *traces*(B). To show that f is a simulation relation from A to B we have to show that the following two conditions are met [1]:

1. If $s \in$ *start*(A), then $f(s) \cap$ *start*$(B) \neq \emptyset$.
2. If s is a reachable state of A, $u \in f(s)$ is a reachable state of B, and $(s, \pi, s') \in$ *trans*(A), then there is an execution fragment α of B starting with u and ending with some $u' \in f(s')$, such that *trace*$(\alpha) =$ *trace*(π).



(a) Step 1: Automata D_1 and D_2 receive input i and transition

(b) Step 2: D_1 transitions and exhibits b and D_2 transitions by receiving b

Dashed arrows: transitions with input actions
Solid arrows: transitions with output actions
Bold arrows: Action and transition the automaton takes *currently*
Ellipse: automaton's interface
Circles: states
Bold circles: states the automaton transitions to after taking the *current* transition

Figure 3.7: Decomposed automata D_1 and D_2 stepping through trace ib

Case 1: *traces*$(A) \supseteq$ *traces*(ΠA_i). If $q_{(\Pi A_i)} = \langle q_1, \ldots, q_n \rangle$ is a state of ΠA_i and q_A is a state of A we define $(q_{(\Pi A_i)}, q_A) \in f$ provided that q_A is the state such that $q_{(\Pi A_i)_i} = q_A$ for all i, i.e., $q_{(\Pi A_i)} = \langle q_A, \ldots, q_A \rangle$ (Note that this is well defined since $f \subseteq$ *states*$(A) \times$ *states*(B).)

Now we check the two conditions verifying that f is a simulation relation. For the first condition, since A has only one start state q_A, the constructed composed automaton will have only one start state, namely $\langle q_A, \ldots, q_A \rangle$. Thus, the first constraint is met.

Now, for the second condition, i.e., the step condition, suppose that $q_{(\Pi A_i)}$ is a reachable state of ΠA_i, $q_A \in f(q_{(\Pi A_i)})$ is a reachable state of A, and $(q_{(\Pi A_i)}, \pi, q'_{(\Pi A_i)})$ is a step of ΠA_i. Note that any *reachable* state $q_{(\Pi A_i)}$ of ΠA_i will be of the form $q_{(\Pi A_i)} = \langle q_A, \ldots, q_A \rangle$ for some state q_A of A. This can be proven by an inductive argument using the construction of the *DetComp* algorithm: the base case, i.e., the start state is easy to see. For the inductive argument, notice that by construction (and as we explained previously) all automata move to the same state regardless

of the action executed.

Now, we prove the step condition by considering cases based on the type of action performed:

1. π is an input action.

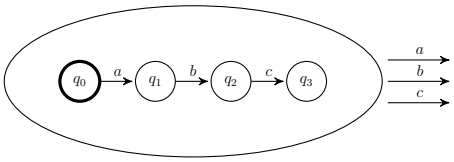
   Let $q_{(\Pi A_i)} = \langle q_A, \ldots, q_A \rangle$ (for the reason we explained above). Since π is broadcasted, and since every A_i is deterministic, all components will take the same transition, assume (q_A, π, q'_A), and end up in the same state q'_A. This is the same step that A takes from $q_A = f(q_{(\Pi A_i)})$ to q'_A, and thus the requirement that the traces exhibited by the two automata are equal is met, since *trace*$(\pi) =$ *trace*(π).

2. π is an output action.

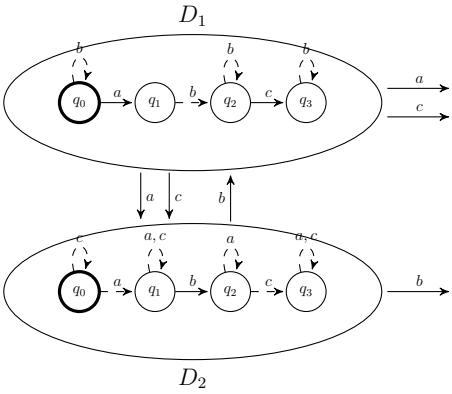
   Let $q_{(\Pi A_i)} = \langle q_A, \ldots, q_A \rangle$ and let π be the action of one of the components, say A_k. Then A_k will take the step (q_A, π, q'_A). But, by construction, π is also an input action of all other components, which will also move from their state q_A to q'_A. This is because the transition relation for every A_i is exactly the same; the only thing that differs is the type of some actions which for an A_i might be an output action, but for the rest is an input action. However, this typing of actions does not affect how the components transition from one state to another. Thus, ΠA_i moves from $s = \langle q, \ldots, q \rangle$ to $s' = \langle q', \ldots, q' \rangle$. A can also move from $q = f(s)$ to q' by taking a π step, which preserves the correspondance.

Case 2: *traces*$(A) \subseteq$ *traces*(ΠA_i). This case is similar to the previous, by taking g the inverse of f, i.e., if $q_{(\Pi A_i)} = \langle q_1, \ldots, q_n \rangle$ is a state of ΠA_i and q_A is a state of A we define $(q_A, q_{(\Pi A_i)}) \in g$ provided that $q_{(\Pi A_i)_i} = q_A$ for all i, i.e., $q_{(\Pi A_i)} = \langle q_A, \ldots, q_A \rangle$ (Note that, as with f, g is well defined because $g \subseteq$ *states*$(A) \times$ *states*(B).)

□



(a) Closed action-deterministic centralized automaton C



(b) Decomposed automata D_1 and D_2 for closed action-deterministic centralized automaton C and signatures $S_1 = \{a, c\}$ and $S_2 = \{b\}$

Dashed arrows: transitions with input actions
Solid arrows: transitions with output actions
Ellipse: automaton's interface
Circles: states
Bold circles: start states

Figure 3.8: Decomposing a closed action-deterministic centralized automaton

Closed action-deterministic automata. In the introduction of this section we mentioned the distinction between *state non-determinism* and *action non-determinism*. The first type of non-determinism refers to cases where an automaton may have multiple end states for a given state and action. The second type refers to a property that I/O automata have, namely that from a given state more than one local action might be enabled. In the *DetDecomp* algorithm we assumed that the centralized automata are state deterministic. However, if we strengthen the assumption of state determinism we can apply some simple optimizations in the *DetDecomp* algorithm. Specifically, we strengthen state determinism assumption in two ways: (1) we assume that the centralized automaton is *closed*, i.e., there are no inputs to the automaton: $\mathit{Input}(A) = \emptyset$; and (2) we assume that the centralized automaton is *action deterministic*, i.e., from each state there is at most one local action enabled. Although these two assumptions might seem very restrictive, they are important because they correspond to many policies and decentralization algorithms discussed in

the literature (e.g., serializable signatures [91, 92]). For instance, if we compose an automaton that models the monitored system, with an automaton that models the attacker, then the result composed automaton is a closed automaton (i.e., the first assumption is met). As we mentioned in Section 3.4.1 such a composition is useful when we have some formally stated assumptions about the attacker, such as the causality assumption from Section 3.3.

A closed action-deterministic automaton implies that there is no concurrency in the system. Thus, we do not need to worry about synchronizing the decomposed automata. Specifically, since, from each state q only one local action is enabled, say π, then only one automaton can take that step and move to some state q'. But this also holds for the (new) state q': only one automaton can transition from q' to some state q'' by taking an action, say π'. This means that the automaton that takes the first action, π, does not need to broadcast the action to all other automata; it only needs to notify the automaton that will take the next action, i.e., π'. This means that (1) less messages need to be exchanged, and (2) no concurrency-control algorithm is needed, since automata remain synchronized through simple notifications. An example decomposed closed action-deterministic automaton is depicted in Fig. 3.8. Note that whenever D_1 wants to output action a, then this action is only sent to D_2 (even if there were more automata in the system). Also, after action a is sent, D_1 will move from state q_1 only if D_2 takes the b step. This means that even if a was not simultaneously output to the environment *and* sent to D_2, we could still achieve the same behavior: D_1 sends a to the environment first, and then notifies D_2.

In Section 3.3.1 we assumed that the attacker can send inputs to multiple nodes at the same time (even though she blocks at each node until the node responds). This means that the closed action-determinism assumption does not hold, since we allow concurrency among the nodes of a (monitored) distributed system. Thus, the whole system can be modeled as a closed action-deterministic automaton only if the attacker restricts her activity to one action per step. For instance, attack #4 can be represented as a single straight-line program, but attack #5 must be

represented as two straight-line programs – the attacker can concurrently scan hosts for open ports and query the DNS server.

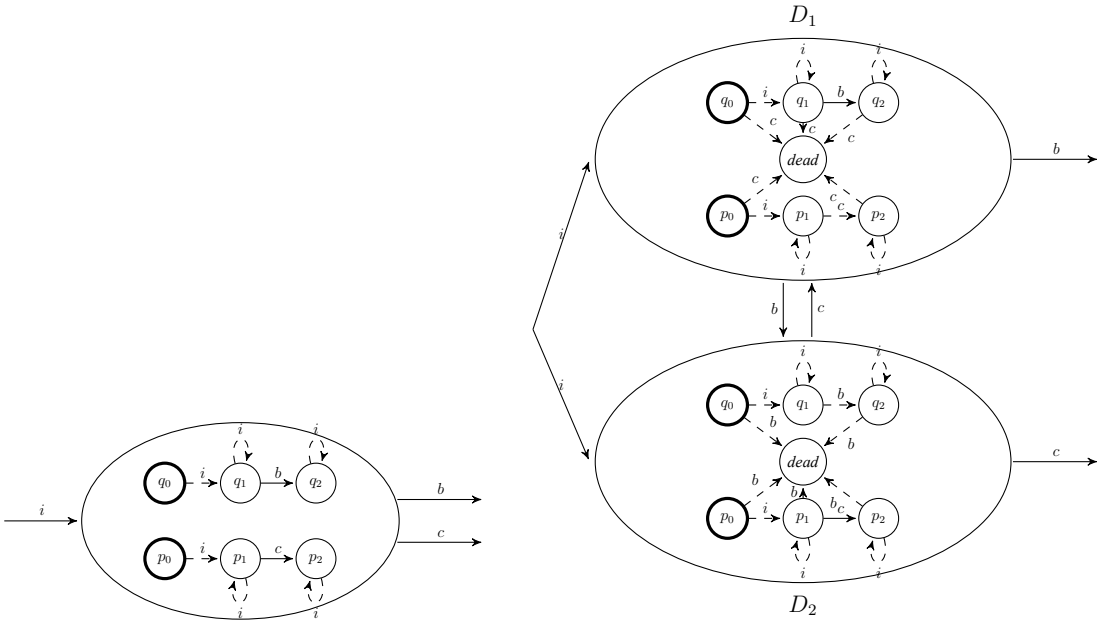
3.4.3.2 Non-Deterministic Automata

In the previous section we discussed a decomposition algorithm for deterministic centralized automata. However, as we mentioned in Section 3.4.3.1, I/O automata can be non-deterministic both in terms of states and local actions. Non-determinism in I/O automata is useful for declaring specifications (rather than implementations), i.e., describing what allowed behaviors a more refined automaton must meet. In the context of security, non-deterministic automata could be used, for instance, for collaborative detection of attacks among different organizations [122, 123]. For example, non-determinism could be used by an organization to describe (abstractly) an attack detected in its network, while hiding the details of its network architecture; this signature could then be shared with other organizations while ensuring that the signature (1) is applicable to network architectures of the other organizations (e.g., using appropriate decomposition algorithms like the ones we describe in this chapter) and (2) provides confidentiality to the organization sharing the information [122]. Another example where non-determinism can be useful is to specify multi-step attacks where the execution of a single step (i.e., an attack) can (non-deterministically) either succeed or fail. For instance, a non-deterministic signature could specify that the detection of an exploit for Microsoft IIS could lead to either a *compromised* state or an *ignore* state [109]. Whenever this signature is refined (i.e., applied) on a specific network this non-determinism can then be resolved based on the type of the web server the specific network uses: if the web server in the network is a Linux based server, then a Microsoft IIS exploit will be unsuccessful and thus the detection process will lead to a *unsuccessful* state. From this state the detection mechanism can decide to either not notify the security analyst or log the attack as *not severe*. This approach can greatly help in not distracting or overwhelming the analyst with alerts, which in turn can help

the analyst to prioritize and focus on other more severe or successful attacks [50, 109].

In this section, we present a decomposition algorithm that assumes that the centralized automaton is non-deterministic. This means that the automaton can have multiple start states, and each action can transition the automaton to potentially multiple states. The *DetDecomp* algorithm we presented in Section 3.4.3.1 does not work for non-deterministic automata: whenever one of the automata broadcasts the local action it took, the rest of the automata do not know to which state the automaton moved; thus the components of the transformed automaton might diverge, i.e., end up on different states. Note, that this is an issue only for local actions. If the automata diverge while (or before) receiving inputs then the automata are going to exhibit the same behavior regardless of the states they are in; this is because inputs are broadcast and automata are input-enabled and thus all components are going to receive (and exhibit) the same sequence of inputs.

As we discussed in Section 3.4.3.1 the idea behind the *DetDecomp* algorithm was to *lift* the scheduler from choosing local actions to choosing components. However, for non-deterministic automata we have to make additional guarantees because as we explained above, the components might end up in different states. Thus, we need to make sure that components do not diverge whenever a local action is broadcasted. The idea that we are going to use next is the following: instead of making sure that *every* component automaton is in the *same* state after every step (as we did in the *DetDecomp* algorithm), we will make sure that every automaton either is in a *possibly consistent* state, or it has *definitely diverged*, in which case it will be *sacrificed*. More specifically, each component automaton will be a (slightly modified) copy of the centralized non-deterministic automaton. To illustrate the key idea, let us assume that one automaton is chosen to execute a local action, and this action is broadcasted to the rest of the automata. If some of the recipient automata are at a state from which the local action they receive is not enabled in the (original) centralized automaton (i.e., if the original automaton was at the state the the recipient is then this



(a) Centralized non-deterministic automaton C (b) Decomposed automata D_1 and D_2

Dashed arrows: transitions with input actions
Solid arrows: transitions with output actions
Ellipse: automaton's interface
Circles: states
Bold circles: start states

Figure 3.9: Applying *NonDetDecomp* algorithm to decompose non-deterministic centralized automaton C to automata D_1 and D_2

local action would not have been enabled), then this means that at some point in their execution they diverged, and thus they are going to move to a state, from which they cannot take any more local actions (i.e., they sacrifice themselves for making a wrong choice in the past). Next we discuss more details, and the implementation, of this idea.

***NonDetDecomp* Algorithm:** We assume that we are given a (centralized) non-deterministic I/O automaton A, and a set of disjoint external signatures S_i, one for each node i in the network, i.e., for all i and j such that $i \neq j$, $S_i \neq S_j$. The external signature S_i models the external interface of node i: if node i has input events I and output events O, then $S_i = I \cup O$. We also assume

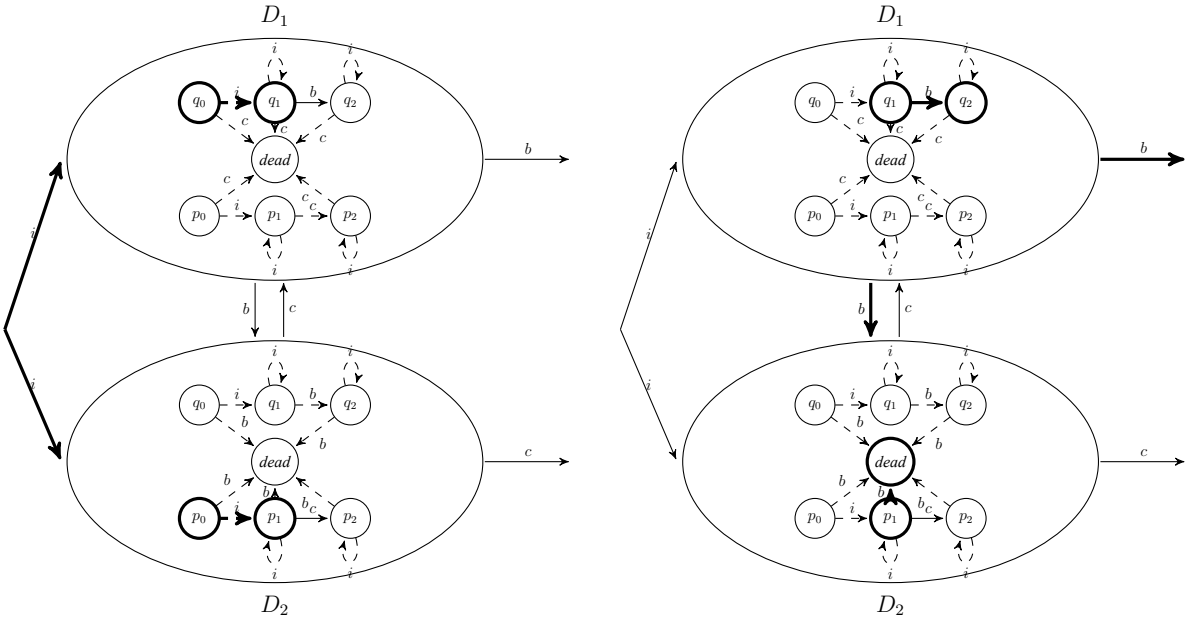
that the union of all the external signatures S_i is the complete signature of the I/O automaton A, i.e., $\mathit{Sig}(A) = \bigcup S_i$.

The goal is to decompose A into a set of automata A_i, one for each signature S_i such that the traces of the composition of A_i are the same as the traces of A. The decomposition algorithm works by following the next steps:

1. Extend each S_i by adding to the set of its input actions the output actions of every other node S_j, i.e., $\mathit{Input}(S_i) = \bigcup \mathit{Output}(S_j)$, $j \neq i$.
2. Extend each S_i by adding to the set of its input actions the input actions of every other node S_j, i.e., $\mathit{Input}(S_i) = \bigcup \mathit{Input}(S_j)$, $j \neq i$.
3. Make n copies of A, one for each node i and add a state, named *dead* to each A_i. Consider the set $X_i = \mathit{Output}(A) - \mathit{Output}(S_i)$. X_i contains the output actions of all nodes other than i, which after step 1 are input actions to node i. At each copy A_i, at each state where an action of X_i is not enabled, add a transition to the *dead* state.

Fig. 3.9 depicts the application of the above transformation to a global non-deterministic automaton A, which after receiving an input i it can either output a b or a c (depending on the start state it was in). Fig. 3.10 depicts how the decomposed automaton steps through the trace ib. In Fig. 3.10a the two automata start in different states, and after receiving i they end up in states q_1 and p_1, respectively. From there, if the top automaton is scheduled, depicted in Fig. 3.10b, then it steps to state q_2 while broadcasting b to both the environment and the bottom automaton. Finally, the bottom automaton instantly receives b and moves to the *dead* state. This way, it is blocked from outputting a c which would invalidate the specification of the global automaton (since the trace ibc cannot be produced by A).

Theorem 3.4.3. *Given an automaton A the* NonDetDecomp *algorithm produces a set of automata A_i, such that $\mathit{Sig}(A_i) = S_i$ and $\mathit{traces}(A) = \mathit{traces}(\Pi A_i)$.*



(a) Step 1: Automata D_1 and D_2 receive input i and transition

(b) D_1 transitions and exhibits b and D_2 transitions to the *dead* state after receiving b, since it had diverged (when it was at a different start state)

Dashed arrows: transitions with input actions
Solid arrows: transitions with output actions
Bold arrows: Action and transition the automaton takes *currently*
Ellipse: automaton's interface
Circles: states
Bold circles: beginning and end states in the automaton's *current* transition

Figure 3.10: Decomposed non-deterministic automata D_1 and D_2 stepping through trace ib

Proof. The proof is by simulation. In the first case we give a simulation relation f from ΠA_i to A, and in the second case a simulation relation g from A to ΠA_i. Then the results follow from *Theorem 8.12* in [1] which says that if there is a simulation relation from an automaton A to an automaton B, then $traces(A) \subseteq traces(B)$. To show that f is a simulation relation from A to B we have to show that the following two conditions are met [1]:

1. If $s \in start(A)$, then $f(s) \cap start(B) \neq \emptyset$.
2. If s is a reachable state of A, $u \in f(s)$ is a reachable state of B, and $(s, \pi, s') \in trans(A)$,

then there is an execution fragment α of B starting with u and ending with some $u' \in f(s')$, such that *trace*$(\alpha) =$ *trace*(π).

Case 1: *traces*$(A) \supseteq$ *traces*(ΠA_i). If $q_{(\Pi A_i)} = \langle q_1, \ldots, q_n \rangle$ is a state of ΠA_i and q_A is a state of A we define $(q_{(\Pi A_i)}, q_A) \in f$ provided that q_A appears in the n-tuple $q_{(\Pi A_i)}$, i.e., there exists i, $1 \leq i \leq n$, such that $q_i = q_A$. Note that the state $q_{(\Pi A_i)} = \langle q_1, \ldots, q_n \rangle$, where $q_i =$ *dead* for all i, is not a reachable state of ΠA_i. This can be proven by an inductive argument on the construction of the decomposition algorithm: the base case is trivial. For the inductive step, assume that there are some q_i's that are not equal to *dead* (which may be more than one). Then, if an input transition is taken, by construction, all q_i's will move to q_i''s that are also not *dead*. If a local transition is taken, then at least one q_i will transition to a q_i' that is not *dead*, namely the component that takes the local transition. Thus the claim is proven.

Now we check the two conditions verifying that f is a simulation relation. The start condition is trivial, since every start state $q_{(\Pi A_i)}$ of ΠA_i is a tuple that consists only of elements of *start*(A). Thus if $q_{(\Pi A_i)} \in$ *start*(ΠA_i) then $f(q_{(\Pi A_i)}) \cap$ *start*$(A) \neq \emptyset$, since, by definition of f, $f(q_{(\Pi A_i)})$ contains only elements of *start*(A).

Now, for the step condition, suppose that $q_{(\Pi A_i)}$ is a reachable state of ΠA_i, $q_A \in f(q_{(\Pi A_i)})$ is a reachable state of A, and $(q_{(\Pi A_i)}, \pi, q'_{(\Pi A_i)})$ is a step of ΠA_i. We consider cases based on the type of action performed:

1. π is an input action.

   Let $q_{(\Pi A_i)} = \langle q_1, \ldots, q_n \rangle$ and, without loss of generality, let $q_A = q_1$. Since π is broadcasted, all components in the composition will transition from state q_i to some state q_i'. This includes the first component that will transition from q_1 to some state q_1'. Thus the composed automaton will move to a state $q'_{(\Pi A_i)} = \langle q_1', \ldots, q_n' \rangle$. But since A is input enabled (as an I/O automaton) then its transition relation will contain the transition (q_1, π, q_1').

Thus, we have that $q_1 \in f(q_{(\Pi A_i)})$, $q'_1 \in f(q_{(\Pi A_i)})$ (with both q_1 and q'_1 reachable states), $(q_1, \pi, q'_1) \in$ *trans*(A), and thus $(q_1, \pi, q'_1) \in$ *execfragms*(A). But, *trace*$((q_1, \pi, q'_1)) = \pi = trace(\pi)$, thus this case holds.

2. π is an output action.

   Let $q_{(\Pi A_i)} = \langle q_1, \ldots, q_n \rangle$ and, without loss of generality, let π be an output action of component A_1. Since π is broadcasted to all other components A_i, $i \neq 1$, then by the third step of the *NonDetDecomp* algorithm there are two cases: first, if A_i originally had a π-transition enabled from the state that it is in, i.e., q_i, then (by construction) A_i it will move to the end state q'_i of that particular transition; second, if A_i did not have a π-transition enabled from the state that it is in, i.e., q_i, then (by construction) it will move to a *dead* state. Thus the composed automaton will move to some state $q'_{(\Pi A_i)} = \langle q'_1, \ldots, q'_n \rangle$, where some of the q'_i's, for $i \neq 1$, could be *dead* states. Note that the transition (q_1, π, q'_1) of component A_1 is a transition that the original automaton A could have taken. Moreover, by the definition of f, $q'_1 \in f(q'_{(\Pi A_i)})$ since $q'_1 \in$ *states*(A). Thus, we have that $q_1 \in f(q_{(\Pi A_i)})$, $q'_1 \in f(q_{(\Pi A_i)})$ (with both q_1 and q'_1 reachable states), $(q_1, \pi, q'_1) \in$ *trans*(A), and thus $(q_1, \pi, q'_1) \in$ *execfragms*(A). But, *trace*$((q_1, \pi, q'_1)) = \pi = trace(\pi)$, thus this case also holds.

Case 2: *traces*$(A) \subseteq$ *traces*(ΠA_i). This case is similar to the previous, by using as g the inverse of f, i.e., if $q_{(\Pi A_i)} = \langle q_1, \ldots, q_n \rangle$ is a state of ΠA_i and q_A is a state of A we define $(q_A, q_{(\Pi A_i)}) \in g$ provided that there exists i such that $q_{(\Pi A_i)_i} = q_A$ (Note that, as with f, g is well defined because $g \subseteq$ *states*$(A) \times$ *states*(B).) Intuitively, this case holds because of our inductive claim that the state $q_{(\Pi A_i)} = \langle q_1, \ldots, q_n \rangle$, where $q_i =$ *dead* for all i, is not a reachable state of ΠA_i. Thus, the simulation essentially holds because there is always a component A_i of ΠA_i that will be operational (i.e., not in a *dead* state) and, by construction, can behave as A. □

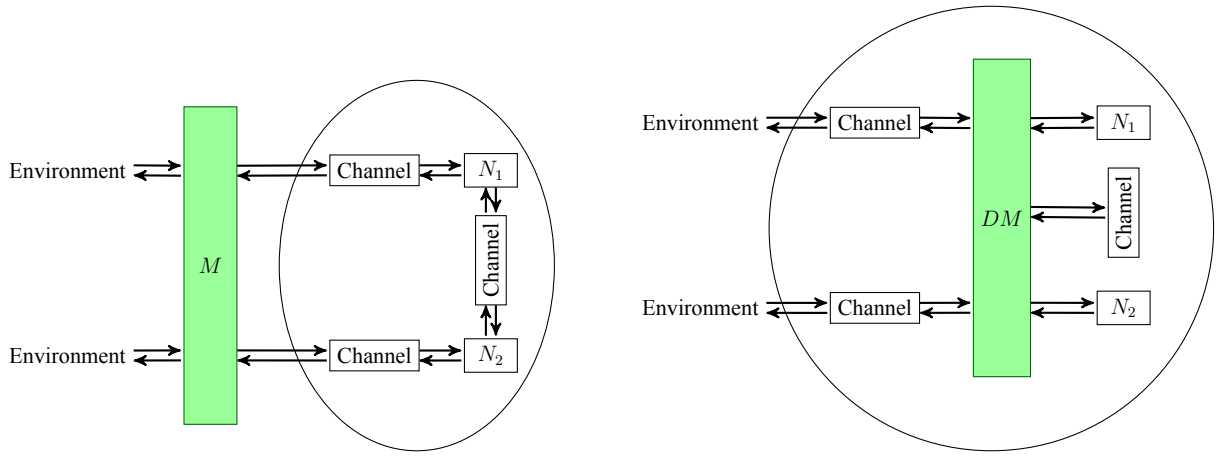
For the rest of this chapter, we will focus on deterministic automata (as they were formally defined in this section). However, the ideas that we will explore can be applied in non-deterministic cases, as long as the principles that we discuss in this section are taken into account, e.g., the use of dead states.

3.4.4 A Blueprint for Decomposition Algorithms

In Section 3.4.2 we showed how to reduce the problem of characterizing the enforceable policies in distributed systems to the problem of characterizing the global monitors that can be decomposed over a distributed system (Theorem 3.4.1). Then, in Section 3.4.3, we presented two algorithms that decompose global monitors to a set of decentralized monitors that can be placed on the nodes of a distributed system. The algorithms were proven correct under the assumption that all events in the system (inputs and outputs) are broadcasted and delivered instantly to all nodes. Pictorially, these algorithms can take us from global monitors of Fig. 3.3b to distributed monitors of Fig. 3.4. Although these algorithms provide us with some useful insight on how to decentralize global monitors, the strong assumptions about the network do not make them very useful in the general case of point-to-point asynchronous networks. In this section we will explore and identify the conditions under which such instant broadcast communication can be implemented (or simulated) in typical asynchronous message-passing systems.

We have broken down the process of decentralizing global monitors over distributed system into three high-level transformation steps. Although we could present a single algorithm that performs the transformation from a global monitor to distributed monitors. The three steps provide a blueprint that describes how to decentralize global monitors in an incremental and modular way. This approach has three main benefits:

1. It allows us to present the constraints in the decentralized algorithms in a simpler way.
2. It provides modularity in the final decentralized algorithm: each step requires to transform the monitor from a given model to another while maintaining some behavioral equivalence. Any algorithm that meets the given constraints can be used to implement this transformation step. Thus, different algorithms for each step can be combined together to create a larger number of decentralized algorithms.



(a) Original global monitor M

(b) Blueprint step 1: global monitor M transformed to distributed monitor *DM*

Figure 3.11: First transformation step of the blueprint for decomposition algorithms

3. It provides modularity in the proving of the correctness of a decentralized algorithm. If we substitute a component (i.e., implementation of a step) of a given algorithm with another implementation, it suffices to show that this new implementation meets the specification of the step it implements; the rest of the proof, for the other steps, remains the same. This can greatly simplify the proving process of decentralized algorithms.

The three steps of the blueprint are depicted in Fig. 3.12. The solid arrows show the order of the transformations between steps. The dashed arrows show the background, constraints, and algorithms that are discussed in each step.

Step 1: Global monitor to distributed monitor. In the first step we do not really make any changes to the monitor itself. Intuitively, in this step we push the global monitor inside the distributed system, transforming the global monitor into a component we call the *distributed monitor*[10]. Pictorially, we transform a global monitor M as shown in Fig. 3.11a to a distributed

[10]As we discussed in Section 3.4.2 the distributed monitor can be seen as composition of the distributed monitors (modulo I/O automata algebraic manipulations).

monitor *DM* as shown in Fig. 3.11b. The changes in this transformation are only syntactic: we just rename the actions of the monitor, e.g., the input actions from the environment (Fig. 3.11a) are renamed to input actions from the channels (Fig. 3.11b). Even though this step is trivial from a transformational perspective, it is very important from a characterization perspective. This step allows us to identify a key limitation of decomposing global monitors over a distributed system: we cannot always simulate a total order (centralized systems – global monitor) using a partial order (distributed systems – distributed monitor). More specifically, notice that in Fig. 3.11a the global monitor M can directly communicate with the environment. Thus, if the environment sends an event a through the top interface, and then an event b through the bottom interface, the monitor is going to receive the two events in exactly this order. However, as shown in Fig. 3.11b, the distributed monitor *DM* has to communicate through communication channels with the environment. Thus, for the same sequence of events, i.e., ab, if the bottom communication channel is slower than the top, then *DM* is going to receive the sequence ba, instead of ab. This means that *DM* can not be certain for the order that the environment sent the events.

Step 2: Distributed monitor to distributed shared memory monitor. In the second step, the high level idea is to consider the monitor (or parts of the monitor) as a shared variable (or multiple shared variables) in the shared memory model. In the shared memory model system, an I/O automaton contains a number of processes that (1) communicate with the environment and (2) directly access one or more shared variables. Thus, in this step we transform a monitor as the one shown in Fig. 3.11b to a monitor like the one in Fig. 3.13a. As we discussed in Section 3.2 the shared memory model is more complex than centralized models, since it allows for concurrency, but it is simpler than message passing models because it abstracts away from communication issues. After we transform the monitor from an I/O automaton to an I/O automaton of the shared memory model, we will have reduced the problem of decentralizing a monitor over a distributed

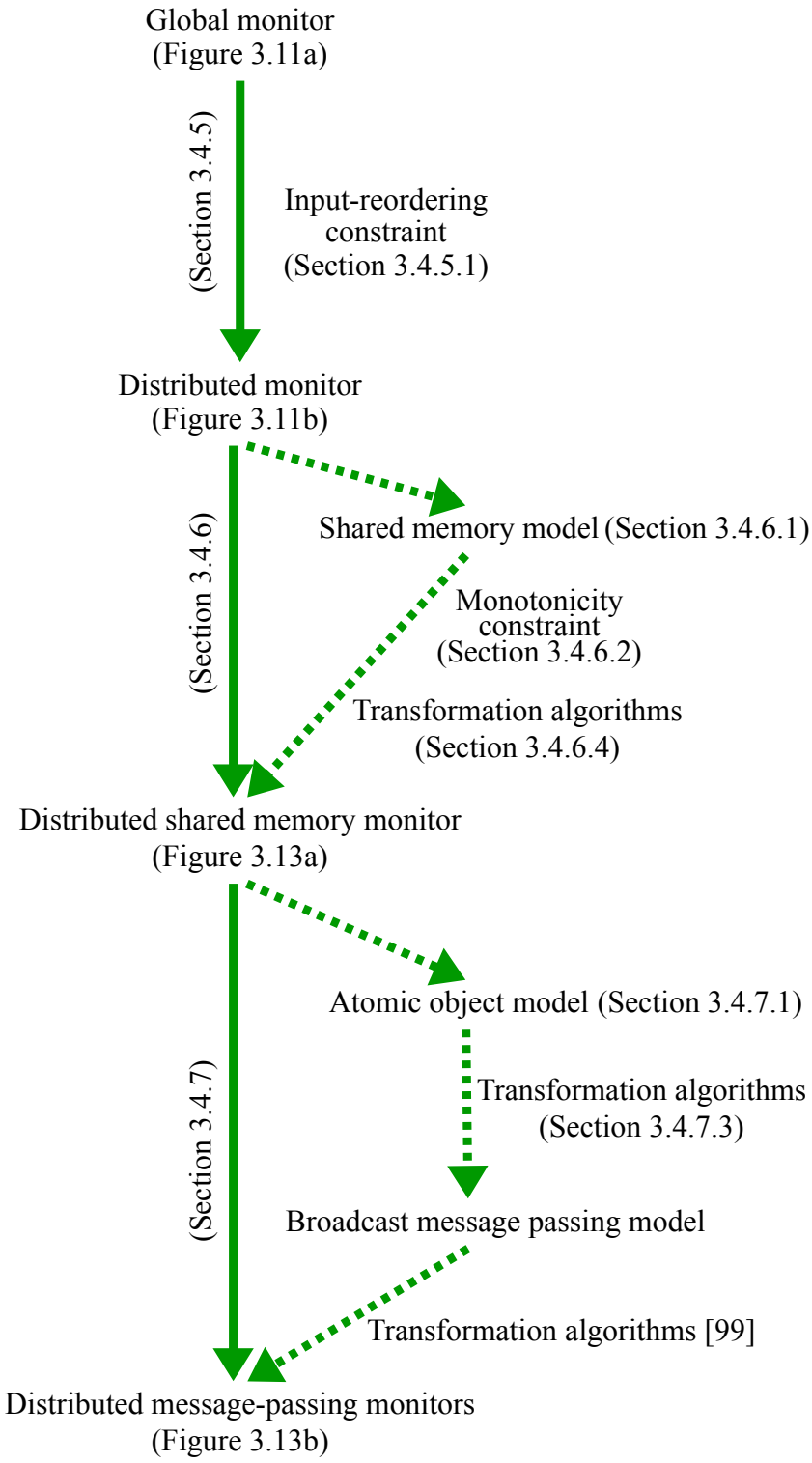
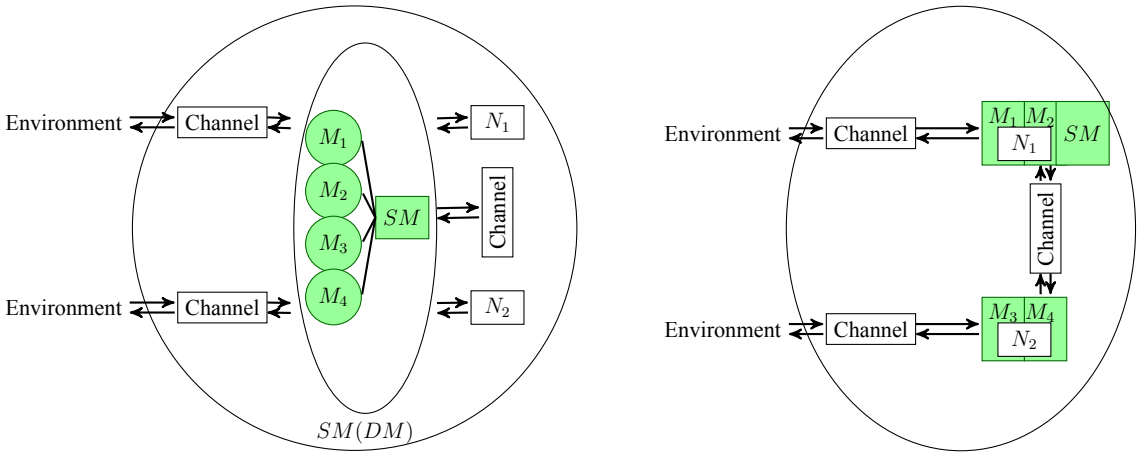


Figure 3.12: Blueprint – Steps to decentralize global monitor



(a) Blueprint step 2: distributed monitor *DM* transformed to distributed shared memory monitor *SM*

(b) Blueprint step 3: distributed shared memory monitor *SM* transformed to distributed message-passing monitors

Figure 3.13: Second and third transformation steps of the blueprint for decomposition algorithms

system to the problem of a simulating a shared variable in a message passing model (which we do in step 3 – Section 3.4.7). In order to keep the presentation simple, as we mentioned in Section 3.4.3, we are going to assume that the monitor is deterministic, and we will make comments about non-deterministic issues whenever necessary[11].

Although the first step of the blueprint, as we discussed previously, is more important from a characterization perspective than a transformational perspective, the second step is important from both perspectives. This is because in the second step the main part of the decomposition occurs. Essentially, we step from a centralized systems model to a distributed systems model.

From a characterization perspective, this means that we have to account for the differences between the two models (see Section 3.2), which includes the issue of *global knowledge*. For example, in Fig. 3.13a, if we assume that *SM* is the component that takes the enforcement decisions, then a decision taken at a given instance in time and pertains to what action M_1 should

[11] In general, non-deterministic monitors can be simulated by using the algorithms that we present in this thesis, and extending them by resolving the non-determinism through appropriate agreement protocols, where, for instance, the local monitors resolve the non-determinism by asking a dedicated component.

output, does not take into account any input that might have arrived in the meantime at another node, e.g., M_4. Thus, the decisions will always be made without the most *current* information (i.e., inputs received). We discuss this issue more in Section 3.4.6.2.

From a transformation perspective this second step of the blueprint is also important because the distributed monitor (from step 1) has to be transformed to a (decomposed) shared memory system monitor – a non-trivial step. In Section 3.4.6.4 we present two (novel) algorithms to achieve this transformation. The first one transforms the distributed monitor to a *centralized* shared memory system monitor, i.e., the enforcement decisions are taken at a central location (e.g., in Fig. 3.13a the decisions are taken by *SM* and the M_i's simply forward events they receive). The second algorithm transforms the distributed monitor to a *decentralized* shared memory system monitor, i.e., the enforcement decisions are taken by multiple components (e.g., in Fig. 3.13a the decisions are taken by the M_i's and *SM* acts as a *synchronization reference point*).

Step 3: Distributed shared memory monitor to distributed message-passing monitors. Finally, the goal of the third step is to simulate the shared variable(s) of the previous step over a message passing model. This is the final step of the transformation, where we transform a distributed shared memory monitor, as the one in Fig, 3.13a, to a decentralized monitor in the message passing model, as shown in Fig. 3.13b. There are a lot of algorithms that achieve this goal in the literature [1]. In this thesis, and for completeness purposes, we discuss an algorithm that places the shared variable on one of the nodes of the network, and every other node forwards requests to this dedicated node which, in turn, applies the requests to the shared variable and returns the responses to the appropriate senders [1]. In addition, we briefly mention some variations of this basic algorithm where variables are place on more than one nodes, such as the well-known *replicated state machine algorithm* that places a copy of the shared variable on every node and ensures that all copies are in consistent state through the use of logical time [1, 99, 124, 125].

These algorithms assume broadcast communication channels which does not meet our requirements for point-to-point communication. However, there are several algorithms[12] to simulate broadcast channels over point-to-point message passing models [99].

[12]Since these algorithms are not essential in the reading of this thesis, the interested reader can find more information in [99].

3.4.5 Transformation of Global Monitors to Distributed Monitors

In distributed systems the nodes do not (explicitly) control the actions they receive from the environment.This lack of control, combined with the fact that information takes time to propagate in distributed systems (see Section 3.2), can result in the inability of nodes to globally order the inputs they receive in the order that the environment sent them. For instance, the monitor in Fig. 3.11a can arrange the actions received through the two interfaces (i.e., the top and bottom environment) in the same order that they were sent: if the environment sends an input a through the top interface and then a second input b through the bottom interface then the monitor will know the order in which the environment sent the inputs. On the other hand, the monitor in Fig. 3.11b may not be able to infer the original order of inputs because of the intermediate communication channels: if the top channel is slower than the bottom then it is possible that the input from the bottom interface arrives first, i.e., the monitor will observe the sequence ba instead of the one originally sent, ab. So, in this case the monitor cannot know with certainty the order in which the environment sent the inputs. The fundamental issue is that the communication channels are two distinct entities and they cannot simulate the (atomic) monitor in Fig. 3.11a in terms of receipt of inputs. In this section we formalize this issue as the property of *input reordering*.

3.4.5.1 Input reordering automata

To look a little bit closer at this issue, let us consider a centralized I/O automaton A with inputs a and b and outputs x and y that exhibits the following behavior: if input a is received first, then the automaton outputs the sequence xy; else, if if input b is received first, then the automaton outputs the sequence yx (i.e., the reverse output sequence). The transition relation of such an automaton is depicted in Fig. 3.14. Dashed arrows represent transitions with input actions and concrete arrows represent transitions with output actions.

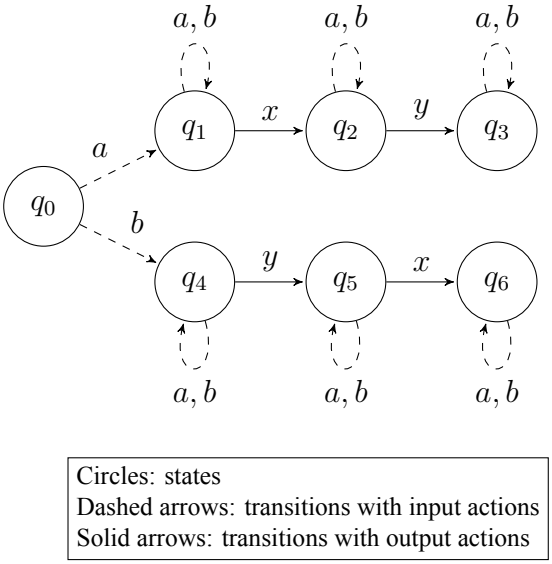
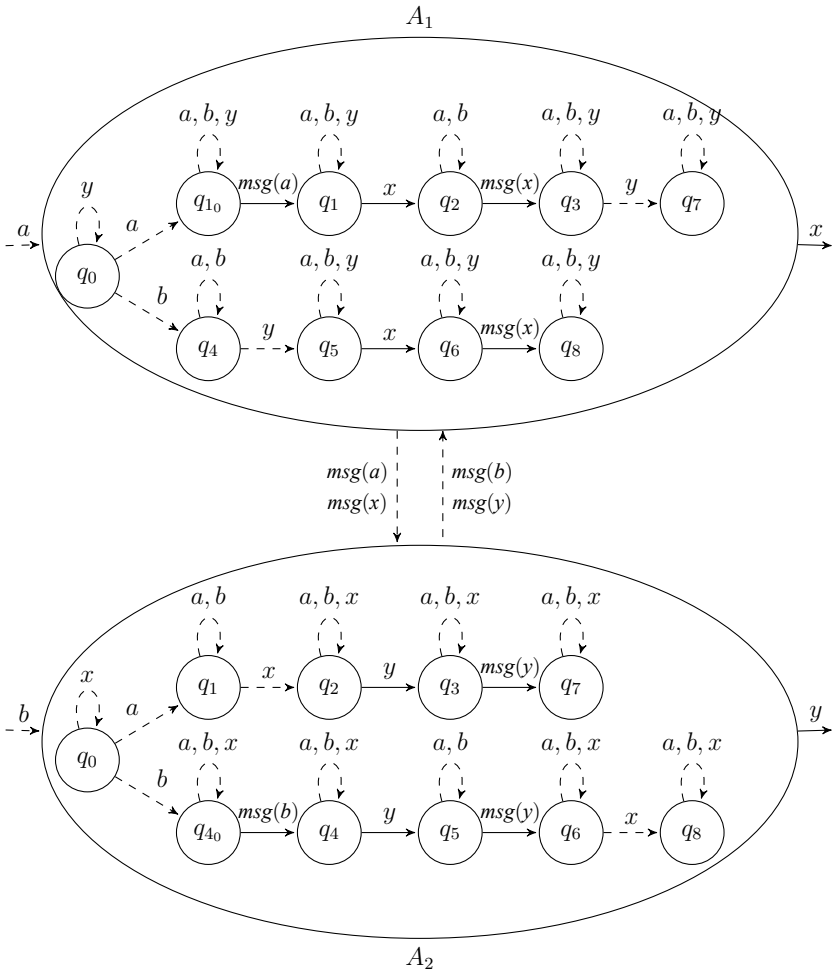


Figure 3.14: Transition relation of an automaton that is not input reordering

Suppose that we want to decompose this automaton to two automata A_1 and A_2 with signatures $\{a, x\}$ and $\{b, y\}$ respectively. Since the outputs of A must be output by different nodes, i.e., x by A_1 and y by A_2, A_1 and A_2 must coordinate in order to simulate the behavior of A and produce the correct sequences of outputs. In order to output the correct sequence of actions, A_1 and A_2 need to synchronize and agree on the order that the inputs were sent by the environment (since a and b are received by different nodes). One natural way to decompose A into A_1 and A_2 is to use a similar technique to the one that we used in Section 3.4.3: make two copies of the transition relation of Fig. 3.14, one for each automaton, and extend their transition relations with *communication* transitions that notify the other automaton (1) what input they received, and (2) what output action they have decided to exhibit. Fig. 3.15 shows one such decomposition. For instance, when A_1 receives an input a then it moves to state q_{1_0}. From that state it notifies A_2 by performing an internal communication action *msg*(a). A_2 receives this action[13] and moves to state q_1. Notice

[13]We have simplified the self-loops on each state as follows: input transitions that are labeled with a have the implicit label *msg*(a), and input transitions that are labeled with b have the implicit label *msg*(b). This was done to declutter the state-transition diagrams; it does not mean that a and b are broadcasted and received by both automata.



Ellipse: automaton's interface
Circles: states
Dashed arrows outside ellipses: input actions sent by the environment
Solid arrows outside ellipses: output actions sent to the environment
Dashed arrows between ellipses: actions for communication between automata
Dashed arrows inside ellipses: transitions with input actions
Solid arrows inside ellipses: transitions with output actions

Figure 3.15: Decomposed automata A_1 and A_2 of the automaton whose transition relation is depicted in Fig. 3.14

that A_2 has only input actions enabled in q_1 and the only way to move to state q_2 is to receive a $msg(x)$ action from A_1. This means that A_2 blocks until A_1 has output x. When A_1 outputs x it notifies A_2 which updates its state to q_2, from which it finally outputs y. This behavior meets the

specification of A which said that if a is received first then the automaton outputs the sequence xy.

The issue that we discussed above, i.e., the automata cannot infer the order in which the environment sent the inputs, can be illustrated by the following scenario. Let us assume that both automata are in their start states and consider the global state of the system, i.e., $\langle q_0, q_0 \rangle$. Moreover, let us assume that the environment sends actions a and b and while the environment sends the actions the automata do not get a chance to execute a local action. After A_1 receives a it moves to state q_{1_0} and the global state becomes $\langle q_{1_0}, q_0 \rangle$. Then, A_2 receives b and moves to state q_{4_0} and the global state becomes $\langle q_{1_0}, q_{4_0} \rangle$. Notice that from this global state both automata will notify each other while proceeding to exhibit their respective output action. Intuitively, A_1 thinks that a happened first, while A_2 thinks that b happened first. No matter what protocol or algorithm we try to design so that A_1 and A_2 infer the correct ordering of inputs we will run into the same problem: a and b happened *concurrently* from the perspective of the distributed system. Even if the automata coordinate to agree on a sequence (e.g., they agree that a happened first) this might not be the sequence that the environment actually sent. Note that the two automata *could* coordinate on what output sequence to produce. This means that if they agree to output xy then this is what the environment will receive. As we will see in Section 3.4.6.4 this will give the automata an opportunity to synchronize and coordinate (under specific assumptions about the environment).

The following definition formalizes the above intuition.

Definition 17. *(Input reordering) Given an automaton A and a partition S of the external actions of A, we say that A* is input reordering with respect to *S if and only if for every trace t of A, for all partitions S_i in S, and for every trace $t' \neq t$ of A such that:*

1. $t|S_i = t'|S_i$, *and*
2. $t|\mathit{Output}(A) = t'|\mathit{Output}(A)$,

the set of traces that extend t is equal to the set of traces that extend t'.

Constraint (1) says that the two traces must agree on the local ordering of events from each node's perspective. Constraint (2) says that the two traces must agree on the output actions of the global automaton. In other words, if we consider a given trace of the automaton and we consider all possible traces that we can construct by rearranging the actions of the trace while respecting (1) the ordering of actions that happen locally at each component, and (2) the global ordering of outputs, then the traces that extend each of these constructed traces must be the same set. For instance, let us consider two components with signatures $\{a, x\}$ and $\{b, y\}$, where x and y are output actions. Let us also consider the global trace of the two automata $t = abxy$. The local sequences of the components are ax and by, and the global output sequence is xy. Then the *valid* rearrangements according to Definition 17 are the traces:

1. $t_1 = axby$: the local orderings ax and by are preserved, and the global output ordering xy is also preserved, and
2. $t_2 = baxy$: the local orderings ax and by are preserved, and the global output ordering xy is also preserved.

Note that the sequence $byax$ is not a valid rearrangement. Although it maintains the local orderings by and ax, it violates the global output sequence ordering since y appears before x.

Definition 17 does not require that all local sequences are interleaved (i.e., both inputs and outputs). To illustrate why consider the following example: the centralized automaton A is exhibiting the following behavior: output x after you receive a, output y after your receive b, and output k if the output sequence is xy or output l if the output sequence is yx. Now let us assume that we want to decompose A into two automata with signatures $\{a, x, k\}$ and $\{b, y, l\}$. Notice that our example behavioral specification of A meets the requirements of the definition of input reordering: a and b can be interchanged and lead to same output sequences, but x and y cannot:

xy leads to xyk and yx leads to yxl. We do not require that the two automata's output actions are interleaved because the automata can execute an *agreement protocol* and output the correct sequence of events. A simple example agreement protocol is to have the automaton with the lowest id output first. In our example, when A_2 receives an input b it will ask A_1 if it has received any input actions. If A_1 responds positively then A_2 will back off, and wait for A_1 to output x. Thus the automata will output the sequence xyk, which is correct regardless of whether the input sequence was ab or ba. It A_1 responds negatively then A_1 will wait for A_2 to output y first, and the two automata will output the sequence yxl. Note that the output sequence yxl is also correct, even if A_1 receives an a while its negative response to A_2 is in transit. Thus, the nodes can agree on a sequence of outputs and exhibit the correct behavior of the specification of the centralized automaton A.

The following theorem formally expresses the fact that input reordering is a necessary condition for decomposing a single automaton over a distributed system.

Theorem 3.4.4. *Assume an automaton A with non-empty sets of input and output actions and a partition P of its actions (i.e., $P = P_1 \cup \ldots \cup P_n$, and $P_i \cap P_j = \emptyset$, $1 \leq i, j \leq n$, $i \neq j$) such that at least two of the partitions are non-empty. If there exist automata $A_1, \ldots, A_n$ whose signatures are equal to the signatures of the partitions (i.e., Sig$(A_i) =$ Sig(P_i), $1 \leq i \leq n$) and traces$(\Pi_i A_i) =$ traces(A), then A is input reordering with respect to P.*

Proof sketch. The proof is by contradiction. Assume that A is not input reordering with respect to P. Then this means that there are two traces t and t' of A and $\Pi_i A_i$, and (without loss of generality) two components S_1 and S_2 such that the inputs of t and t' from S_1 and S_2 are the same, the output sequences of $\Pi_i A_i$ are also the same for t and t', but the extensions of t and t' are not the same. Since I/O automata are input enabled, the last statement implies that, without loss of generality, t and t' must differ on some output action x (i.e., w.l.o.g., there exists an extension of t that contains x, but no extension of t' contains x). Notice that t and t' must have some input actions

of S_1 and S_2 in different order, otherwise they are the same trace. Since one of the traces leads to outputting x when the other does not then the two traces must go through a different state at some point (potentially, from the beginning if the automaton is non-deterministic). If the automaton is non-deterministic, then the two components must agree on their start state otherwise they might diverge and then the assumption about t and t' does not hold. Thus, the only other case is for the two traces to go on a different state due to an ordering of input actions of S_1 and S_2. But since we can construct an environment and a scheduler that moves both components with input actions at the same time, then the composed automaton (by definition of I/O automata composition) must move to the same state. Which means that the automata cannot distinguish between the ordering of the two input sequences. Thus, the trace extensions of the two automata are going to be the same, since they are going to be at the same state, which contradicts our assumption. Thus, A has to be input reordering with respect to P.

□

As an application of Theorem 3.4.4 consider figures Fig. 3.11a and Fig. 3.11b. Let (1) automaton A be the global monitor M in Fig. 3.11a, (2) automaton A_1 be the communication channel between M and N_1, (3) automaton A_2 be the communication channel between M and N_2, and (4) automaton A_3 be the distributed monitor *DM* in Fig. 3.11b. The inputs of A are the inputs of A_1 and A_2, and the outputs of A are the outputs of A_3. Thus, if the external behavior of the composition of A_1, A_2, and A_3 is the same as the behavior of A, then A is input reordering. Or, equivalently, if A is not input reordering, then there is no way for the composition of A_1, A_2, and A_3 to exhibit the same external behavior as A. In other words, if the global monitor in Fig. 3.11a is not input reordering then there is no distributed monitor (as the one depicted in Fig. 3.11b, i.e., with more than one intermediate communication channels) that can enforce the same policy. Thus, input reordering is a first characterization of the global monitors that can be decomposed over a distributed system. Moreover, since Definition 17 is *behavioral*, i.e., defined over traces,

input reordering is also a necessary condition for a policy to be enforceable over a distributed system (by the discussion in Section 3.4.2.

3.4.6 Transformation of Distributed Monitors to Distributed Shared Memory Monitors

In this section we show how to transform an I/O automaton A (e.g., Fig. 3.11b), which models a global monitor M, to a behaviorally equivalent I/O automaton $SM(A)$ in the shared memory model (e.g., Fig. 3.13a). We show that under certain constraints our transformation preserves the external behavior of A and thus $SM(A)$ can substitute for A in a monitored system that uses M as the monitor. As discussed in Section 3.4.4, one of the reasons we have broken down the process of decentralizing global monitors over distributed system into three high-level transformation steps is to allow for modularity in the construction of decomposition algorithms. In this section we will see the first instance of this benefit. Namely, we will present two different algorithms that transform A to two (different) I/O automata in the shared memory model: the first one corresponds to centralized enforcement scenarios, i.e., the enforcement decisions happen at a single node in the distributed system; the second one corresponds to de-centralized enforcement scenarios, where decisions are made at different nodes in the distributed system.

We begin with a short description of the asynchronous shared memory model that was originally introduced by Lynch [1] (Section 3.4.6.1). Then we describe one more fundamental limitation of distributed enforcement, monotonicity (Section 3.4.6.2). Finally, we describe the two algorithms that transform an I/O automaton that models a global monitor to an automaton in the shared memory system (Section 3.4.6.4). The reader familiar with the shared memory model can skip to Section 3.4.6.2, whereas the interested reader may refer to Lynch [1] for a more detailed presentation of the shared memory model.

3.4.6.1 Asynchronous Shared Memory

In this section we discuss shared memory systems, as they were originally defined by Lynch [1]. The familiar reader can skip this section; the interested reader can refer to Lynch [1] for a more detailed presentation.

An *asynchronous shared memory system* is modeled as an I/O automaton (we will use the meta-variable A to refer to a shared memory automaton) with specific structure [1]. The automaton consists of a (finite) set of n processes, where each process is a state machine (i.e., *not* an I/O automaton). Processes can interact with each other through a (finite) set of shared variables, and with the environment through a designated *port*, $port_i$, $1 \leq i \leq n$. The interactions of a shared memory automaton are depicted in Fig. 3.16.

Each process i has a set of *states*, $states_i$, and a subset $start_i$, of *start states*. Each shared variable x in the system has an associated set of *values*, $values_x$, with a subset $intial_x$ of *initial values*. Each state of the system automaton consists of a pair $(states_i, values_x)$, for each process i and shared variable x. Start states are defined similarly.

Each action of the automaton A is associated with a process i. The only input and output actions of each process i (and A's) are the ones that are associated with the process's port (i.e., the actions that are used for communication between the environment and the process through the corresponding port). The rest of the actions of each process i (and A's) are internal actions. Some of these internal actions may be used for interacting with a shared variable; the rest are used for local computation.

Transitions that contain internal actions π of a process i that are associated with local computation involve *only* the state of i, i.e., such transitions are triples of the form $((s, v), \pi, (s', v))$, where $s, s' \in states_i$, and v is any value that a shared variable x can have. Otherwise, if a transition contains an action π used to access shared variable x, then only the state of i and value of x

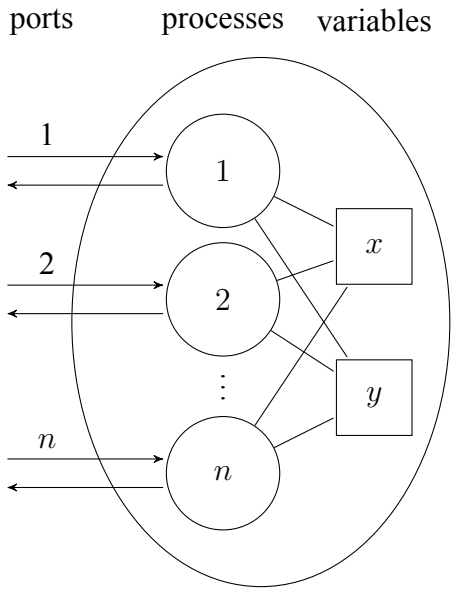


Figure 3.16: An asynchronous shared memory system (Diagram adopted from Lynch [1])

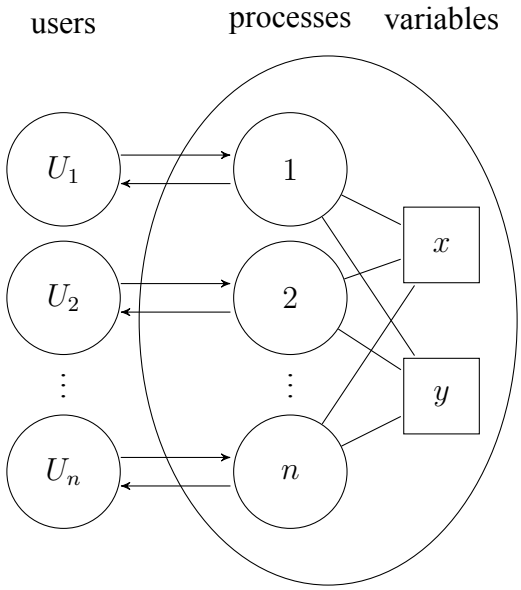


Figure 3.17: Users and shared memory system (Diagram adopted from Lynch [1])

are involved in the transition. Whether π is enabled depends *only* on the state of i and not on the value of x.

Environment. Similarly to Section 3.4.1, sometimes it will be useful to model the environment as an I/O automaton. In particular, we will assume that each process i of the automaton A interacts through its port with a user automaton U_i, as depicted in Fig. 3.17. For instance, consider attack #4 in Section 3.3: the attacker first scans the network to identify running web servers, and once she has the results of the scan, she sends malicious HTTP traffic to the appropriate IP addresses. This causal behavior of the attacker, i.e., that the exploitation phase precedes, and depends on the results of, the scanning phase can be modeled as follows. The attacker is modeled as an I/O automaton (e.g., U_1 in Fig. 3.17) with output actions scan and exploit(IP), input actions scan_result(IP), and a transition relation that blocks after executing the scan action, waits until the input action scan_result(IP) is received, and finally proceeds by outputting the action exploit(IP) (if a vulnerable IP is received). Because the users U_i and A are I/O automata, we can compute their composition and reason about the behavior of the interaction. This will be useful, as we will see next, to model assumptions about the causal behavior of the attackers, and reason about the correctness of decomposition algorithms.

Shared Variable Types [1]. The definition of a shared memory system allows a process to access variables in arbitrary ways. However, typically, in practice, variables can be accessed only through a pre-specified interface of invocations and responses [1]. In order to restrict the operations through which a shared variable can be accessed, the notion of a *variable type* is introduced. A *variable type* consists of [1]:

1. a set V of values,
2. an initial value $v_0 \in V$,
3. a set of *invocations*,
4. a set of *responses*, and

5. a function f : *invocations* $\times V \rightarrow$ *responses* $\times V$.

When it is said that a shared variable x in a shared memory system A is of a given variable type, it means that [1]:

1. $\mathit{values}_x = V$,
2. $\mathit{initial}_x = \{v_0\}$, and
3. all the transitions involving x must be describable in terms of the invocations and responses allowed by the type.

In this thesis, we will use three types of variable types for describing decomposition algorithms: (1) read/write shared variables (with values V, invocations *read* and *write*$(v), v \in V$, and responses $v \in V$ and *ack*); (2) read-modify-write shared variables (with a single *read-modify-write* operation); and (3) shared variables where f is some arbitrary function. The first two variable types will be useful to model the state of a centralized monitor as a shared variable (i.e., shared state). After the centralized monitor is decomposed to local monitors, each local monitor will use the shared variable as a (shared) global state. The third type will be used to model the centralized monitor itself as the shared variable (since the monitor is essentially a function mapping inputs to outputs), and then by using algorithms that simulate shared memory systems over networks, the monitored distributed system will be simulating the centralized monitor. Next, we describe how read/write and read-modify-write shared variables are expressed using variable types (as they were originally described by Lynch [1]).

It is instructive to see how an automaton's transition can be though of as an access to a read-modify-write variable (this will be one of the insights that will guide one of our decomposition algorithms). A transition of an automaton is a tripple (s, π, s'), denoting that the automaton from a state s will execute action π and move to the new state s'. If we think the states of the automaton as a shared variable, then the transition of the automaton can be though of as *reading* the shared

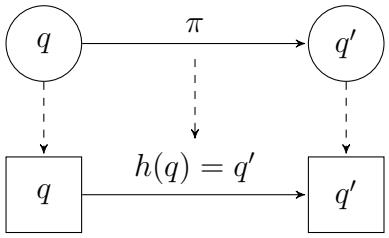


Figure 3.18: State transitions as accesses to read-modify-write variables

variable/state (i.e., s), modifying the state to s' using a function h (i.e., π), and writing the result of $h(s) = s'$, to the shared variable/state (i.e., s'). Fig. 3.18 depicts this relationship.

Executions and traces of variable types [1]. *Executions* of a variable type are defined as finite sequences $v_0, a_1, b_1, v_1, a_2, b_2, v_2, \ldots, v_r$ or infinite sequences $v_0, a_1, b_1, v_1, a_2, b_2, v_2, \ldots$, where: (1) v's are the values in V, (2) v_0 is the initial vale of the variable type, (3) the a's are invocations, (4) the b's are responses, and (5) the quadruples $v_k, a_{k+1}, b_{k+1}, v_{k+1}$ satisfy the function of the type, i.e., $(b_{k+1}, v_{k+1}) = f(a_{k+1}, v_k)$.

3.4.6.2 Monotonicity

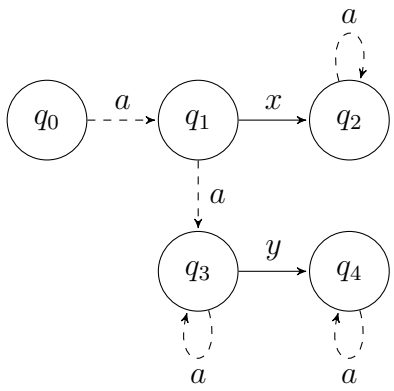
In Section 3.4.5 we presented one of the fundamental limitations of distributed systems: a system does not (explicitly) control the actions it receives from the environment.This limitation guided us to identify one of the first constraints that global monitors must meet in order to be decomposable over a distributed system: input reordering.

In this section we deal with another fundamental limitation of distributed systems, *global knowledge* (see Section 3.2): even though a centralized monitor has immediate access to all global information that it needs to make a decision, e.g., output an action, distributed monitors do not.

For example, consider an automaton A whose transition relation is depicted in Fig 3.19. The automaton can output x if it receives exactly one a, but if it receives more than one a before it is given a chance to take a local action, then A will output y. Essentially, the information and knowledge about inputs received travels *instantly* in centralized systems and is immediately available. This, however might not be the case in distributed systems. Assume for example, that we need to decompose this automaton to two automata A_1 and A_2 with signatures $\{a\}$ and $\{x, y\}$ respectively (i.e., A_1 is responsible for input, and A_2 is responsible for output). Such a decomposition is shown in Fig 3.20. Note that whenever A_1[14] receives an a it needs to forward this information to A_2 through a message *msg*(a).

The problem in this example is that the automata A_1 and A_2 in Fig. 3.20 transition independently. To be more precise, there is an external scheduler, outside the control of A_1 and A_2, that decides which automaton's turn it is. So if A_1 is scheduled first to execute an action then if it receives an a it will transition to q_1. From there, assume that the scheduler schedules A_1 again. This means that A_1 will output *msg*(a), essentially propagating the information that an a

[14]Notice, that A_1 has redundant states, since all it does is send message *msg*(a) every time it receives an a from the environment. However, we wanted to illustrate the relation between A_1 and A_2 as they move through states.



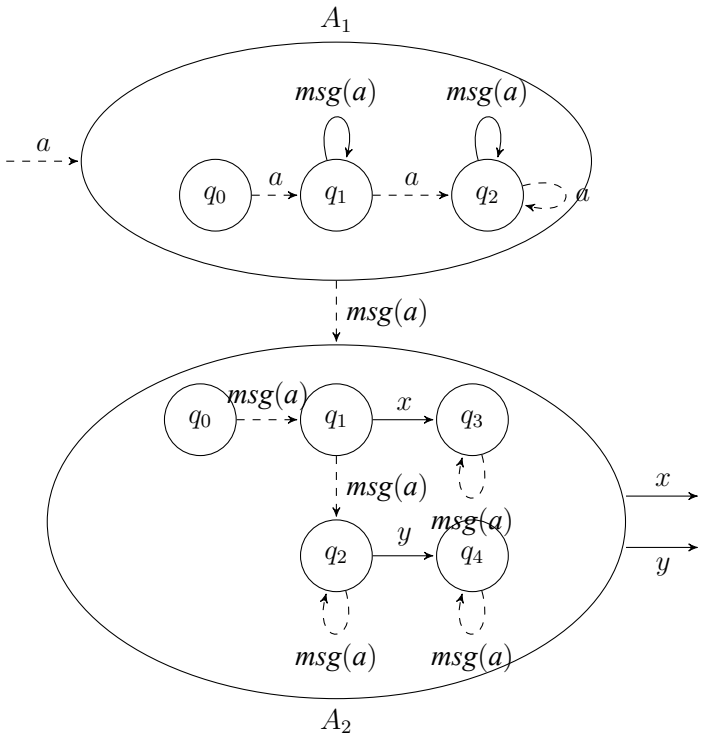
Circles: states
Dashed arrows: transitions with input actions
Solid arrows: transitions with output actions

Figure 3.19: Transition relation of a centralized automaton A that is not monotone

was received to A_2[15]. After A_2 receives the message it will be in state q_1. Note that the trace so far, i.e. the sequence of external actions, is a. At this point, if A_2 is scheduled it will output x, by transitioning to q_3, and the trace of the system will be ax. This trace is a valid trace of A in Fig. 3.19. However, if, while A_2 is in state q_1, A_1 is scheduled and A_1 receives a second a then it will move to state q_2 and the observed trace will be aa. If A_2 is scheduled again, since it is still in state q_1, it will output x and the observable trace will be aax. But this is not a valid trace of A; it should have been, aay. This means that if A_1 and A_2 were decomposing a central IDS A then they would either miss an attack or misclassify a correct behavior, as compared to A.

The problem is that since inputs are not broadcasted atomically, as we assumed in Section 3.4.3, inconsistencies might arise like the one above. Note that this problem is fundamental because we require that the set of traces of the centralized monitor is equal to the set of traces of the decentralized monitors (Sections 2.4.1 and 3.4.1.3). More concretely, A_2 cannot wait in q_1 to see if more input will arrive because if no input arrives then A_2 will not output anything (the

[15]We assume that the message is received instantly, even though a more appropriate modeling would include some communication channel. However, even without communication channels the argument still holds.



Ellipse: automaton's interface
Circles: states
Dashed arrows outside ellipses: input actions received by the environment
Solid arrows outside ellipses: output actions sent to the environment
Dashed arrows between ellipses: actions for communication between automata
Dashed arrows inside ellipses: transitions with input actions
Solid arrows inside ellipses: transitions with output actions

Figure 3.20: Decomposed automata A_1 and A_2 that attempt to simulate automaton A depicted in Fig. 3.19

same holds if A_1 was waiting before sending a message – it does not know if the environment is done sending inputs). In addition, A_1 cannot send notifications saying that I do not have more input because by the time this message is sent, more input might arrive. This is very similar to the situation we faced in Section 3.3 and example #5: if the webserver receives an HTTP request and the DNS server is unavailable (or responds extremely slowly) then either the webserver will have to make a false positive (by either not responding until the DNS server is online again, or choosing to fail-safe and treat the request as malicious – in both cases, a denial of service to a legitimate user), or a false negative (by assuming that the request is valid). The three (fundamental) problems here are: (1) information takes time to travel, (2) global knowledge is not immediately accessible, and (3) the environment cannot be trusted.

This leads us to the definition of *monotonicity*. Monotonicity formally captures the situation described above. It essentially says that whenever a local node is at a state that it can commit to an output (based on input it has already received), e.g., x in our previous example, then this output is still valid even if more inputs arrive in the future (i.e., it cannot be invalidated by future inputs). This is very similar to the notion of monotone functions where, for example, if f is a monotone function and $f(a) = x$ then for all extensions t of a, x is a prefix of $f(t)$, e.g., $f(aa) = xx$ is valid output but $f(aa) = y$ is not.

In our case, however, we cannot use the previous definition of monotonicity directly. The problem is that *locally* the automaton can be non-monotone, as we saw in Fig. 3.19. Local information is always instantly accesible. Thus, our definition of monotonicity should capture this scenario, by allowing non-monotone transitions that are local but prohibiting non-monotone transitions for remote input[16]. More formally, given an automaton A and a partition S of the actions

[16]The situation is a little bit more complicated. Specifically, the global monitor in Fig. 3.11a receives actions from the environment instantly and it receives actions from the nodes of the distributed system through channels. Whenever we push the monitor inside the distributed system (Section 3.4.5), the situation is reversed: the monitor can receive actions from the nodes of the system instantly but it receives actions from the environment through the channels. This means that although a global monitor can be non-monotone w.r.t. the environment, the decentralized

of A we say that *A is monotone with respect to S* if and only if for all partitions S_i in S, if we are given any trace t of A then there does not exist a trace u of A such that:

1. $u|(S - S_i) = (t|(S - S_i)); ((\textit{Input}(S - S_i))^*)$, where the set $(S - S_i)$ denotes the actions of the system excluding the actions of node i; the equality says that (with respect to remote actions of node i) u is equal to t extended by remote inputs,
2. $t|\textit{Input}(S_i) = u|\textit{Input}(S_i)$, i.e., t and u have the same inputs from S_i, and
3. $t|\textit{Output}(S_i) \not\preceq u|\textit{Output}(S_i)$, i.e., the outputs of u that belong to S_i are not a prefix of the outputs of t that belong to S_i.

In constraint (1) we do not require that u is an extension with respect to outputs of t as well because nodes have control over their outputs and thus they could synchronize appropriately. For example, if one node has actions $\{a, k, l\}$, where a is input and k, l are outputs, and another node has only output actions $\{x, y, z, w\}$, then the traces akx and $aaly$, can be decomposed, because the non-monotone behavior is local to the first node (i.e., on a output k, on aa output l instead), which can then notify appropriately the other node. That is, on k, send a message *msg*(k) which will result to x, and on l, send a message *msg*(l) which will result to y. However, if we had two additional required traces, $aalaz$, and $aalaaw$, then the problem we discussed previously arises.

The following theorem formally characterized the above informal discussion.

Theorem 3.4.5. *Given an automaton A with non-empty sets of input and output action and a partition of its actions $P = P_1 \cup \ldots \cup P_n$ such that at least two of the partitions are non-empty, then*

monitor cannot. To be more specific, using the example of Fig. 3.19, when the monitor receives inputs from the environment it can exhibit the correct behavior ax, and aay. But when the monitor receives the inputs from communication channels then it might output x upon receiving a, but it could be that the second a is in transit; i.e., the observed input is aa, and the observed output will be x – and this violates the specification. However, this is more of a technical issue. For instance, we can solve this problem by assuming that some components of the decentralized monitor are placed directly between the environment and the communication channels. Another option is to assume that the monitor never communicates instantly with nodes; there is always some channel between them. This means that we assume that monitors are *globally monotone*, i.e., no component is non-monotone. Although next we give the general definition of monotonicity, for the rest of this chapter we will assume that all monitors are globally monotone in order to present the rest of the results in a simpler way.

if there exist automata $A_1, \ldots, A_n$ *whose signatures match the partitions and traces*$(\Pi_i A_i) =$ *traces*(A) *then* A *is monotone with respect to* P.

Proof. The proof is by contradiction. Assume that A is not monotone and such automata $A_1, \ldots, A_n$ exist. Without loss of generality assume that we have two partitions: the first contains input actions, and the second contains output actions.

Take any state q of A, where (1) an input and an output transition lead to different states (e.g., producing actions a and x respectively), (2) the input belongs to an A_i and output belongs to an A_j, $i \neq j$, and (3) from the state that the input transition leads, another output of A_j is enabled, say y. Such a state exists because we assumed that A is not monotone.

Since the sets of traces are equal, there exist two simulation relations (i.e., a bisimulation) that relate states of $\Pi_i A_i$ with A and state of A with $\Pi_i A_i$. Now let A_j be at the state that the local output action is enabled (i.e., the corresponding global state of A is q). From that state, either A_j must block waiting for more input from A_i, in which case A can produce a trace that includes x, but $\Pi_i A_i$ will not. On the other hand, if A_j takes the transition to output x we can always construct an environment and a scheduler that will schedule A_i and send one more input a, before A_j has a chance to output. Thus, $\Pi_i A_i$ will produce a trace that A will not. We can see that in both cases the equality of sets of traces does not hold and thus A has to be monotone. □

Notice that if the set of inputs of A is empty, then the automaton is trivially monotone. This justifies the simplicity of the closed action-deterministic automata decomposition algorithm we discussed in Section 3.4.3.1.

From a security perspective, the theorem says that a centrally specified security policy can be enforced in a distributed system *only if* it is monotone. If the policy is not monotone then we need to either output less traces or more traces than the centrally specified policy. If we output less traces then we will either miss some attacks (by not outputting alerts) or result in a denial of

service situation (if the user does not receive a valid response). On the other hand, if we allow more traces then either we will allow some attacks to happen or we will admit that some of the responses we send to the users might not be correct.

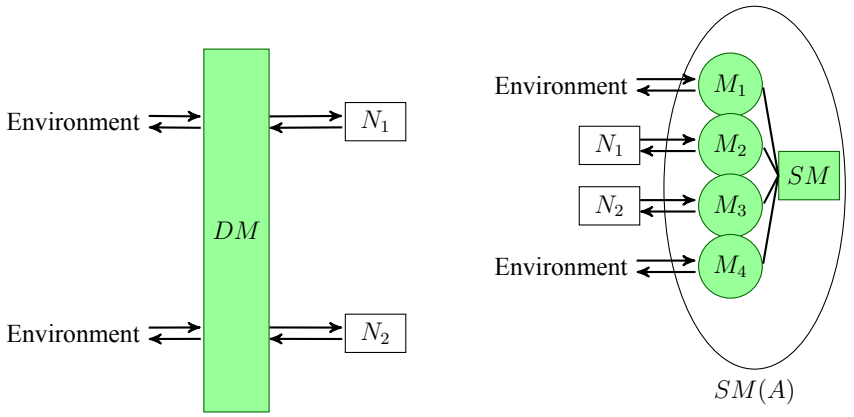
Monotonicity formally characterizes the design choices of solutions that have been introduced in previous work of distributed intrusion detection [91, 92]. More specifically, in previous work distributed attacks are specified as transition graphs [91, 92]. The nodes of the graph are distinguished into two types: positive and negative. Positive nodes represent events that are necessary for the attack to take place. On the other hand, negative nodes represent those events which cancel out positive events. Specifically, while events are received by each monitoring component the component goes through the transition graph. If the component reaches a positive node then it raises an alarm. However, if additional events arrive, after the component has reached a positive node, it is possible for the transition system to move from a positive node to a negative node. When reaching to a negative node the component marks the previous alarm as a false alarm. As the authors describe, enforcement components might reach a state where they cannot be sure if an attack has happened or not (e.g., because not all security relevant actions have been received). In this case, if the corresponding component does not take action there is a risk of more damage to the system. Thus, the solution the authors recommend is to take action as if the attack is real and if later it is realized that the observed events did not constitute an attack then a notification is generated indicating that the previous alarm was a false alarm. Note that this policy (i.e., enforcement) specification is monotone: on input a output x, but if later another input a arrives, output another action y, with the semantics that y cancels out (the already output) x.

3.4.6.3 Input Reordering and Causality Assumptions

In Section 3.4.5.1 we discussed a fundamental constraint that we need to consider when decentralizing a global monitor: input reordering. The idea behind the constraint is that if we receive input on n different nodes then we cannot (always) know how to order the (local) sequences that happen on each node (even though we can trivially order events arriving at a single node).

The goal of this section is to transform a monitor from the I/O automaton model to the shared memory system model (as the one depicted in Fig. 3.17). More concretely, if we ignore communication channels (since we assumed that we cannot monitor them), our goal is to transform a distributed monitor (as depicted in Fig. 3.21a) to a distributed shared memory monitor (as depicted in Fig. 3.21b). Note that: (1) the monitor in Fig. 3.21a is the same monitor as the monitor depicted in Fig. 3.11b but without the communication channels, and (2) in Fig. 3.21b we have moved N_1 and N_2 to the left so that the monitored system resembles the generic shared memory system depicted in Fig. 3.17.

Note that *DM* (Fig. 3.21a) must be input reordering because even though *DM* can totally order the events it receives, the components in the distributed shared memory monitor (Fig. 3.21b) might not be able to. However, in our case we can relax the constraints of input reordering because of our causality assumption (see Section 3.3.1). More concretely, we can use causality to order events exchanged between some part of the environment and a node. For instance, in Fig. 3.21b due to the causality assumption we can order events exchanged between (a) the top environment and N_1, and (b) the bottom environment and N_2. Thus, instead of instantiating the definition of input reordering using the signatures of each component that the shared memory monitor interacts with, i.e., top environment, bottom environment, N_1, and N_2, we can instantiate it with only two signatures: (a) the signature of the composition of the top environment and N_1, and (b) the signature of the composition of the bottom environment and N_2. This means that



(a) Monitor *DM*

(b) Monitor *DM* transformed to distributed shared memory monitor *SM*

Figure 3.21: Transformation of monitor *DM* to distributed shared memory monitor *SM*

input reordering does not apply on each of the components interacting with the distributed shared memory monitor but rather it applies among the pairs of nodes and environments. For instance, in example #5 even if the monitor receives the DNS request after the DNS response it can order the events because of the causality assumption. As we will discuss in Section 3.5, without the causality assumption a monitor might not be able to enforce the described policy, e.g., because it might not be able to order the events between nodes and the environment accurately. Of course, in practice, there are often implicit causality assumptions which can help to order events; for instance an HTTP response is always preceded by a request. However, these causality assumptions are essentially protocol specifications which, as we discussed in Section 3.3.2, might not always be available or correctly implemented. So care needs to be taken about the causality assumptions made and the way they are used in the decentralizing of the monitors.

Another point that needs to be emphasized is the choice of the component that we choose to decentralize. In our framework, there are two choices: (a) decentralize the monitor, e.g., DM in Fig. 3.21a; and (b) decentralize the monitored target, e.g., the circled component in Fig. 3.11b. In

the first case, as we discussed preciously, we can assume a weaker version of input reordering. In the second case, however, the input reordering of the environment components suffices. This is because when we consider the monitored target as a unit (i.e., as a composed automaton) we know by the definition of composition (Section 2.2) that there are no inputs between the communicating components; i.e., the inputs of the monitor from the target are now outputs and thus under our control. As we will discuss in the next section, in order for our decentralized algorithms to work we need to control the order that local actions happen which is achieved by implementing a step-wise simulation of each local component. But, since we assumed that we are decentralizing the monitored target which includes the nodes of the underlying system we need to directly control the actions of the nodes of the systems. This will be possible only if we adopt a solution that attaches the monitor to the nodes using, for instance, inlining or re-writing [29, 103, 104]. Since in this thesis we are not considering these types of monitoring we will continue by assuming that we are decomposing the monitor, and not the monitored target, and we will adopt the weak input reordering assumption for the rest of the chapter.

3.4.6.4 Algorithms for Transforming Distributed Monitors to Distributed Shared Memory Monitors

In this section we present two algorithms that transform input reordering and monotone distributed monitors from the I/O automaton model to the shared memory model, i.e., to an I/O automaton where n processes access shared variables and interact with n users. The users of the shared memory system will be the environment and the (renamed) nodes of the distributed systems that will be trying to execute security relevant actions; the processes of the shared memory system will be the monitoring components that are intercepting these actions and (together with the shared variables) simulate the behavior of the distributed monitor. The first algorithm encodes the transition relation of the distributed monitor to the shared variable, essentially transforming the distributed monitor to a centralized monitoring architecture. The second algorithm encodes to the shared variable the global state of the monitor while each of the process simulates some part of the transition relation of the distributed monitor. The output of this second algorithm corresponds to a decentralized monitoring architecture.

Monitor as shared variable. In the first algorithm, which we will denote by $SM_C(A)$, the shared variable *SM* is essentially a state machine that performs the same computation as A. Because A is an I/O automaton, it has input and local actions. In order to express A as a shared variable type, i.e., a state machine, each transition of A must be encoded as a pair of invocations and responses. Thus, *SM* will be simulating A in a step-wise manner. An input action i, is encoded as an invocation $apply(i)$, with corresponding responses *ack* and $\perp$. The response *ack* denotes that the input has been processed by the shared variable, whereas the response $\perp$ denotes that the monitor is unable to accept the input and the process should try again (we will explain next why this is needed). Every local action o is encoded as an invocation $output(?)$, with corresponding responses $resp(o)$, which denotes the output action that monitor wants to output, and $\perp$

which means that the process should query the monitor again next. Finally, *SM* has as part of its state an extra variable *lock*, which is being used to indicate that the monitor has sent to a process an action to be output and its waiting for the acknowledgement from that process. Whenever *SM* receives an *output*$(?)$ and responds with a *resp*(o) it sets the *lock*. For any invocation *apply*(i) or *output*$(?)$ it receives from a process while the *lock* is set it responds with $\perp$. When it receives an invocation *output*$(!)$, meaning that the output was sent by the process, then the *lock* is released and the step-simulation of A is continued.

Each process P_i (of the shared memory model) acts as an interface between the environment and the shared variable. For each component of the environment U_i, there is a P_i that has the corresponding actions of A, i.e., $\bigcup_i$ *ExtSig*(P_i) $=$ *ExtSig*(A). In addition, each P_i has internal actions to communicate with the shared variable and input and output buffers as part of its state. Each P_i receives input from the environment and places it in its input buffer. Note that the users that interact with each P_i can be either an attacker or some node of the distributed system. In addition, note that there might be cases where certain nodes of the distributed system do not produce outputs, unless they receive some input first. For example, the webserver in example #5 never issues a request; it just waits for information that the DNS server and other monitoring components might send and it responds to these inputs. However, in order for our transformation algorithms to work correctly we need the users of the shared memory system (including the distributed system's nodes) to initiate communication. Thus, we modify each node of the network that must receive input before producing output to issue a dummy request indicating that it is ready to receive input. Then, after a user sends its inputs to the shared memory system, it is P_i's turn to execute local actions (while the user waits for results). This is because of our assumption that the attacker, i.e., the user, and the monitored system are causal (Section 3.3).

Each P_i tries to apply the input action in its input buffer to the shared variable; if the shared variable responds with $\perp$ then the P_i goes into a loop and keeps trying. Otherwise, it goes into a

state from which, in a busy-wait fashion, it asks the shared variable for an output action. When the shared variable responds with an output action P_i places it in the output buffer. In the meantime the shared variable is locked, i.e., not accepting any more requests from other P_is, and waits for the acknowledgement that the output has been produced. This acts a synchronization mechanism so that the outputs of the system can be ordered (see Section 3.4.5.1). Then, P_i removes the output action from the buffer and sends it to the environment. Finally, P_i sends an *output*(!) invocation to the shared variable so that the variable can continue with its simulation. This algorithm contains a lot of busy-waiting in order to synchronize but this is because of the nature of shared memory systems where invocation and responses happen in a single step. As we will discuss in Section 3.4.7.3, when the shared memory system is transformed to decentralized monitors over a network the monitors we can use a publish-subscribe approach to avoid this busy waiting and reduce communication.

Next we describe in a more detail the construction and structure of each process P_i:

1. **States:** P_i maintains two buffers: one for the received inputs, *inp_buffer*, and one for the pending outputs, *out_buffer*. In addition, it has a variable *state* which can take the values i, q, or o. The value of the variable indicates if P_i is waiting for input from the environment, querying the shared variable for output, or ready to output the shared variable's suggestion. Initially, *state* is set to i and the buffers are empty.
2. **Actions**:
    - For each input action i of A that is an output of U_i, P_i has an input action i that as an effect places i in *inp_buffer*.
    - For each input action i, P_i has a local action *apply*(i). The precondition of *apply*(i) is that i is in the head of *inv_buffer*. If the response from the shared variable is $\perp$ then P_i stays in the same state. Otherwise, the effect is that input action i is applied on

shared variable *SM* and then removed from *inv_buffer*. Finally, *state* is set to q.

- A local action *output*(?), which asks *SM* what output action to perform next. The precondition of *output*(?) is that the *state* flag is q, q. If the response from the shared variable is $\perp$, then P_i stays in the same state. Otherwise, the effect is appending to the *resp_buffer* the response of *SM* which is the output action o that should be performed next. Also, *state* becomes o.
- For each output action o of A that is an input action of U_i, *SM*(A) has a corresponding action o. The precondition of each o is that *state* is o and that *resp_buffer* is non-empty. The effect of the action removes o from *resp_buffer*.
- A local action *output*(!). The precondition of the action is that *state* is o and that *resp_buffer* is empty. The effect of the action changes *state* to i.

In order for our substitution to work, we need to show that the composition of all users with the distributed monitor A, i.e., $\bigcup_i U_i \times A$, has the same behavior as the composition of the users with the transformed automaton, i.e., $\bigcup_i U_i \times SM_C(A)$. This can be proven by simulation relations.

The simulation relation from $\bigcup_i U_i \times A$ to $\bigcup_i U_i \times SM_C(A)$ in order to prove that *traces*($\bigcup_i U_i \times A$) $\subseteq$ *traces*($\bigcup_i U_i \times SM_C(A)$) is easy to see: each state of $\bigcup_i U_i \times A$ is related to the states of $\bigcup_i U_i \times SM_C(A)$ that have exactly the same components.

In the other direction, without loss of generality, let us assume that we have two users, U_1 and U_2. Then the states of $\bigcup_i U_i \times A$ are of the form $q = \langle u_1, m, u_2 \rangle$, where u_1 is a state of U_1, m is a state of A, and u_2 is a state of U_2. Similarly, the states of $\bigcup_i U_i \times SM_C(A)$ are of the form $q = \langle u_1, (p_1, v), (p_2, v), u_2 \rangle$ where p_1 is the state of P_1, p_2 is the state of P_2, and v is the value of the shared variable. The idea is that v, the value of the shared variable, will be related with m, i.e., the state that A would be in. In particular, the simulation relation between m-values and v-values must respect monotonicity and input reordering. For, instance, if the shared memory

system moves from state $q = \langle u_1, (p_1, v), (p_2, v), u_2 \rangle$ to state $q' = \langle u_1, (p_1, v'), (p_2, v'), u_2 \rangle$, then by definition, either $v' = v$, or v' will be related to the same set of states that v was related too such that input reordering and monotonicity are preserved. First, it is clear that both automata start in equivalent states. Next, it is easy to see that for every step that $\bigcup_i U_i \times A$ takes, there is an equivalent step in $\bigcup_i U_i \times SM_C(A)$. The only interesting case is when $\bigcup_i U_i \times SM_C(A)$ performs internal actions that access the shared variable *and* change the value of v to some value v'. By construction, that internal action can either be an *apply*(i) or *output*$(?)$. In the former case, the input reordering property is applied to derived the trace equivalence, whereas in the latter case the monotonicity property is applied. The fact that the order of outputs is preserved is guaranteed by construction, by the locking mechanism. This is because, whenever one process is assigned to output something, all other processes requesting for outputs are blocked, until the shared variable receives an acknowledgement from that process. This completes the argument that the two composed automata have equal trace behaviors.

Monitor as process. In this second algorithm, which we will denote by $SM_D(A)$, the *logic* of the enforcement is distributed among the processes P_i's. As before, the signatures of each P_i correspond to the signature of each U_i that accesses the monitor. The shared variable *SM* is a Read-Modify-Write variable and maintains the state that the system is in, together with a state *lock*. The shared variable will be the synchronization primitive between processes. We, again, make the assumption that the nodes of the system that wait for some input before producing output are modified to issue a dummy request indicating that they are ready to receive input.

The idea of this algorithm is very similar to the previous algorithm. The main difference is that instead of having the shared variable simulate the monitor A the simulation is done locally by each P_i in a decentralized manner: each P_i simulates the actions that correspond to its signature. Specifically, each P_i maintains as state an input buffer *inp_buffer*, an output buffer *out_buffer*,

and a variable *state* which can take the values i, q, or o, initially set to i. After an input in received by a P_i it is enqueued at the *inp_buffer* and the *state* becomes q. Whenever the automaton gets a chance to perform a local action it tries to get access to the shared variable *SM*. If it cannot it keeps trying; otherwise it reads the state *lock*; if it is set if releases the variable *SM* and tries again. If *lock* is not set, then it (1) reads the state q that *SM* is in, (2) applies (locally) the transition that corresponds to the head of the *inp_buffer*, e.g., (q, i, q'), (3) moves to the new state, e.g., q', (4) removes i from the head of the input buffer *inp_buffer*, (5) writes to *SM* the new state, e.g., q', and (6) changes its variable *state* to o. Note that because *SM* is a Read-Modify-Write variable, this is a valid step: steps (2) - (4) are the modify part of the access. If, on the other hand, p is in a state that it can perform an output action, it performs the following steps: it first tries to get access to the shared variable *SM*. If it cannot, it keeps trying; otherwise it reads the state *lock*; if it is set if releases the *SM* and tries again. If *lock* is not set then it performs the following steps: (1) it reads the value of *SM*, i.e., the current global state q, (2) it goes locally to the read state q and checks if there is an enabled output action. If there is none, it releases the lock – this means that it is not its turn to output. If an output action is enabled then it places o to the output buffer, sets the *lock* and releases *SM*; (3) it outputs the action o and tries to get access to *SM* again; when it does, it (4) writes the end state q' to the *SM*, releases the lock, changes its local state to i, and releases *SM*.

To show that the composition of all users with the distributed monitor A, i.e., $\bigcup_i U_i \times A$, has the same behavior as the composition of the users with the transformed automaton, i.e., $\bigcup_i U_i \times SM_D(A)$, we can use a similar argument and simulation relations to the ones used in proving the behavior equivalence for $\bigcup_i U_i \times SM_C(A)$.

Discussion. One drawback of the second algorithm (i.e., the one that pushes the logic of the monitor to the processes) is that it needs to use more shared variables due to the need for more

concurrency control and thus more messages need to be exchanged. On the other hand, one advantage of the second algorithm is that the shared variables can consume less space and thus this approach might be preferable when space is an issue or when the state can be dynamically moved around the system so that communication is minimized. For example, if part of the network becomes inactive then the shared variable can be moved closer to the active part of the network thus improving performance and reducing communication costs.

3.4.7 Transformation of Distributed Shared Memory Monitors to Distributed Message-Passing Monitors

In this section we discuss how a monitor expressed in the shared memory model can be transformed to a monitor in the asynchronous network model. The presentation of the material is brief and, in a way, incomplete; the goal of this section is to give a high-level overview of the main ideas in the transformation from a shared memory model to an asynchronous network model, while discussing any details that are necessary in order to have complete correctness proofs, e.g., how the assumptions of Thm. 3.4.6 are satisfied in our framework (i.e., definitions and assumptions of Section 3.3 and Section 3.4). The familiar reader can skip this section; the interested reader can refer to Lynch [1] for a more detailed presentation.

In Section 3.4.7.1 we discuss *atomic objects* and an algorithm that substitutes shared variables in shared memory systems with atomic objects. The definition of atomic objects that we discuss in this section and the related substitution algorithm were originally introduced by Lynch [1]. We include them here because they are important to understand how shared memory systems can be simulated over asynchronous networks; in fact, many proofs of algorithms that simulate shared memory systems or shared variables over asynchronous send/receive networks (as the ones presented in Section 3.4.7.3) are reduced to proofs that use atomic objects. Then, in Section 3.4.7.2 we discuss how the assumptions made by the substitution algorithm of Section 3.4.7.1 are met in our framework (i.e., definitions and assumptions of Section 3.3 and Section 3.4). This is important in order to have complete correctness proofs of the decomposition process from a global monitor to distributed message-passing monitors (Section 3.4.4). Finally, in Section 3.4.7.3 we discuss a basic algorithm (originally introduced by Lynch [1]) that simulates a shared memory system over an asynchronous network by placing the shared variable on a single node. We also discuss a few variations that place the shared variable on multiple nodes thus achieving better

fault-tolerance (these variations were also introduced by Lynch [1]). Although these algorithms (the basic one and its variations) are not part of the contributions of this thesis, they are included here because, as discussed in Section 3.4.4, they are necessary to transform a distributed shared memory monitor, as the one in Fig, 3.13a, to a decentralized monitor in the message passing model, as shown in Fig. 3.13b.

3.4.7.1 Atomic Objects

In this section we discuss atomic objects, as they were originally defined by Lynch [1]. The material in this section is not new, and the familiar reader can skip this section; we include a discussion of atomic objects here because they are important to understand how shared memory systems can be simulated over asynchronous networks.

An *atomic object* (or *linearizable object*) is similar to a shared variable [1]. The main difference is that a process accesses a shared variable through an invocation and response atomically; however, accesses to an atomic object by a process (i.e., pairs of invocations and responses) can be interleaved with accesses from other processes. The main property of atomic objects is that, even though accesses from different processes can be interleaved, the processes are not aware of the interleaving: they think that they are accessing shared variables (atomically). This is very useful in asynchronous distributed systems, because by using atomic objects, and related algorithms, users of the distributed system obtain a view of the system that looks like a single, centralized object. As we have discussed before, in this thesis we use the results and algorithms of atomic objects for decentralizing global security policies. We also look at the assumptions under which these algorithms are correct, and how they affect the enforceability of centralized security polices—i.e., when atomic objects are not implementable by distributed algorithms.

Atomic object definition. Given a variable type T (Section 3.4.6.1), an *atomic object* A of type T is an I/O automaton that meets the following requirements [1]:

1. **External Interface:** A is accessed by n user automata U_i through n *ports* (one port for each user automaton). For each port i, A has some input actions a_i (corresponding to invocations a of T), and some output actions b_i (corresponding to responses b of T).
2. **Well-formedness:** For every execution of $A \times U$, where $U = \Pi U_i$, the sequences of invocations and responses between every U_i and A are alternating, and start with an invocation.
3. **Atomicity:** If α is a well-formed execution of $A \times U$, then for every pair of matching invocation and responses in α we can place a *serialization point* $*_\pi$ somewhere between them, such that if we move the pairs adjacently to their corresponding serialization points, the resulting sequence still consists of valid invocations and responses of the variable type T[17].
4. **Failure-free termination:** Every invocation in $A \times U$ has a response.

Relationship between atomic objects and shared variables. An important relationship between atomic objects and shared variables is that there exist algorithms that substitute shared variables with atomic objects, without the users noticing any difference in the behavior of the underlying system [1]. Next we describe such a substitution, that was first introduced by Lynch [1].

Assume that we are given an automaton A defined in the shared memory model that interacts with n user automata U_i. Also assume that there are functions *turn*$_i$ (one for each port i) that take as input a finite execution α of $A \times U$ and output either *system* or *user*, indicating whether the action that should extend α should originate from A or U. Finally, assume that we are given

[17] Note that our definition of atomicity is simpler than the one introduced by Lynch [1], as it mentions only matching invocations and responses (whereas Lynch's original definition mentions incomplete operations as well). This is because in this thesis we do not consider failures or faults and thus there is no reason to consider cases of incomplete operations.

atomic objects B_x, one for each shared variable x of A: each B_x has the same type with the shared variable x that it corresponds to. Now the construction is described. First, automata P_i are defined, one automaton for each process i. Each P_i has as inputs the inputs of A on port i and the responses of each B_x on port i, and as outputs the outputs of A on port i and the invocations for each B_x on port i. P_i's transition relation simulates the transitions of each process i with the difference that for each access to a shared variable x, P_i: (a) stops the simulation of process i, (b) issues an invocation to B_x, (c) waits to receive a response from B_x, and (d) continues the simulation of the process after it has received the response. Finally, the automaton *Trans*(A) is constructed by composing together the automata P_i and the automata B_x.

The following theorem (introduced by Lynch [1]) formalizes the fact that the users cannot distinguish between a shared memory system that uses shared variables and the transformed system (as described above) that uses atomic objects.

Theorem 3.4.6. [1]. *Suppose that α is any execution of the system Trans$(A) \times U$. Then there is an execution α' of $A \times U$ such that α and α' are indistinguishable to U, i.e., $\alpha|U = \alpha'|U$.*

Implementing Read-Modify-Write atomic objects in terms of Read/write variables [1]. The second algorithm in Section 3.4.6.4 used a Read-Modify-Write shared variable, as opposed to a regular Read/Write variable. Thus, a result that will be important in the transformation algorithms from shared memory models to asynchronous networks in Section 3.4.7.3, is the implementation of a Read-Modify-Write atomic object in the shared memory model with Read/Write shared variables (or objects). An algorithm that achieves such an implementation is introduced by Lynch [1] and relies on using several read/write shared variables and lockout-free mutual exclusion algorithms (e.g., *PetersonNP* [1]). The main idea is to use the mutual exclusion algorithm in order to allow processes to access a variable atomically through many distinct steps (using a lock), thus simulating a single read-modify-write access.

3.4.7.2 Substitution of Shared Variables by Atomic Objects in Distributed Shared Memory Monitors

Theorem 3.4.6 is important in proving that algorithms from the shared memory model can be transformed to algorithms in the asynchronous network model. Essentially, this theorem connects the two last steps of our blueprint (Section 3.4.4). In order to apply this theorem in our context we have to verify that our transformation algorithms in Section 3.4.6.4 meet the technical assumption of the existence of a $turn_i$ function. It is easy to see that the assumption is met by the assumptions made in the algorithms, and specifically the causality assumption of Section 3.3. There is only one instance where the $turn_i$ function does not operate as expected: after a process of the shared memory system has produced an output, it needs to release the lock that it has on the shared variable. Thus, at this point the system is enabled (releasing the lock), and so is the environment (since the user received its response and can proceed with the next request). However, this does not affect the correctness of the theorem (with respect to traces): keeping the proof of the theorem as is, we can rearrange the events of the execution α such that the release of the lock happens before the sending of the input from the user, which would be the behavior of a valid $turn_i$ function. The reason that this re-arrangement is valid, and does not change the behavior of the system, is that the received input is buffered locally; thus it does not have any effect globally; when the process is scheduled again, it first releases the lock and then tries to apply the input to the shared variable. Thus the behavior is the same as if the user was waiting for the release of the lock before sending input.

3.4.7.3 Transformation from the Shared Memory Model to the Network Model

In this section we discuss how to transform an automaton A expressed in the shared memory model to a set of automata that can be placed on the nodes of a given distributed system in a way that they have the same external behavior with A. The transformations discussed in this section were introduced by Lynch [1] and are not part of the contributions of this thesis. We include them here because they are necessary to transform a distributed shared memory monitor, as the one in Fig, 3.13a, to a decentralized monitor in the message passing model, as shown in Fig. 3.13b.

Let us assume that A interacts with n users through n ports. Remember that the algorithms presented in Section 3.4.6.4 that transformed global monitors to the shared memory model, by construction, made sure that the n users are essentially the nodes that the monitors need to be attached to, and the processes of A are the monitoring components. In addition, it is assumed that there exists a function $turn_i$, as was explained in Section 3.4.7.1. As we discussed in Section 3.4.7.2, our causality assumptions from Section 3.3.1 are in accordance with this assumption.

The goal is to construct an asynchronous message-passing system B, with n nodes n_i, that behaves similarly to A. More specifically, for any trace t of $B \times U$, where $U = \Pi U_i$, there should be a trace t' of $A \times U$, such that $t = t'$[18]. The final monitored system will be the composition of each user U_i, i.e., the node of the underlying distributed system, with node n_i, i.e., the corresponding monitoring component[19].

***SimpleShVarSim* algorithm [1].** The idea of the algorithm is to place the shared variables of A at some node n_i of B. Each node n_i will simulate the corresponding process P_i of A, but each time P_i tries to access a shared variable x, n_i: (a) sends an appropriate message to the node

[18]Note that our presentation is simpler from [1] because we do not deal with failures in this thesis, and because we are using traces. However, the results still hold.

[19]In addition, as discussed in Section 2.3.1, some renaming and hiding might be necessary to achieve the final monitored system.

that owns the variable (including a local message if x is hosted on n_i itself), (b) blocks until it receives a response, and (c) continues the simulation after the response is received. When a node n_i receives a message (including a local one) to access a variable x it hosts, it applies the request to x and sends the response with a message to the originator of the request[20]. The correctness of this transformation relies on atomic objects and Thm. 3.4.6 (which is the reason that we included Section 3.4.7.1 in this Chapter).

In the *SimpleShVarSim* algorithm each variable is placed on a single node without any requirements on the location of the variables. Depending on the underlying distributed system and policy that needs to be enforced, the location of the variables might affect the efficiency of enforcement [1]. Thus, in practice, the location of the variables might have to be chosen carefully. A framework that can help with such decisions is presented in Chapter 4.

In Section 3.4.6.4 we mentioned that our algorithms include a lot of *busy-waiting* because the monitoring components had to repeatedly access the shared variable in order to (1) guarantee synchronization and (2) simulate the behavior of a centralized I/O automaton. The *SimpleShVarSim* algorithm can be modified in order to improve communication complexity, by implementing a publish-subscribe paradigm, where the nodes that try to access the variable subscribe for notifications from the node that hosts the variable, and the node that hosts the variable publishes any changes to the value of the variable [1].

The advantage of *SimpleShVarSim* algorithm is its simplicity. However, sometimes it might be necessary to allow more than one nodes to host a single shared variable, e.g., for fault-tolerance reasons. In this case, an algorithm that will ensure atomicity of concurrent accesses is needed (similarly to Section 3.4.7.2), in order to ensure that all nodes maintain consistent copies of the variables [1]. For example, the *replicated state machine algorithm* places a copy of the shared

[20]A more detailed description of the algorithm, together with pseudocode for each process, and the proof of correctness can be found in [1].

variable on every node, and ensures that all copies are in consistent state through the use of logical time [1, 99, 124, 125]. By using logical time, nodes can agree on the order of events and thus all requests made to a shared variable are applied in the same order by each node. This way, all nodes maintain consistent copies of the shared variable and the distributed system gives the illusion to the users that there is a single copy of the shared variable.

3.5 Synchronous Enforceability

In Section 3.4 we identified certain characteristics of the security policies that are enforceable in asynchronous distributed systems, i.e., input reordering (see Section 3.4.5.1) and monotonicity (see Section 3.4.6.2), by characterizing which global monitors (i.e., centrally specified policies) can be decomposed over asynchronous distributed systems. This characterization was *constructive* in the sense that we presented a blueprint for constructing algorithms that decompose global monitors over distributed systems. Specifically, we gave two example algorithms: the first one corresponded to *centralized enforcement*, where all enforcement decisions are made at a single monitor in the monitored distributed system, and the second one corresponded to *decentralized enforcement*, where decisions are made locally by each monitor. As discussed in Section 3.4.6.4 and Section 3.4.7.3 in order for the distributed monitors to correctly simulate the global monitor the decomposition algorithms must provide the distributed monitors with the ability to control concurrency. It is well known that concurrency control algorithms, such as locking or logical clocks, can be very expensive in terms of messages exchanged and communication [1, 126]. The same is true for fault-tolerant algorithms which make use of redundancy in order to recover from faults, e.g., by maintaining multiple copies of shared variables (Section 3.4.7.3). Remember that fault tolerance was one of the key motivations for decentralized and hierarchical enforcement (as we discussed in Section 3.1 and Section 1).

Previous work in decentralized security policy enforcement has introduced decomposition algorithms that are simpler than the ones presented in this thesis [91, 92, 93, 94, 95]. Thus, one could think that our goal to characterize the security policies enforceable in distributed systems led us to introduce algorithms that are (unnecessarily) complicated. However, this is not the case. The reason that previous work has introduced simpler algorithms than ours is because they assumed that the underlying distributed system is synchronized (e.g., they relied on the Network

Time Protocol [127] (NTP) to synchronize the nodes of the system). Given that most algorithms in decentralizing security policies are assuming that the underlying network is synchronous, in this section we discuss how our results of Section 3.4 are affected by assuming a synchronous underlying communication network. First, in Section 3.5.1 we give a brief description of the formal model of synchronous networks since it is quite different from the model of asynchronous networks (the interested reader may refer to Lynch [1] for a more detailed presentation). Next, in Section 3.5.2 we discuss how the characteristics of synchronous networks affect our main results of Section 3.4.

3.5.1 Background (Synchronous Networks)

In this section we briefly present the model for synchronous network systems (which is different than the asynchronous network model) as it was originally introduced by Lynch [1]. A more detailed presentation can be found in [1].

Synchronous Network Systems. A *synchronous network system* is defined as a collection of n processes located at the n nodes of a directed network graph $G = (V, E)$, i.e., $n = |V|$ [1]. For each node i, *out-nbrs*$_i$ denotes the set of nodes in the digraph G that are connected to i through edges that have i as source, and *in-nbrs*$_i$ denotes the set of nodes that are connected to i through edges that have i as the destination. The length of the shortest directed path from i to j in G is denoted as *distance*(i,j), and the *diameter* of the network is denoted as *diam*. A fixed *message alphabet* M is assumed, with *null* indicating that no message is sent.

A *process* is associated with each node $i \in V$, which consists of:

- a (possibly infinite) set of *states states*$_i$,
- a nonempty subset of *states*$_i$, *start states start*$_i$,

- a *message-generation function* $msgs_i$, from $states_i \times out - nbrs_i$ to $M \cup \{null\}$, and
- a *state-transition* function $trans_i$, from $states_i$ and vectors of elements of $M \cup \{null\}$ to $states_i$.

Each edge (i, j) in G has a corresponding *channel* that can carry at most one message at a time.

The system executes by performing the following steps: (1) each process computes the messages to send to the outgoing neighbors by applying its message-generation function to its current state, and places these messages to the corresponding channels; (2) each process removes the messages from the channels, and moves to a new state by applying its state-transition function to its current state and the incoming messages.

These two steps define a *round*. There are no limits on the amount of computation taken by the message-generation and state-transition functions.

3.5.2 Decentralize Monitors in Synchronous Networks

As we discussed in Section 3.4, one of the aspects of distributed systems (Section 3.2) that decentralized algorithms have to deal with is the lack of global time and global ordering (i.e., partial order instead of total order). The effects of global time and ordering in enforceable policies was discussed in Section 3.4.5 (input reordering), and in Section 3.4.3 (closed action-deterministic automata).

Algorithms that decentralize global monitors over synchronized distributed systems can be simpler because in synchronous networks the issues of global time and ordering are simpler to deal with than asynchronous networks. For instance, some decentralizing algorithms make the assumption that the components of distributed systems communicate through a synchronous bus [128]. This means that a node can receive events from all other nodes at a single instant in

time, i.e., in a round, as it was formally defined in Section 3.5.1 (note that this is not exactly *instant broadcast* as we assumed in Section 3.4.3). Other solutions (e.g., [92, 95, 109]) rely on synchronization protocols, such as NTP, to achieve synchronization and simulate rounds of synchronous networks.

Input reordering is one of the constraints that was introduced due to the existence of partial ordering of events, i.e., concurrency. Input reordering is still a constraint on decomposable monitors (and enforceable policies) in synchronous networks but it only applies within rounds. Since every action can be tagged with the number of the round it occurred, the components can know the exact ordering of actions unless two actions happen in the same round (which means that they will have the same timestamp). Thus, the only policies that are not enforceable are the policies that require that different orderings of concurrent events lead to different outcomes. For example, let us consider a variation of the example attack #5 discussed in Section 3.3. Let us assume that a network IDS is observing traffic and is trying to detect attack #5. Let us also assume that the IDS observes the following four events: an HTTP request to the webserver, an HTTP response from the webserver, a request to the DNS server, and a response to the DNS server. Even though the IDS might be able to order the four events into two pairs, a HTTP request/response and a DNS request/response, it might not be able to order the two pairs. Thus, the IDS does not know if the DNS traffic preceded the HTTP traffic (i.e., an attack), or if the HTTP traffic preceded the DNS traffic (i.e., not an attack)[21]. To overcome this problem we suggested a solution (see Section 3.3) that used synchronization between the HTTP server and the DNS server in order to decide whether the DNS request happened before the HTTP request[22]. However, such a situation will never occur in a synchronous system since all events are ordered, i.e., the monitors can classify exactly when the requests were made and decide whether the HTTP request happened

[21] As we discussed in Section 3.3 the situation might be more difficult if we do not have protocol specifications, or causality assumptions, that allow us to order events between a node and the environment.

[22] Note that if the two events happen concurrently then the trace represents a valid behavior.

before or *after* the DNS response was sent back. Thus, in synchronous networks causality assumptions are not explicitly needed for ordering events since the ordering can be inferred from the timestamps that the events have. This means that there are policies that are not enforceable in asynchronous networks (e.g., due to lack of causality assumptions) but are enforceable in synchronous networks. This argument can be also seen by the relationship between logical clocks and real-time clocks: even though there are several variations of logical clocks to order events in asynchronous networks, they cannot exactly represent the real-time temporal behavior of the system [129].

Monotonicity (see Section 3.4.6.2), on the other hand, is still a valid constraint of decomposable global monitors in synchronous networks. Information, and thus global knowledge, takes time to propagate through the network and thus a node that is about to output an event is always at risk that some security relevant input, that might invalidate its output, has been (recently) received by some remote node.

Despite the simplicity that synchronous networks offer, we decided to focus on asynchronous system in this thesis for the following reasons:

1. Many real-world systems are not synchronous (they use other means of synchronization, such as logical and vector clocks) [130, 131, 132]. Our algorithms and results are applicable in such systems, e.g., identifying *bad* behaviors (either in real-time or by analyzing audit-logs), whereas algorithms for synchronous networks will not correctly simulate the intended global behavior since the assumptions of real-time ordering are not met.
2. Many decentralization algorithms rely on NPT to achieve synchronization (e.g., [92, 95, 109]). However, it was recently shown that NTP is susceptible to attacks which can lead to the nodes being desynchronized [133, 134]. If an attacker were to attack the NTP protocol then the decentralized algorithms would not give correct results [109]. Our approach and algorithms are applicable in cases where the NTP is attacked or when timestamps in

logs are not appropriately maintained. This point is also important for the evaluation of decentralized IDS. Typically IDSs are evaluated using datasets from previously organized evaluations [51, 135, 136]. Some of these datasets contain multi-step attacks and were used to evaluate IDSs that were correlating attack information [51]. But in addition to the existing criticism that these datasets have received, e.g., for not representing real world traffic [109], it must be emphasized that these datasets did not include attacks on the synchronization protocols (i.e., NTP). Thus, evaluations of decentralization algorithms and systems using the above datasets might not be a good indicator of their performance when used in the real-world and in fact might give a false sense of security.

3. Our results in Section 3.4 provide an upper bound of the policies enforceable in distributed systems, both synchronous and asynchronous. This is useful in cases where, for example, after an attack it is realized that the timestamps of the logged actions are not correct, e.g., because NTP was attacked, then our results can characterize the attacks that can be identified using the existing logs.

3.6 Hierarchical Enforceability

Enforcing security policies over distributed systems using decentralized monitors can achieve better fault-tolerance, communication efficiency and computational efficiency [91, 92, 93, 94, 95, 96]. In Section 3.5 we explained that it is simpler to decentralize a security policy over a synchronous distributed system, as compared to an asynchronous system. We also argued that designing algorithms for decentralizing security policies over asynchronous systems has several advantages, such as their application (1) to real-world systems that are by nature asynchronous (e.g., [130, 131, 132], and (2) in situations where determined attackers disrupt the protocols that are being used by distributed systems to achieve synchrony (e.g., attacks against the network time protocol [133, 134]).

Hierarchical enforcement is another approach that has been suggested to achieve better fault-tolerance, communication efficiency and computational efficiency in enforcing security policies over distributed systems [87, 88, 89, 92, 137]. In hierarchical enforcement security mechanisms are organized in hierarchical fashion: at the lowest level of the hierarchy mechanisms are responsible to collect data from a subset of the nodes on the network; the security mechanisms perform some analysis on the collected data and forwards the results to mechanisms at higher levels, which further analyze the data. Since data are analyzed at each level of the hierarchy, mechanisms at higher levels have less analysis to perform and thus reduced computational load. In addition, the analysis performed at each level can reduce the data that need to be sent further up the hierarchy thus reducing the overall communication load as well.

The motivation behind hierarchical approaches was the detection of attacks similar to the example attacks $\#1-\#3$ in Section 3.3 [87]. Hierarchical enforcement designs are very promising in enforcing security policies over distributed systems. Thus, designing algorithms that transform global monitors to equivalent distributed monitors that operate in a hierarchical fashion is highly

desirable.

In asynchronous systems there is a limitation that can (significantly) reduce the benefits of hierarchical enforcement approaches: nodes in asynchronous systems typically use logical time in order to synchronize [124]. Since such synchronization is needed to order events on different nodes (e.g., simulating a shared memory system over an asynchronous network [1]), all nodes, regardless of what level in the hierarchy they belong to, need to exchange the same amount of information. It may seem that solutions such as hierarchical clocks [138] could be used to exploit the hierarchical structure of the mechanisms in order to minimize the amount of information that needs to be exchanged among nodes for synchronization. However, it has been shown that any attempt to minimize the amount of information used for representing logical time (e.g., hierarchical clocks) results in a loss of accuracy in the causal relationship that can be expressed by the corresponding solutions [139]. In fact, in order for nodes to achieve an accurate view of the causal relationship of events in the system, *vector clocks* need to be used [140, 141]: logical clocks may not provide an ordering of events that is isomorphic to the real causal relationship [129]. Since vector clocks require a larger representation than logical clocks, more information needs to be exchanged among nodes. Thus, any solution that does not use vector clocks, e.g., uses hierarchical clocks instead, will have the drawback that it will order events in a way that may not be consistent with the real causal relationship. Thus, policies that depend on the real causal relationship of events will not be enforceable in a hierarchical manner unless the same amount of information is exchanged among nodes (which defeats the premise of decreased computational and communication load).

The problem of transforming global monitors to (efficient) hierarchical distributed monitors has a more fundamental limitation than representation of logical time: such a transformation is, in general, infeasible. The reason is that there are algorithms, i.e., centrally-specified security policies, that require that nodes exchange all the information they have in order to enforce the

global policy. Such algorithms include algorithms that require the evaluation of a predicate over the global state of the system, e.g., identifying if the system has reached to a deadlock [1]. In these cases, a node cannot reduce the information it forwards to other nodes by doing a local computation on the information it receives from other nodes.

Despite the above negative results, hierarchical approaches can still be used to efficiently enforce (some) global distributed policies. For instance, it is known that there exist non-trivial global algorithms that can be computed over a distributed system in constant time, i.e., independently of the size of the network [98, 142, 143], and efficient (and fault-tolerant) data structures for distributed systems that could be used in hierarchical settings, such as *conflict-free replicated data types* [144]; such data structures could be used for example to store the state of a global monitor, similarly to the way it was done in Section 3.4.4.

Next, we present a result that characterizes a set of centrally-specified security policies that can be efficiently enforced hierarchically over a distributed system. Let us consider the hierarchical structure of the monitored system as a spanning tree. Spanning trees of networks can be used to broadcast information over the nodes of a distributed system in $O(diam)$ rounds, by using, for example, a flooding algorithm that propagates the information from the root of the tree to its leafs [1]. The reverse operation, i.e., propagating information from the leafs up the tree towards the root, is called *convergecast* [98, 99]. Convergecast can be used, for example, to send an acknowledgement from the leafs of the spanning tree to the root that they received the information that the root broadcasted. During this process, a node at a given level can wait to receive all acknowledgements from its children and then send a single acknowledgement to its parent node. Thus, if we assume a network of seven nodes where the root node R has two children and each child has also two children, by sending acknowledgments using a convergecast R will receive only two acknowledgements, instead of six. The example of sending acknowledgements to a broadcast operation using convergecast is an instance of a *global semigroup function* [98]. A

global semigroup function f has the following properties: it is (1) associative, (2) commutative, and (3) the representation of $f(x)$ is small compared to the representation of the input x. Due to the associativity and commutative properties, these functions can be computed by each node of the tree independently from other nodes at the same level. In addition, because of property (3), the size of the messages that are propagated up the tree will remain small. Functions that are global semigroup functions include *addition*, *maximum* and *logical conditions* [98].

Examples #1, #2 and #3 that were used as motivating examples for hierarchical enforcement approaches (e.g., [87]) are instances of global semigroup functions. For instance, doorknob attacks (i.e., example #1) can be detected by keeping a counter of each IP address that contacts a host; hosts can then add their counters and once the counter for a particular IP address exceeds a threshold an alarm is raised. Attacks #2 and #3 can be detected in a similar manner. Other attacks, such as port scanning and IP scanning also fall under this category of attacks. Thus, global semigroup functions are a first characterization of policies that are efficiently enforceable in a hierarchical manner.

3.7 Distributed Security Automata

In Section 2.3 we showed how to encode truncation (i.e., security) automata [27, 145] and suppression automata [145] to corresponding monitors in our framework. A truncation monitor intercepts the security relevant actions that a target application wants to execute and takes one of the following two actions: if the attempted action will not violate the security policy then the monitor outputs the action verbatim; otherwise, the monitor *halts* the target application. This means that after the application tries to take a *bad* step it is not allowed to take any further steps even if future steps could have been valid. Suppression monitors operate in a similar way to truncation monitors but they *suppress* intercepted security relevant actions that will violate the security policy and allow the target application to continue executing. Thus, even though both truncation and suppression monitors enforce safety policies [145], a suppression monitor can be more permissive by allowing an application to exhibit more correct behavior than a truncation monitor. Truncation monitors can be *simpler* than suppression monitors because, intuitively, they have less bookkeeping to do. This can also been seen by the fact that truncation monitors have simpler operational semantics than suppression monitors [145].

It is important to note that in a distributed system where multiple truncation monitors are installed, if the monitors need to simulate the behavior of a global truncation monitor then they must be able to communicate. For example, consider a distributed system with three monitored components A, B, and C, with signatures $\mathit{Sig}(A) = \{a\}$, $\mathit{Sig}(B) = \{b\}$, and $\mathit{Sig}(C) = \{c\}$. Let us assume, that the sequence $\langle a, b, c\rangle$ is not allowed by the policy, whereas the sequences $\langle a\rangle$ and $\langle a, b\rangle$ are allowed. If the underlying system tries to execute sequence $\langle a, b, c, a\rangle$, then once node C intercepts the action c it needs to notify node A that the computation has been invalidated and must be stopped. Thus, nodes need to communicate and synchronize.

An interesting question is whether communication between automata is considered security

relevant and, in particular, whether it is considered part of their operational semantics. For instance, let us consider two truncation automata that communicate trough actions *send* and *receive*. If neither of these actions is security relevant (i.e., an action that the target would exhibit and the monitor would intercept), it is not clear whether the automata should be classified as truncation automata (since they either accept security relevant actions or halt the target) or as edit automata (since they can exhibit actions to the output stream that were not previously intercepted). However, if we compose the truncation automata and hide their communication then the result will be a truncation automaton: it will either accept an action or halt the system. For this reason in this section we will assume that communication between automata is not considered security relevant and not part of their enforcement-related operational semantics.

We mentioned that in order for the distributed truncation monitors to simulate a global truncation monitor they need to communicate. However, if the distributed truncation monitors do not communicate with each other then they may be able to simulate the behavior of a global (stronger) suppression monitor. This is because if one of the truncation monitors halts the execution of its local node, the rest of the nodes in the system can continue to operate. Thus, it seems that distributed truncation monitors may be able to simulate monitors with more complex behavior. This idea is formalized in the following theorem.

Theorem 3.7.1. *Given n truncation monitors $T_1, \ldots, T_n$, there exists suppression monitor S such that traces$(S) =$ traces$(\Pi_i T_i)$.*

Proof sketch. Let S be the composition of T_i's. Consider that each T_i intercepts actions from an application A_i. Thus, the suppression monitor S receives actions from the application $A = \Pi_i A_i$. This means that a truncation monitor T that interacts with application A, will terminate A, if a bad action is sent by a component A_i. On the other hand, S will terminate the specific A_i, but the rest of the components will continue operating. Thus S is indeed a suppression monitor. □

Thm. 3.7.1 describes suppression monitors that can enforce policies that contain traces with the following property: if a security relevant action a invalidates a trace t, then no valid extension of t contains a. Intuitively, this means that if we terminate the component that tried to exhibit action a after trace t then we can still obtain the rest of the valid behaviors of the system (i.e., the extensions of t). More concretely, let us assume that we have a system with two components: component A exhibiting action a, and component B exhibiting action b. A policy that allows only b's can be enforced even if component A is terminated because all valid traces, i.e., sequences of b's, can be exhibited by component B which is still operating.

However, there is a class of suppression monitors that cannot be described by Thm. 3.7.1. Consider a policy that considers two actions a and b. Let us assume that all sequences of a's are valid, but sequences of b's are allowed to appear only if an a action had been output. For this policy there is a suppression monitor S that upon receiving the sequence $\langle b, a, b\rangle$ can produce the sequence $\langle a, b\rangle$ by suppressing the first b and allowing the second one. But no combination of truncation monitors can achieve this result because a local truncation monitor would have to terminate the component that produces the b's. This idea can be formally stated as follows:

Theorem 3.7.2. *There exists a suppression monitor S that cannot be implemented using a finite number of truncation monitors.*

Proof sketch. The proof is by construction, formalizing the details of the previous example in a straightforward manner. □

Note that Thm. 3.7.2 places a cardinality constraint on the number of truncation monitors that can be used in the implementation. This is because one possible solution to the problem of allowing actions that were previously prohibited is to have multiple truncation monitors for each node. Thus, if a bad action is detected the truncation monitor trivially *halts* the target by doing nothing, and notifies another truncation monitor to take over (remember that communication is

allowed between truncation monitors). Because the I/O automata model requires that composed automata have disjoint output actions (see Section 2.2), this would require some modification in the theory, e.g., consider some set of actions as *equivalent*. But, even if we assume that we have equivalent output actions, another practical limitation is that there are cases where we would need an infinite number of truncation monitors (e.g., if an action a needs to be blocked and then allowed, infinitely often). This might be possible in practice in environments where truncation monitors can be generated dynamically, i.e., every time a truncation monitor terminates another one is created at run-time to take over[23]; but this cannot be done statically, i.e., we cannot create a mechanism with infinitely many components. Based on the previous theorems and discussion we have the following corollary:

Corollary 3.7.1. *Given a security policy P and a distributed signature S then a suppression monitor globally enforces P if and only if there exist (potentially infinite) communicating truncation monitors that globally enforce P.*

Note that in order to simulate the enforcement capabilities of a suppression monitor over a distributed system using truncation monitors then the constraints analyzed in Section 3.4 must be met (in addition to the ones discussed in this section). Similar results can be proven for simulating edit monitors using distributed suppression (i.e., truncation) and insertion monitors [145].

[23] Such scenarios can be modeled using dynamic I/O automata [146].

3.8 Related Work

In Section 2.7 we discussed some previous work on formal frameworks for run-time monitors. This work characterized the enforceable policies by run-time monitors expressible in the respectively introduced frameworks (e.g., security automata and safety policies [27]). However, as we mentioned, these frameworks modeled and characterized enforceable policies by *individual* monitors. To the best of our knowledge, we are the first to characterize the security policies that are enforceable by run-time monitors in distributed systems, including centralized, decentralized and hierarchical monitoring architectures. Even though no work, to the best of our knowledge, has characterized the enforceable security policies over distributed systems, there is a large body of work that discusses algorithms and architectures to enforce security policies over distributed systems. We discuss some of this related work next.

Distributed monitoring of policies specified in temporal logics. An area of research related to decentralized (distributed) monitoring that has received attention focuses on automatically synthesizing monitors from security policies that are specified in some temporal logic. Although there is work on evaluating LTL formulas using centralized monitors (e.g., distributed systems [147], and multi-threaded applications [148]), we will discuss related work that focuses on synthesis of decentralized or hierarchical architectures. Sen et al. introduced Past Time Distributed Temporal Logic (PT-DTL) to specify and monitor violations of safety properties in distributed systems [149]. PT-DTL is a temporal logic that is a based on Past Time Linear Temporal Logic (PT-LTL); PT-LTL is used for specifying and monitoring violations of safety properties of software systems [149]. In order for a monitor to evaluate a formula written in PT-LTL, the global state of the system has to be computed, which in distributed systems might require the exchange of a large number of messages. PT-DTL allows to specify safety properties of synchronous dis-

tributed message passing systems. In order to avoid the expensive exchange of messages among all nodes in the network (in order to compute the global state of the system) PT-DTL contains an epistemic operator @ that allows a monitor to take into account the *last known state* of just a given remote node (specified through the @ operator). Thus, in the evaluation of a formula only the nodes that are explicitly mentioned in the formula need to be contacted. The algorithm uses vector clocks to synchronize the processes. A more general logic, i.e., Linear Temporal Logic (LTL), was used by Bauer and Falcone to specify policies that can be automatically decentralized over a synchronous distributed system [128]. Their algorithm assumes that (1) messages can be totally ordered and (2) nodes can send messages to all other nodes within a single instance, i.e., if a message is send at time t, it is received by the recipient at time $t + 1$. Extending this work, Falcone et al. present an algorithm for monitoring policies expressed in a subset of LTL formulas assuming a global clock but lifting the assumption that messages are received within a single round after they are sent [150]. A similar approach, but for a hierarchical monitoring architectures (instead of decentralized) is introduced by Colombo and Falcone [151]. A decentralized algorithm for monitoring LTL specifications over asynchronous systems (i.e., without the assumption of a global clock) is introduced by Mostafa and Bonakdarpour [152]. Due to the inability to totally order events in the system their algorithm relies on vector clocks to evaluate the global predicates defined by the LTL formulas, similarly to previous work on evaluating global predicates over asynchronous distributed systems [1, 129]. Although the algorithms in the papers described above bear some similarities with the some of algorithms presented in this chapter (e.g., the decentralized algorithms for synchronous systems and PT-DTL are similar to our algorithm for closed action-deterministic automata from Section 3.4.3) the main differences are that (1) our algorithms work on a larger class of centrally specified policies (i.e., arbitrary I/O automata, not just safety-LTL specifications) and (2) the main focus of our work is not too provide efficient algorithms for decentralizing algorithms but to use algorithms as means to characterize enforceable

security policies.

Distributed intrusion detection systems and firewalls. Detecting attacks in distributed systems can be achieved by collecting information from all nodes of the distributed system (e.g., audit logs) at a central location and then analyzing the data at this central location. Several papers have described such approaches, e.g., [52, 83, 84, 85, 86]; however, as we did with related work on temporal logics, we will focus on related work that presents hierarchical or decentralized approaches to detecting attacks.

Staniford-Chen et al. present a hierarchical approach to detecting large-scale attacks such as sweeps, worms and coordinated attacks [87]. Their approach constructs at run-time a graph that represents the causal relationship of events in the network. The graphs are specified through appropriate rule sets which include *combining conditions* that allow to merge graphs to larger graphs. The monitors are organized in a hierarchical fashion: lower levels in the hierarhcy construct graphs which they forward to the higher levels in the hierarchy; however, the forwarded graphs are reduced versions that contain only essential information about the graph. Then, monitors at the higher level can collect (reduced) graphs from their children in order to detect attacks. Their algorithm assumes a synchronous, non-faulty network. A similar approach is suggested in NetStat, a network-based intrusion detection system [88]. Debar an Wespi propose a hierarchical architecture of *probes* and *aggregation and correlation components* that can collect alerts at lower levels and correlate and aggregate them at higher levels. The analysts can observe the more condensed view of the security traffic in the network at the higher levels and thus more efficiently detect attacks [89].

Cooperating security managers was one of the first systems to implement a decentralized approach in detecting attacks over a network [90]. However, the focus of the system was to detect only a specific type of attacks, namely users' login chains. *Cards* is a distributed system

for detecting in a decentralized manner a larger class of multi-step attacks [91, 92, 93]. Cards uses *signatures*, i.e., event patterns, to specify attacks that may occur across multiple systems. It also uses positive and negative events in order to classify attacks as true positives (positive events) or false positives (negative events). The papers focus on a specific type of signatures, called *serializable signatures*, that describe sequences of events that can be totally ordered (i.e., closed action-deterministic automata from Section 3.4.3) Another system for decentralizing attack patterns over a distributed system was introduced by Kruegel et al. [94, 95]. The authors recognize that the use of peer-to-peer monitoring systems to enforce policies that specify arbitrarily complex patterns could result in a potential message explosion among the monitoring agents. Thus, the authors focus on a less expressive policy specification language that allows policies that can be specified as tree patterns. In addition to distributed intrusion detection systems, decentralized architectures for enforcing security policies have also been suggested for distributed firewalls [96, 97]. The algorithms introduced in the above papers assume a synchronous network. Our algorithms and analyses are designed for asynchronous systems (for reasons described in Section 3.5). However, we discuss the relationship of enforcement in asynchronous systems to enforcement in synchronous systens (Section 3.5) and to hierarchical enforcement approaches (Section 3.6).

Common knowledge in distributed systems. Two high-level conclusions that one can derive from our results in Section 3.4 and Section 3.5 are the following: (1) not every centralized (i.e., global monitor) can be simulated over a distributed system, and (2) if we make timing assumptions about the underlying distributed system, then we can enforce more security policies.

These conclusions are not technical consequences of the framework that we have introduced (i.e., I/O automata, and our definitions of monitors, targets, and enforcement), but are rather expressions (within our framework) of fundamental limitations of distributed systems.

Halpern and Moses showed that common knowledge in distributed systems (i.e., facts that are "publicly known" by all members of the system) with unreliable channels or asynchronous communication cannot be obtained [153]. Since common knowledge is a necessary condition to perform many tasks that require coordination, the authors concluded that there are tasks that can be (trivially) performed in centralized systems but are impossible to perform in distributed systems. However, as the authors discuss, many real-world problems do not require strong notions of common knowledge in order to be solved, and this is why there are many practical implementations of, e.g., distributed agreement and coordination protocols. Our results are very similar to the ones of Halpern and Moses. The main differences are that (1) we used a different formalism to express and derive these fundamental results, and (2) we have characterized (through monotonicity and input-reordering) the *weaker* notion of common knowledge that suffices to perform centralized tasks in a distributed manner (i.e., distributedly enforce a centralized security policy).

Halpern and Moses also showed that knowledge that can be obtained in distributed systems with bounded-delay communication channels (called $\epsilon-$common knowledge, where ϵ represents an upper bound on the guaranteed delivery of messages) is weaker than the general notion of common knowledge: if common knowledge can be obtained, then $\epsilon-$common knowledge can be obtained as well; but the converse is not true [153]. If we consider ϵ to be the diameter of the distributed system, then ϵ also represents the upper bound of message delivery in synchronous networks. Thus, our second result, namely that timing assumptions help us enforce more security policies, is another perspective of Halpern and Moses' result: if a policy can be enforced in an asynchronous system, then it can be enforced in a synchronous systems as well; but the converse is not true (since the policy might not be input-reordering across multiple rounds).

3.9 Conclusions

Formal models for run-time monitors have helped us improve our understanding of the powers and limitations of enforcement mechanisms [27, 30] and have aided in the design and implementation of security policy specification languages [81, 82, 104, 105]. However, these models are not suitable for analyzing the enforcement capabilities of monitors in distributed systems.

In this chapter we extended the formal framework we introduced in Chapter 2 to allow modeling and reasoning about enforcing security policies in distributed systems. We first discussed some key differences between centralized and distributed systems which provide some intuition behind the limitations of enforcing security policies in a distributed manner (Section 3.2). We also presented some motivating examples of multi-step and distributed attacks that guided our assumption that attackers are *causal* (Section 3.3).

Next, we discussed how to characterize the security policies that are enforceable in asynchronous distributed systems (Section 3.4). The characterization was based on the *constructive* characterization of the centrally-specified monitors that can be decomposed over a given distributed system (Section 3.4.2). In this characterization we identified two key constraints: *input reordering* and *monotonicity* and we presented two novel algorithms that can be used to decompose centrally-specified monitors over distributed systems (Sections 3.4.4-3.4.7). Then, we discussed how our characterization of enforceable policies in asynchronous distributed systems is affected by assuming that there is a global clock in our system (Section 3.5.1). Synchronous systems model practical situations where, for instance, protocols such as NTP are used to allow a finer-grained ordering of events in the system. We also gave a first characterization of the policies that are (efficiently) enforceable in a hierarchical manner (Section 3.6). Finally, we illustrated how in distributed systems it may be possible to simulate the global behavior of powerful monitors using weaker' monitors that interact with each other (Section 3.7) and we discussed related

work (Section 3.8).

Chapter 4

Probabilistic-Cost Enforcement

In Chapter 3 we discussed how multiple monitors can be used to enforce policies over distributed systems. Three of the monitoring architectures that we focused on were: centralized monitors (all enforcement decisions are made by a single node on the network), decentralized monitors (enforcement decisions can be made by any node on the system), and hierarchical monitors (the monitors form a tree, and decisions are made by monitors at higher levels in the hierarchy).

These three architectures have different benefits and drawbacks. For instance, centralized monitors are single points of failure, but they can be very efficient in terms of communication: all necessary information to enforce a policy is maintained at a single location, and there is no need for monitors to synchronize and exchange state information (as for instance in decentralized architectures). As another example, as we discussed in Section 3.6, there are certain policies where hierarchical monitors can be very efficient in terms of computational and communication resources, but in other cases such architectures might not be as efficient as, for example, centralized monitors.

The efficiency of a distributed monitoring architecture does not only depend on the nature of the security policy that needs to be enforced. In practice, other factors that affect the design choice

of the architecture to be implemented include: the underlying architecture of the distributed system, probabilistic knowledge that the security engineer might have about the environment (e.g., what is the probability that an attack might happen), and probabilistic knowledge the security engineer might have about the system itself (e.g., what is the probability that a node might crash).

In Section 3.4.4 we presented a blueprint for designing different decomposition algorithms. However, this blueprint did not account for the efficiency of the different algorithms that could be derived. In this chapter we provide the technical basis that allows the comparison of different monitoring architectures. Specifically, we extend our basic framework of Chapter 2, using Probabilistic I/O Automata [154, 155]. We show how to use our new framework to assign probabilities and costs to different monitors, how to use these probabilities and costs to calculate the expected costs of monitors, and how to compare different monitoring designs. It is important to note that this framework is orthogonal to the framework of Chapter 3, i.e., it can be used to calculate the expected cost of different distributed monitoring architectures. Even though in this chapter we focus on introducing the (sound) technical machinery needed to reason about the expected cost of different monitoring designs, as future work, we plan to show how this framework can be used to compare different distributed monitoring designs, and design algorithms that optimally decompose centralized monitors.

4.1 Introduction

In Section 3.3.1 we described one type of intrusion detection systems: specification-based IDSs [110, 116]. These systems use a formal specification of the good behaviors of a monitored system (e.g., a TCP protocol state machine [110]) to classify the intercepted traffic as either good (if it obeys the specification) or bad (if it does not meet the specification). In Section 3.3.2 we mentioned that, in practice, specification-based detection approaches may lead to false positives or false negatives

because the same protocol might be implemented in different ways by vendors [110, 116, 118]

Differences in protocols' implementations by different operating systems have been exploited by attackers in order to evade detection by IDSs [156]. Attackers purposefully create traffic that is interpreted in different ways by end hosts that run different operating systems. Since the IDS might not know what is the operating system of the destination of an intercepted sequence of packets, it will not know whether the traffic constitutes an attack or not. One solution to this problem is to convert such ambiguous flows to unambiguous ones, so that there is only one well-behaved flow, interpreted identically by all endpoints [40]. This can be achieved by interposing a security mechanism between the external and the internal network that performs this conversion.

Two examples of run-time enforcement mechanisms that perform such a task are transport layer proxies and TCP scrubbers [40]. Both of these mechanisms convert ambiguous TCP flows to unambiguous ones, but they perform the conversion in different ways. Transport layer proxies interpose between a client and a server and create two connections: one between the client and the proxy, and one between the proxy and the server. On the other hand, TCP scrubbers leave the bulk of the TCP processing to the end points: they maintain the current state of the connection and a copy of packets sent by the external host but not acknowledged by the internal receiver. Fig. 4.1 (adapted from [40]) depicts the differences between the two mechanisms in a specific scenario. Although both mechanisms correctly enforce the same high-level "no ambiguity" policy, the proxy requires twice the amount of buffering as the scrubber, which suggests that the proxy is more costly (in terms of computational resources) [40].

Previous work started looking at cost as a metric to classify and compare such monitors. Drabik et al. introduced a framework that calculated the overall cost of enforcement based on costs assigned to the enforcement actions performed by the monitor [157]; this framework can be used to calculate and compare the cost of different monitors' implementations. This framework provides means to reason about cost-aware enforcement, but its enforcement model does not

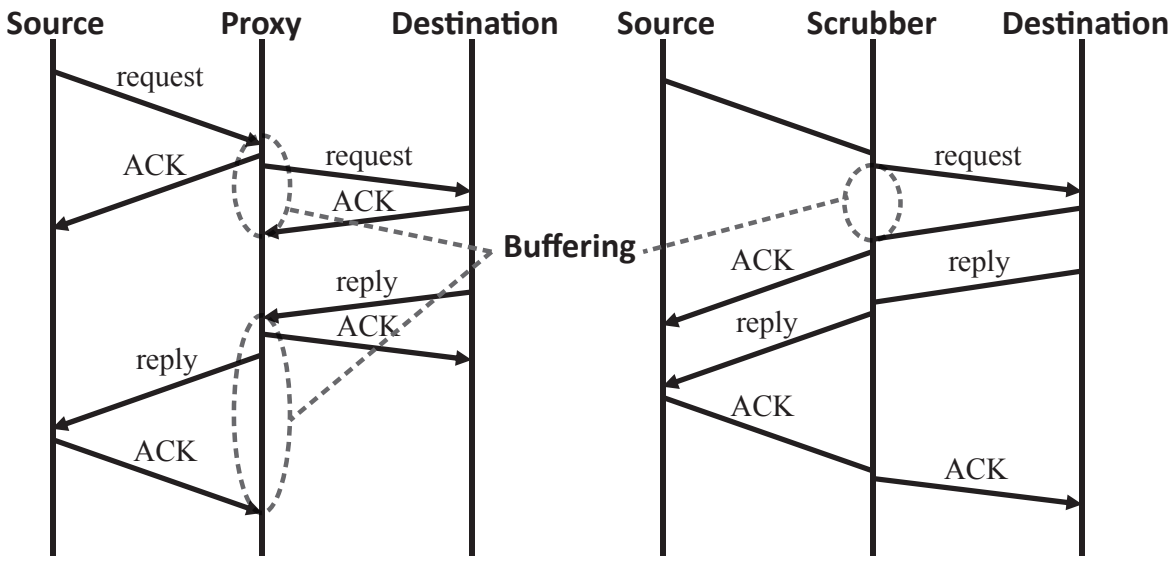


Figure 4.1: TCP transport layer proxies and scrubbers. The circled portions represent the amount of time that data is buffered.

capture interactions between the target and its environment, including the monitor; recent work has shown that capturing such interactions can be valuable [36]. In addition, in practice the cost of running an application may depend on the ordering of its actions, which may in turn depend on the scheduling strategy. Finally, one might also wish to ensure that a monitor enforces a cost policy, which defines which costs are acceptable; practical cost policies can depend on a probabilistic model of the system's behavior, e.g., take into account the likelihood of particular events. For example, a security policy that describes how to protect a system against different attacks might depend on the probability that these attacks, e.g., a DDOS attack or insider attack, will occur against that particular system.

The main contribution of this chapter is a formal framework that enables us to (1) model monitors that interact with probabilistic targets and environments (i.e., targets and environments whose behavior we can characterize probabilistically), (2) check whether such monitors enforce a given security policy, and (3) calculate and compare their cost of enforcement. More precisely:

1. Our framework is based on probabilistic I/O automata [154, 155]. This allows us to reason

about partially ordered events in distributed and concurrent systems, and the probabilities of events and sequences of events.

2. We extend probabilistic I/O automata with *abstract schedulers* to allow fair comparison of systems where a policy is enforced on a target by different monitors.
3. We define cost security policies and cost enforcement, richer notions of (boolean) security policies and enforcement [27]. Cost security policies assign a cost to each trace, allowing richer classification of traces than just as bad or good. We also show how to encode boolean security policies as cost security policies.
4. Finally, we show how to use our framework to compare monitors' implementations and we identify the sufficient conditions for constructing cost-optimal monitors.

4.2 Background

We introduce our notation in Section 4.2.1 and then in Section 4.2.2 briefly review probabilistic I/O automata (PIOA) [154, 155], which we build on in this chapter: more details can be found in standard PIOA references, e.g., [154, 155]. In Section 4.2.3 we extend PIOA by introducing the notion of *abstract schedulers*, which we use in the cost comparison of monitors in Section 4.5. Finally, in Section 4.2.4 we show how to use PIOA to model practical scenarios through a running example that we will use in the rest of the paper to illustrate the main ideas of our framework.

4.2.1 Preliminaries

We write $\mathbb{R}^{\geq 0}$ and $\mathbb{R}^{+}$ for the sets of nonnegative real numbers and positive real numbers respectively. Given a function $f : X \to Y$, we write $\mathsf{dom}(f)$ for the domain of f, i.e, X, and $\mathsf{range}(f)$ for the range of f, i.e., Y. Given some $X' \subseteq X$, we write $f \upharpoonright X'$ for the function whose domain is X' and range is $\{f(x) \mid x \in X'\}$.

A $\sigma-$field over a set X is a set $\mathcal{F} \subseteq 2^X$ that contains the empty set and is closed under complement and countable union. A pair $(X, \mathcal{F})$ where $\mathcal{F}$ is a $\sigma-$field over X, is called a *measurable space*. A measure on a measurable space $(X, \mathcal{F})$ is a function $\mu : \mathcal{F} \to [0, \infty]$ that is countably additive: for each countable family $\{X_i\}_i$ of pairwise disjoint elements of $\mathcal{F}$, $\mu(\cup_i X_i) = \Sigma_i \mu(X_i)$.

A *probability measure* on $(X, \mathcal{F})$ is a measure on $(X, \mathcal{F})$ such that $\mu(X) = 1$. A *sub-probability measure* on $(X, \mathcal{F})$ is a measure on $(X, \mathcal{F})$ such that $\mu(X) \leq 1$. A *discrete probability measure* on a set X is a probability measure μ on $(X, 2^X)$, such that, for each $C \subseteq X$, $\mu(C) = \sum_{c \in C} \mu(\{c\})$. A *discrete sub-probability measure* on a set X is a sub-probability measure μ on $(X, 2^X)$, such that, for each $C \subseteq X$, $\mu(C) = \sum_{c \in C} \mu(\{c\})$. We define $\mathsf{Disc}(X)$

and $\mathsf{SubDisc}(X)$ to be, respectively, the set of discrete probability measures and discrete sub-probability measures on X. In the sequel, we often omit the set notation when we refer to the measure of a singleton set.

A *support* of a probability measure μ is a measurable set C such that $\mu(C) = 1$. If μ is a discrete probability measure, then we denote by $\mathsf{supp}(\mu)$ the set of elements that have non-zero measure (thus $\mathsf{supp}(\mu)$ is a support of μ). We let $\delta(x)$ denote the *Dirac measure* for x, the discrete probability measure that assigns probability 1 to $\{x\}$.

A *signed measure* on $(X, \mathcal{F})$ is a function $\nu : \mathcal{F} \to [-\infty, \infty]$ such that: (1) $\nu(\emptyset) = 0$, (2) ν assumes at most one of the values $\pm\infty$, and (3) for each countable family $\{X_i\}_i$ of pairwise disjoint elements of $\mathcal{F}$, $\nu(\cup_i X_i) = \Sigma_i \mu(X_i)$ with the sum converging absolutely if $\nu(\cup_i X_i)$ is finite.

Given two discrete measures μ_1, μ_2 on $(X, 2^X)$ and $(Y, 2^Y)$, respectively, we denote by $\mu_1 \times \mu_2$ the *product measure*, that is, the measure on $(X \times Y, 2^{X \times Y})$ such that $\mu_1 \times \mu_2(x, y) = \mu_1(x) \cdot \mu_2(y)$ (i.e., component-wise multiplication) for each $x \in X$, $y \in Y$.

If $\{\rho_i\}_{i \in I}$ is a countable family of measures on $(X, \mathcal{F}_X)$ and $\{\rho_i\}_{i \in I}$ is a family of non-negative values, then the expression $\Sigma_{i \in I} p_i \rho_i$ denotes a measure ρ on $(X, \mathcal{F}_X)$ such that for each $C \in \mathcal{F}_X$), $\rho(C) = \Sigma_{i \in I} p_i \cdot \rho_i(C)$.

A function $f : X \to Y$ is said to be measurable from $(X, \mathcal{F}_X) \to (Y, \mathcal{F}_Y)$ if the inverse image of each element of $\mathcal{F}_Y$ is an element of $\mathcal{F}_X$; that is, for each $C \in \mathcal{F}_Y$, $f^{-1}(C) \in \mathcal{F}_X$. Note that, if $\mathcal{F}_X$ is 2^X, then any function $f : X \to Y$ is measurable $(X, \mathcal{F}_X) \to (Y, \mathcal{F}_Y)$ for any $(Y, \mathcal{F}_Y)$. Given measurable f from $(X, \mathcal{F}_X) \to (Y, \mathcal{F}_Y)$ and a measure μ on $(X, \mathcal{F}_X)$, the function $f(\mu)$ defined on $\mathcal{F}_Y$ by $f(\mu)(C) = \mu(f^{-1}(C))$ for each $C \in Y$ is a measure on $(Y, \mathcal{F}_Y)$ and is called the *image measure* of μ under f. If $\mathcal{F}_X = 2^X$, $\mathcal{F}_Y = 2^Y$, and μ is a sub-probability measure, then the image measure $f(\mu)$ is a sub-probability satisfying $f(\mu)(Y) = \mu(X)$.

4.2.2 Probabilistic I/O Automata

An *action signature* S is a triple of three disjoint sets of actions: *input*, *output*, and *internal* actions (denoted as *Input*(S), *Output*(S), and *Internal*(S)). The *external* actions *External*(S)=*Input*(S) $\cup$ *Output*(S) model the interaction of the automaton with the evironment. Given a signature S we write *acts*(S) for the set of all actions contained in the signature, i.e., *acts*(S) = *Input*$(S) \cup$ *Output*$(S) \cup$ *Internal*(S).

A probabilistic I/O automaton (PIOA) P is a tuple (*Sig*(P), *states*(P), $\bar{q}_P$, *trans*(P)), where: (1) *Sig*(P) is an action signature; (2) *states*(P) is a (possibly infinite) set of *states*; (3) $\bar{q}_P$ is a *start state*, with $\bar{q}_P \in$ *states*(P); and (4) *trans*$(P) \subseteq$ *states*$(P) \times$ *acts*$(P) \times \mathsf{Disc}($*states*$(P))$ is a transition relation, where $\mathsf{Disc}($*states*$(P))$ is the set of discrete probability measures on *states*(P).

Given a PIOA P, we write *acts*(P) for *acts*(*Sig*(P)). We assume that P satisfies the following conditions: (i) *Input enabling:* For every state $q \in$ *states*(P) and input action $\alpha \in$ *Input*(P), α is enabled[1] in q; and (ii) *Transition determinism:* For every state $q \in$ *states*(P) and action $\alpha \in$ *acts*(P), there is at most one $\mu \in \mathsf{Disc}($*states*$(P))$ such that $(q, \alpha, \mu) \in$ *trans*(P). If there exists exactly one such μ, it is denoted by $\mu_{q,\alpha}$, and we write $\mathsf{tran}_{q,\alpha}$ for the transition $(q, \alpha, \mu_{q,\alpha})$.

A *non-probabilistic execution e* of P is either a finite sequence, q_0, a_1, q_1, a_2, $\ldots$, a_r, q_r, or an infinite sequence q_0, a_1, q_1, a_2, $\ldots$, a_r, q_r, $\ldots$ of alternating states and actions such that: (1) $q_0 = \bar{q}_P$, and (2) for every non-final i, there is a transition $(q_i, a_{i+1}, \mu) \in$ *trans*(P) with $q_{i+1} \in \mathsf{supp}(\mu)$.

We write $\mathsf{fstate}(e)$ for q_0, and, if *e* is finite, we write $\mathsf{lstate}(e)$ for the last state of *e*. The *trace* of an execution *e*, written *traces*(e), is the restriction of *e* to the set of external actions of P. We say that *t* is a *trace* of P if there is an execution *e* of P such that *traces*$(e) = t$. We write $\mathsf{laction}(t)$ for the last action of *t*, when *t* is finite. We use *execs*(P) and *traces*(P) (resp., *execs*$^*(P)$ and

[1] If a PIOA P has a transition $(q, \alpha, \mu) \in$ *trans*(P) then we say that action α is *enabled* in state q.

traces$^*(P)$) to denote the set of all (resp., all finite) executions and traces of an PIO automaton P.

The symbol $\cdot$ denotes the empty sequence. We write $e_1; e_2$ for the concatenation of two executions the first of which has finite length and $\mathsf{lstate}(e_1) = \mathsf{fstate}(e_2)$. When σ_1 is a finite prefix of σ_2, we write $\sigma_1 \preceq \sigma_2$, and, if a strict finite prefix, $\sigma_1 \prec \sigma_2$. Given a finite set A, $(A)^\star$ denotes the set of finite sequences of elements of A and $(A)^\omega$ the set of infinite sequences of elements of A. The set of all finite and infinite sequences of elements of A is $(A)^\infty = (A)^\star \cup (A)^\omega$. Thus, for $A = \Sigma$ $(\Sigma)^\star$ denotes the set of finite sequences of actions and $(\Sigma)^\omega$ the set of infinite sequences of actions. The set of all finite and infinite sequences of actions is $(\Sigma)^\infty = (\Sigma)^\star \cup (\Sigma)^\omega$.

An automaton that models a complex system can be constructed by *composing* automata that model the system's components. When composing automata P_i, where $i \in I$ and I is finite their signatures are called *compatible* if their output actions are disjoint and the internal actions of each automaton are disjoint with all actions of the other automata. More formally, the actions signatures $P_i : i \in I$ or called compatible if for all $i, j \in I$: (1) $\mathit{Output}(P_i) \cap \mathit{Output}(P_j) = \emptyset$; (ii) $\mathit{Internal}(P_i) \cap \mathit{acts}(P_j) = \emptyset$. When the signatures are compatible we say that the corresponding automata and modules are compatible too. The composition $P = \prod_{i \in I} P_i$ of a set of compatible automata $\{P_i : i \in I\}$ is defined as:

1. $\mathit{Sig}(P) = \prod_{i \in I} \mathit{Sig}(P_i) = \Big(\mathit{Output}(P) = \cup_{i \in I} \mathit{Output}(P_i), \quad \mathit{Internal}(P) = \cup_{i \in I} \mathit{Internal}(P_i),$ $\mathit{Input}(P) = \cup_{i \in I} \mathit{Input}(P_i) - \cup_{j \in I} \mathit{Output}(P_j)\Big)$;
2. $\mathit{states}(P) = \prod_{i \in I} \mathit{states}(P_i)$;
3. $\bar{q}_P = \prod_{i \in I} \bar{q}_{P_i}$;
4. $\mathit{trans}(P)$ is equal to the set of triples $(q, a, \prod_{i \in I} \mu_i)$ such that:
   (a) a is enabled in some $q_i \in q$, and
   (b) for all $i \in I$ if $a \in \mathit{acts}(P_i)$ then $(q_i, a, \mu_i) \in \mathit{trans}(P_i)$, otherwise $\mu_i = \delta(q_i)$.

Schedulers. Nondeterministic choices in P are resolved using a *scheduler*. A *scheduler* for P is a function $\sigma : \mathit{execs}^*(P) \to \mathsf{SubDisc}(\mathit{trans}(P))$ s.t., if $(q, a, \mu) \in \mathsf{supp}(\sigma(e))$ then $q = \mathsf{lstate}(e)$. Thus, σ decides (probabilistically) which transition (if any) to take after each finite execution e. Since this decision is a discrete sub-probability measure, it may be the case that σ chooses to *halt* after e with non-zero probability: $1 - \sigma(e)(\mathit{trans}(P)) > 0$.

A scheduler σ together with a finite execution e *generates* a measure $\epsilon_{\sigma,e}$ on the $\sigma-$field $\mathcal{F}_P$ generated by cones of executions, where the cone $C_{e'}$ of a finite execution e' is the set of executions that have e' as prefix. The construction of the $\sigma-$field is standard [154]. The measure of a cone $\epsilon_{\sigma,e}(C_{e'})$ is defined recursively as:

1. 0, if $e' \not\preceq e$ and $e \not\preceq e'$;
2. 1, if $e' \preceq e$;
3. $\epsilon_{\sigma,e}(C_{e''})\mu_{\sigma(e'')}(a, q)$, if e' is of the form $e''\ a\ q$, $e \preceq e''$. Here, $\mu_{\sigma(e'')}(a, q)$ is defined to be $\sigma(e'')(\mathsf{tran}_{\mathsf{lstate}(e''),a})\mu_{\mathsf{lstate}(e''),a}(q)$, that is, the probability that $\sigma(e'')$ chooses a transition labeled by a and that the new state is q.

Standard measure theoretic arguments ensure that $\epsilon_{\sigma,e}$ is a probability measure. For full details the reader is referred to [154]. The measure of a cone of an execution e corresponds, intuitively, to the probability for e to happen.

We note that the *trace* function is a measurable function from $\mathcal{F}_P$ to the $\sigma-$field $\mathcal{F}_{P_T}$ generated by cones of traces. Thus, given a probability measure ϵ on $\mathcal{F}_P$, we define the *trace distribution* of ϵ, denoted $\mathsf{tdist}(\epsilon)$ to be the image measure of ϵ under *trace*, i.e., for each cone of traces C_t, $\mathsf{trace}(\epsilon)(C_t) = \epsilon(\mathsf{trace}^{-1}(C_t))$. We denote by $\mathsf{tdists}(P)$ the set of trace distributions of (probabilistic executions of) P.

4.2.3 Abstract Schedulers

In this section we introduce abstract schedulers, a novel extension of PIOA and one of the contributions of this thesis. Abstract schedulers are used in the cost comparison of monitors (Section 4.5). Given a signature S, an *abstract scheduler* τ for S is a function $\tau : (\mathit{External}(S))^* \rightarrow \mathsf{SubDisc}(\mathit{External}(S))$. τ decides (probabilistically) which action appears after each finite trace[2] *t*. Note that an abstract scheduler τ assigns probabilities to all possible (finite) traces over the given signature.

An abstract scheduler τ together with a finite trace *t generate* a measure $\zeta_{\tau,t}$ on the $\sigma-$field $\mathcal{F}_{P_T}$ generated by cones of traces, where the cone $C_{t'}$ of a finite trace t' is the set of traces that have t' as prefix. The measure of a cone $\zeta_{\tau,t}(C_{t'})$ is defined recursively as:

1. 0, if $t' \not\preceq t$ and $t \not\preceq t'$;
2. 1, if $t' \preceq t$;
3. $\zeta_{\tau,t}(C_{t''})\tau(t'')(\{a\})$, if t' is of the form $t''; a$, $t \preceq t''$.

Standard measure theoretic arguments ensure that $\zeta_{\tau,t}$ is well defined and a probability measure.

Refining abstract schedulers. Abstract schedulers give us (sub-)probabilities for all possible traces over a given signature. However, a given PIOA P might exhibit only a subset of all those possible traces. Thus, we would like to have a way to *refine* an abstract scheduler τ to a scheduler σ that corresponds to the particular PIOA P and is "similar" to τ w.r.t. assigning probabilities. This similarity can be made more precise as follows. First, if an abstract scheduler τ assigns a zero probability to a trace *t*, then this means that *t* cannot happen (e.g., the system stops due to overheating). Thus, even if *t* is a trace that P can exhibit, we would like σ to assign it a zero

[2]Note that the term "trace" is overloaded: it refers to either the result of applying the function trace to an execution *e* or to a sequence of external actions. It will be clear from the context to which of the two cases we refer each time.

probability. Second, assume we have a trace t that can be extended with actions a, b, or c, and an abstract scheduler τ that assigns a non-zero probability to all traces $t; X$, with $X \in \{a, b, c\}$ and $\tau(t)(X) = 1$, i.e., τ does not allow for the system to stop after t. If $t; a$ is a trace that P can exhibit, we would like σ to assign it the same probability as τ. However, if P cannot exhibit that trace, σ should assign it a zero probability. But then σ would be a sub-probability measure, i.e., it would allow for P to halt, whereas τ does not. To solve this problem, we proportionally re-distribute the probabilities that τ assigns to the traces that P can exhibit. These two cases are formalized as follows.

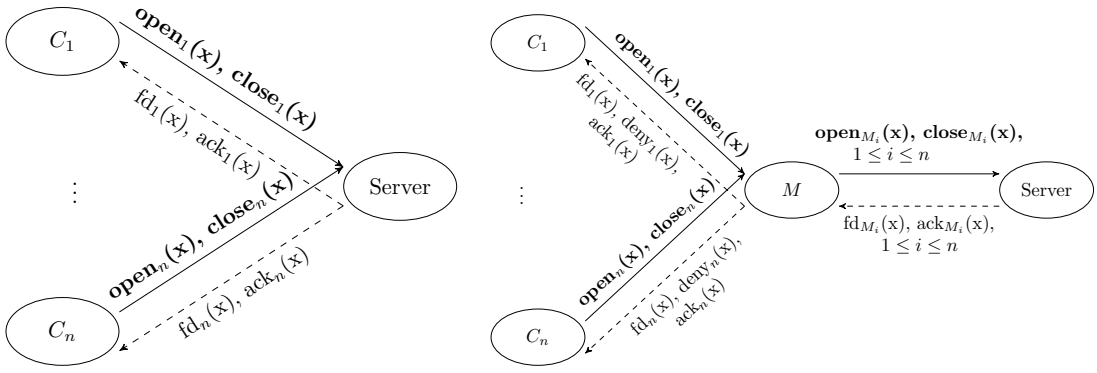
Given an *abstract scheduler* τ over a signature S, and a PIOA P with $\mathit{Sig}(P) = S$, we define the *refinement* function $\mathsf{refn}(\tau, P) = \tau'$, where $\tau' : (\mathit{External}(S))^* \rightarrow \mathsf{SubDisc}(\mathit{External}(S))$, i.e., a function that maps an abstract scheduler and a PIOA to another abstract scheduler, as follows:

Let $t = t'; a \in (\mathit{External}(S))^*$ in

- if $t \notin \mathit{traces}(P)$, then $\tau'(t')(\{a\}) = 0$;
- if $\tau(t')(\{a\}) = 0$, then $\tau'(t')(\{a\}) = 0$;
- otherwise, $\tau'(t')(\{a\}) = \frac{\tau(t')(\{a\})}{\left(\tau(t')(A)\right) + \left(1 - \tau(t')(\mathit{External}(S))\right)}$,
  where $A = \{x \in \mathit{External}(S) \mid t'; x \in \mathit{traces}(P)\}$.

Given an abstract scheduler τ and a PIOA P, standard measure theoretic arguments ensure that if τ together with a finite trace t generate a probability measure $\zeta_{\tau,t}$ on the $\sigma-$field $\mathcal{F}_{P_T}$ generated by cones of traces, so does the abstract scheduler $\mathsf{refn}(\tau, P)$, i.e., it generates a probability measure $\zeta'_{\mathsf{refn}(\tau,P),t}$ on the $\sigma-$field $\mathcal{F}_{P_T}$.

We now formalize the relationship between schedulers and abstract schedulers. Given an abstract scheduler τ over a signature S, and a PIOA P with $\mathit{Sig}(P) = S$, a scheduler σ is *derivable* from τ iff σ is a scheduler for P such that for all executions $e \in \mathit{execs}(P)$ the trace distributions



(a) Clients and server (b) Clients and monitored server

Figure 4.2: Diagrams of interposing a Monitor between Clients and Server

of $\epsilon_{\sigma,e}$ are equal to the probability measures of $\mathsf{trace}(e)$ assigned by the refinement of τ on P, i.e., for all executions $e, e'' \in \mathit{execs}(P)$, $\mathsf{tdist}(\epsilon_{\sigma,e})(C_{e''}) = \zeta'_{\mathsf{refn}(\tau,P),\mathsf{trace}(e)}(C_{\mathsf{trace}(e'')})$.

4.2.4 Running Example Modeled Using PIOA

To illustrate how our framework can be used to model enforcement scenarios we will consider a running example of a file server S, illustrated in Fig. 4.2a.

Clients (C_1 through C_n in the figure) can request to open or close a particular file. The server responds to the requests by returning a file descriptor or an acknowledgment that the file was closed successfully. Given a security policy P stating that at most one client at a time can access a particular file, a monitor is interposed between the clients and the server to enforce P as shown in Fig. 4.2b. The monitor has the ability to *deny* access to a file requested by a client. A *system* consists of communicating clients, monitor, and server.

We now show how to model the running example (Fig. 4.2b) using PIOA. Each client C_i requests to open a file x through an $\mathit{open}_i(x)$ output action. Once he receives a file descriptor through an $\mathit{fd}_i(x)$ input action, he requests to close the file through an $\mathit{close}_i(x)$ action. When he receives an acknowledgment that the file was closed, he enters a state from which he stops re-

Signature: Input: $fd_i(x)$, where x is a filename
$ack_i(x)$, where x is a filename
$deny_i(x)$, where x is a filename
Output: $open_i(x)$, where x is a filename
$close_i(x)$, where x is a filename

States: $p \in \{0, 1, 2, 3\}$

Start States: $p = 0$

Transitions:
$fd_i(x)$
Effect: $p := 2$

$ack_i(x)$
Effect: $p = 1$, i.e., the client stops requesting to open the file

$deny_i(x)$
Effect: $p \in \{0, 1\}$, with $\mu(0) = 0.1$, and $\mu(1) = 0.9$ if i is even, i.e., even-indexed clients request to open the file, if access is denied, with probability 0.1, whereas odd-indexed ones with 0.9

$open_i(x)$
Precondition: $p = 0$
Effect: $p := 1$

$close_i(x)$
Precondition: $p = 2$
Effect: $p := 3$

Figure 4.3: Client$_i$ PIOA definition

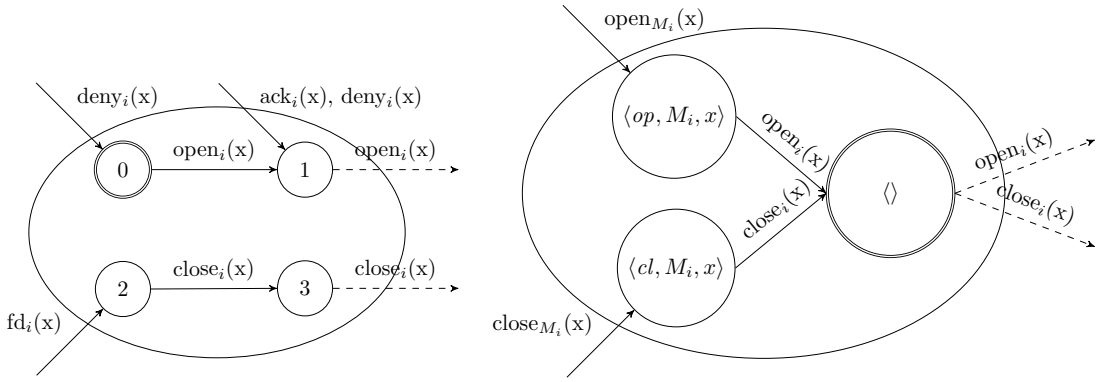
questing access to the file. If, however, he is denied access to the file, he decides probabilistically to either enter a state from which it will request to open the file again, or stop requesting.

The pseudocode[3] for C_i is depicted in Fig. 4.3 and a state diagram in Fig. 4.4a. The ellipse represents the communication interface of the automaton and the circles the automaton's states. Inputs are depicted as arrows entering the automaton, and we only show the effect of the action, i.e., the automaton's end state. Each output action is depicted with two arrows: (1) a straight arrows between states, to depict the precondition and effect on states; and (2) a dashed arrow to show that action becomes visible outside the automaton. The server S implements a stack of size one: it replies with a file descriptor or an acknowledgment of closing a file for the latest request. This means that if a scheduler allows two requests to arrive before the server is given a chance to reply, then the first request is ignored and the last request is served. The pseudocode for S is depicted in Fig. 4.5 and a state diagram in Fig. 4.4b.

To further illustrate some of the capabilities of our framework we introduce two example types of monitor:

- M_{DENY} always denies access to a file that is already open;

[3] We use the precondition pseudocode style that is typical in I/O automata papers (e.g., [154]).



(a) Client PIOA state transition diagram (b) Server PIOA state transition diagram

Figure 4.4: State transition diagrams if Client and Server

Signature: Input: $open_{M_i}(x)$, where x is a filename
$close_{M_i}(x)$, where x is a filename
Output: $fd_{M_i}(x)$, where x is a filename
$ack_{M_i}(x)$, where x is a filename

States: $p = \langle x, y\rangle$, where $x \in \{op, cl\}$, $y \in \{M_i\}_{0 \le i \le n}$

Start States: $p = nil$

Transitions:
$open_{M_i}(x)$
Effect: $p := \langle op, M_i, x\rangle$
$close_{M_i}(x)$
Effect: $p := \langle cl, M_i, x\rangle$
$fd_{M_i}(x)$
Precondition: $p = \langle op, M_i, x\rangle$
Effect: $p := nil$
$ack_{M_i}(x)$
Precondition: $p = \langle cl, M_i, x\rangle$
Effect: $p := nil$

Figure 4.5: Server PIOA definition

- M_{PROB} uses probabilistic information about future requests to make decisions. More precisely, a client i is always granted a request to open a file that is available. Otherwise, if the file is unavailable, i.e., a client j has already opened it, the monitor checks whether (1) after force-closing the file for j, j will ask to re-open the file with probability less than 0.5; and (2) after denying access to i, i will re-ask with probability greater than 0.5. If both hold, the monitor gives access to i; otherwise it denies access.

The pseudocode for M_{DENY} is depicted in Fig. 4.6. The pseudocode for M_{PROB} is depicted in Fig. 4.8. To point out the high level differences in the two monitors with repsect to deciding when to accept or deny a request we provide high-level decision diagrams in Fig. 4.7a and Fig.4.7b.

Let us now consider the composed system $\Pi = C_1 \times \ldots \times C_n \times M \times S$. The states of the

Signature: Input: $open_i(x)$,$close_i(x)$,
$fd_{M_i}(x)$, $ack_{M_i}(x)$,
where x is a filename
Output: $open_{M_i}(x)$, $close_{M_i}(x)$,
$fd_i(x)$, $ack_i(x)$, $deny_i(x)$,
where x is a filename

States: p: list (of triples) of requests
from clients to monitor
q: list (of triples) of responses
from monitor to clients
r: list (of pairs) of active
connections

Start States: $p = q = r = nil$

Transitions:
$open_i(x)$
Effect: $p := p@[\langle op, i, x\rangle]$
$close_i(x)$
Effect: $p := p@[\langle cl, i, x\rangle]$
$fd_{M_i}(x)$
Effect: $q := q@[\langle fd, M_i, x\rangle]$
$ack_{M_i}(x)$
Effect: $q := q@[\langle ack, M_i, x\rangle]$
$open_{M_i}(x)$
Precondition: $p = \langle op, i, x\rangle :: p'$
$\wedge\ \nexists\langle x, j\rangle \in r, j \neq i$
Effect: $p := p'$
$r := r@[\langle x, i\rangle]$
$close_{M_i}(x)$
Precondition: $p = \langle cl, i, x\rangle :: p'$
Effect: $p := p'$
$r := r\backslash[\langle x, i\rangle]$
$fd_i(x)$
Precondition: $q = \langle fd, M_i, x\rangle :: q'$
Effect: $q := q'$
$ack_i(x)$
Precondition: $q = \langle ack, M_i, x\rangle :: q'$
Effect: $q := q'$
$deny_i(x)$
Precondition: $p = \langle op, i, x\rangle :: p'$
$\wedge\exists\langle x, j\rangle \in r, j \neq i$
Effect: $p := p'$

Figure 4.6: M_{DENY} PIOA definition

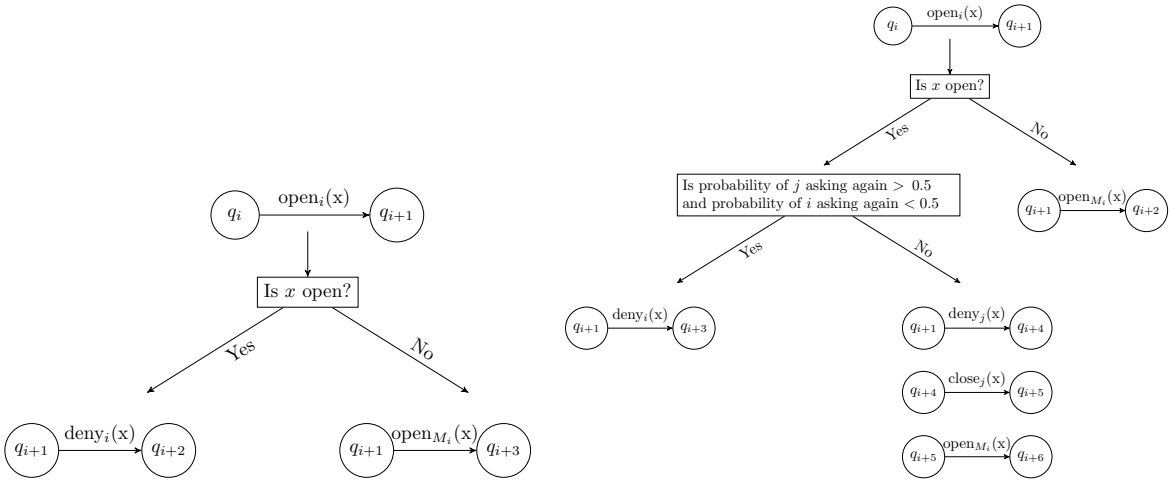
composed system will be $n + 2-$ tuples of the form $q_\Pi = \langle q_{C_1}, \ldots, q_{C_n}, q_M, q_S\rangle$. Two example executions for M_{DENY} are:

- $e_{M_{DENY}} = q_{\Pi_0}\ open_1(x)\ q_{\Pi_1}\ open_{M_1}(x)\ q_{\Pi_2}\ fd_{M_1}(x)\ q_{\Pi_3}\ fd_1(x)\ q_{\Pi_4}\ open_2(x)\ q_{\Pi_5}\ deny_2(x)\ q_{\Pi_6}\ open_2(x)\ q_{\Pi_7}\ deny_2(x)\ q_{\Pi_8}$
- $e'_{M_{DENY}} = q_{\Pi_0}\ open_1(x)\ q_{\Pi_1}\ open_{M_1}(x)\ q_{\Pi_2}\ fd_{M_1}(x)\ q_{\Pi_3}\ fd_1(x)\ q_{\Pi_4}\ open_3(x)\ q_{\Pi_5}\ deny_3(x)\ q_{\Pi_6}\ open_3(x)\ q_{\Pi_7}\ deny_3(x)\ q_{\Pi_8}$

The corresponding traces are:

- $t_{M_{DENY}} = trace(e_{M_{DENY}}) = open_1(x)\ open_{M_1}(x)\ fd_{M_1}(x)\ fd_1(x)\ open_2(x)\ deny_2(x)\ open_2(x)\ deny_2(x)$
- $t'_{M_{DENY}} = trace(e'_{M_{DENY}}) = open_1(x)\ open_{M_1}(x)\ fd_{M_1}(x)\ fd_1(x)\ open_3(x)\ deny_3(x)\ open_3(x)\ deny_3(x)$

In $t_{M_{DENY}}$ client C_1 asks to open file x and he is assigned the file, and after that client C_2 asks to open the same file and is denied access by the monitor. $t'_{M_{DENY}}$ is a similar scenario, with the difference that the client that asks access to x the second time is C_3, i.e., an odd-indexed client.



(a) Monitor M_{DENY} (b) Monitor M_{PROB}

Figure 4.7: Decision Diagrams

Let us consider the scheduler σ that schedules transitions based on the following high-level pattern: $\Big([C_1, \ldots, C_n];\ M^*;\ S;\ M^*;\Big)^\omega$. This pattern says that σ chooses probabilistically one of the clients to execute some transition, and then, deterministically, the monitors gets a chance to execute as many actions as it needs, then the server responds with one transition, and finally the monitor gets again the chance to do as much work as it needs. This pattern repeats finitely, or infinitely, many times.

Let us assume that σ chooses each client to take a turn with probability $P(C_i) = \frac{1}{n}$. Then the probability of $e_{M_{DENY}}$ is given by the measure $\epsilon_{\sigma,\bar{q}}$ on the cone of executions that have $e_{M_{DENY}}$ as prefix, i.e., $\epsilon_{\sigma,\bar{q}}(C_{e_{M_{DENY}}})$. It is easy to calculate that $\epsilon_{\sigma,\bar{q}}(C_{e_{M_{DENY}}}) = \frac{0.1}{n^2}$. We calculate the probabilities of $t_{M_{DENY}}$ and $t'_{M_{DENY}}$ as follows. We know that $\mathsf{tdist}(\epsilon_{\sigma,\bar{q}})(C_{t_{M_{DENY}}}) = \mathsf{trace}(\epsilon_{\sigma,\bar{q}})(C_{t_{M_{DENY}}})$ $= \epsilon_{\sigma,\bar{q}}(\mathsf{trace}^{-1}(C_{t_{M_{DENY}}}))$. Note that, from the PIOA definitions of (the states of) the components of Π, there is a bijective mapping between the execution $e_{M_{DENY}}$ and the trace $t_{M_{DENY}}$, i.e., the set of executions that trace^{-1} maps $C_{t_{M_{DENY}}}$ to, is the cone $C_{e_{M_{DENY}}}$. Thus, the probability of $C_{t_{M_{DENY}}} = \frac{0.1}{n^2}$. The same holds for calculating the probability of $t'_{M_{DENY}}$.

Signature: Same as M_{DENY}

States: Same as M_{DENY} and
$force = \langle x, y, z \rangle$, where
$x \in \{0, 1, 2, 3\}, 0 \leq y, z \leq n$

Start States: Same as M_{DENY} and
$force = \langle 0, \_, \_ \rangle$

Transitions: Same as M_{DENY} with the following changes:

$open_i(x)$
Effect: if $\exists \langle x, j \rangle \in r, j \neq i$
$\wedge$ probability of j asking again < 0.5
$\wedge$ probability of i asking again > 0.5
then $force := \langle 1, i, j \rangle$
else $p := p@[\langle op, i, x \rangle]$

$open_{M_i}(x)$
Precondition: $\big(p = \langle op, i, x \rangle :: p'$
$\wedge (\nexists \langle x, \_ \rangle \in r$
$\vee force = \langle 3, i, x \rangle\big)$
Effect: $p := p'$
$r := r@[\langle x, i \rangle]$
$force := \langle 0, \_, \_ \rangle$

$close_{M_i}(x)$
Precondition: $q = \langle ack, M_i, x \rangle :: q'$
$\vee force = \langle 2, x, i \rangle$
Effect: $q := q'$
$force := \langle 3, x, i \rangle$

$ack_i(x)$
Precondition: $q = \langle ack, M_i, x \rangle :: q'$
Effect: $q := q'$

$deny_i(x)$
Precondition: $(p = \langle op, i, x \rangle :: p'$
$\wedge \exists \langle x, j \rangle \in r, j \neq i$
$\vee force = \langle 1, x, i \rangle$
Effect: $p := p'$
$force := \langle 2, x, i \rangle$

Figure 4.8: M_{PROB} PIOA definition

If we were to consider the execution $e''_{M_{DENY}} = e_{M_{DENY}}\ open_2(x)\ q_{\Pi_5}\ open_{M_2}(x)\ q_{\Pi_6}$, i.e., C_2 tries to access the file again and the monitor gives access to the file, then $\epsilon_{\sigma,\bar{q}}(C_{e''_{M_{DENY}}}) = 0$ (similarly for $t''_{M_{DENY}} = t_{M_{DENY}}; open_2(x)$).

4.3 Probabilistic Cost of Automata

In this section we develop the framework to reason about the cost of an automaton P.

A cost function assigns a real number to every trace over a signature S, i.e., every possible sequence of external actions of S. More formally:

Definition 18. *A* cost function cost *over a signature* S *is a signed measure on the* $\sigma-$*field* $\mathcal{F}_{\tilde{P}_T}$ *generated by cones of traces of an automaton* $\tilde{P}$ *with* $Sig(\tilde{P}) = S$ *that generates all possible traces over its signature*[4]*, i.e.,* $\mathsf{cost} : \mathcal{F}_{\tilde{P}_T} \to [-\infty, \infty]$.

Remember that a cone C_t of a finite trace *t* is the set of traces that have *t* as prefix. Thus, there is a one-to-one correspondence between traces and the cones (of traces) they infer. Although traces are the subject of our analysis, cones are their (sound) mathematical representation.

Note that in this chapter we use the word cost to refer to both expenses and profits. For instance, one can use positive values of costs to possibly represent something you pay (i.e., an expense) and negative values of costs to represent something you gain (i.e., profit). The use of the word when describing practical examples will be clear from the context.

We calculate the expected cost of a trace, called *probabilistic cost*, by multiplying the probability of the trace with its cost. More formally:

Definition 19. *Given a scheduler* σ *and a cost function* cost*, the* probabilistic cost of a cone of a trace C_t *is defined as* $\mathsf{pcost}_\sigma(C_t) = \big(\epsilon_{\sigma,\bar{q}}(\mathsf{trace}^{-1})(C_t)\big)\mathsf{cost}(C_t)$.

Probabilistic costs of traces can be used to assign expected costs to automata: the probabilistic (i.e., expected) cost of an automaton is the set of probabilistic costs of its traces. However, it is often useful for the cost to be a single number, rather than a set. For example, we might want to build a monitor that does not allow a system to overheat, i.e., it never goes above a

[4]Given a signature S one can construct such an automaton $\tilde{P}$ by using a single state and a self-looped labeled transition for each action in the signature.

threshold temperature. In this case the cost of an automaton (e.g., the composition of the monitor automaton with the system automaton) could be the maximal cost of all traces. Similarly, we might want to build a monitor that cools down a system, i.e., lowers a system's temperature below a threshold, infinitely often. Here we could assign the cost of an automaton to be the minimal cost that appears infinitely often in its (infinite) set of traces, and check whether that cost is smaller than the threshold. It is clear that it can be beneficial to abstract the function that maps sets of probabilistic costs of traces to single numbers. We formalize this as follows.

Definition 20. *Given a scheduler σ and a cost function* cost*, the* probabilistic cost of a PIOA P *is defined as* $\mathsf{pcost}^{\mathbb{F}}_{\sigma}(P) = \mathbb{F}_{t \in traces(P)}(\mathsf{pcost}_{\sigma}(C_t))$.

Note that the definition is parametric in the function $\mathbb{F}$.

As an example, consider the infinite set $v = \{v_0, v_1, \ldots\}$, where each v_i is the probabilistic cost of some trace of P (ranging over a finite set of possible costs); then, $\mathbb{F}$ could be (following definitions of Chatterjee et al. [158]):

- $\mathsf{Sup}(v) = sup\{v_n \mid n \geq 0\}$, or
- $\mathsf{LimInf}(v) = \mathit{liminf}_{n\to\infty} v_n = \mathit{lim}_{n\to\infty} inf\{v_i \mid i \geq n\}$.

Sup chooses the maximal number that appears in v (e.g., the maximal temperature that a system can reach). LimInf function chooses the minimal number that appears infinitely often in v (e.g., the temperature that the system goes down to infinitely often).

For the rest of the chapter we will assume that $\mathbb{F}$'s arguments are sets of pairs $\langle t, c\rangle$, where the first component t is a trace and the second component c is (typically) the expected cost of t. This assumption will simplify the presentation and analysis of our results.

Note that if we were to assign costs to actions r_1 and r_2, e.g., 2 and 5 respectively, then cost can assign different numbers to their interleavings that might not clearly relate to the costs of the actions, e.g., $\mathsf{cost}(r_1; r_2) = 0$ and $\mathsf{cost}(r_2; r_1) = 20$.

Next, we show how one can define the cost of a system given cost functions for its components.

Assume that we have cost functions for the clients, the monitor, and the server (resp. cost_{C_i}, cost_M, and cost_S). The *cost of a trace t of the system* Π (i.e., the composition of the clients, monitor, and server) is the sum of the probabilistic costs of each component, whenever that component is allowed to take a step in the trace.

More formally[5]:

- $\mathsf{Cost}(C_\cdot)$[6] $= \mathsf{cost}_{C_i}(C_\cdot) + \mathsf{cost}_M(C_\cdot) + \mathsf{cost}_S(C_\cdot)$.
- $\mathsf{Cost}(C_t) = \mathsf{Cost}(C_{t'}) + \mathit{diff}_X(C_{t';a})$, where $X \in \{C_i, M, S\}$, $t = t'; a$, $a \in \mathit{acts}(X)$, and $\mathit{diff}_X(t'; a) = \mathsf{cost}_X(C_{t';a}) - \mathsf{cost}_X(C_{t'})$.

The probabilistic cost of an trace and the probabilistic cost of an automaton are now defined, as above, using the cost function Cost.

One can alternately define the cost of a system based on costs assigned to smaller components. For example, we can define cost functions over individual actions (or transitions) and use them to define the cost of an trace, and the cost of an automaton. In this case, the cost of traces would be independent of the interleaving of transitions (unless we define costs over transitions and use some coding trick, e.g., allow states to encode the trace history). This approach is similar to the definition of Cost so we do not pursue it here. Note that such an approach can be used to embed the framework of Drabik et.al. in ours [157].

[5]We use addition as the function between costs of actions in the trace. One can define appropriate cost functions by using other functions $\mathbb{G}$ from costs to costs, but we do not pursue it in this thesis.

[6]Remember, $\cdot$ denotes the empty trace.

4.4 Cost Security Policy Enforcement

In this section we define security policies and what it means for a monitor to enforce a security policy on a system.

Cost security policies. A monitor M is a PIOA. A monitor mediates the communication between system components S_i which are also PIOA. Thus, the the output actions of each S_i are inputs to the monitor, and the monitor has corresponding outputs that it forwards to the other components. More formally, given an index set I and a set of components $\{S_i\}$, $i \in I$, we assume that $\mathit{acts}(S_i) \cap \mathit{acts}(S_j) = \emptyset$, for all $i, j \in I$, $i \neq j$. Our goal is to model and reason about the external behavior of the monitored system. Thus, we also assume that $\mathit{Internal}(S_i) = \emptyset$, for all $i \in I$. Since the system components S_i are compatible, we will refer to their composition $\Pi_{i \in I} S_i$ as system S. A monitored system is the PIOA that results from composing M with S.[7]

The cost function defined in Section 4.3 describes the impact of a monitor on a system. A cost function is not necessarily bound to a specific security policy, which allows for the analysis of the same monitor against different policies. In practice, a monitor's purpose is to ensure that some policy is respected by the monitored system. In the running example, the monitor's role is to ensure that a file is not simultaneously open by two clients. Furthermore, since each $deny$ action comes with a cost, it is desirable for the cost of monitoring to be limited. This motivates the need to define a cost security policy.

Definition 21. *Given a (monitored) system* $(M \times S)$*, a* cost security policy Pol *over* $\mathit{Sig}(M \times S)$ *is a cost function over* $\mathit{Sig}(M \times S)$*, i.e., a signed measure* Pol *on the* σ*–field* $\mathcal{F}_{\tilde{P}_T}$[8] *generated by cones of traces of the system, i.e.,* $\mathsf{Pol} : \mathcal{F}_{\tilde{P}_T} \rightarrow [-\infty, \infty]$.

[7] By assumption, M and S are compatible. In scenarios where this is not the case, one can use renaming to make the automata compatible [36, 154, 155].

[8] Remember $\tilde{P}$ is the automaton that has the same signature as $(M \times S)$, in this case, and produces all possible traces over its signature.

When we talk about the signature, actions, etc. of Pol, we refer to the signature, actions, etc of P. Cost security policies associate a cost with each trace. For instance, if a trace *t* corresponds to a particular enforcement interaction between a monitor and a client, then $\mathsf{Pol}(C_t) = 10$ could describe that such enforcement (i.e., *t*) is allowed only if its cost is less than 10. Our definition of policies extends that of security properties [27]: security properties are predicates, i.e., binary functions, on sets of traces, whereas we focus on policies that are functions whose range is the real numbers (as opposed to $\{0, 1\}$). We leave the investigation of enforcement for securities policies defined as sets of sets of traces (e.g., [27, 36, 159]) for future work.

Definition 22. *Given a cost security policy* Pol *and a scheduler* σ *the* probabilistic cost security policy pPol_σ *under* σ *is defined as* $\mathsf{pPol}_\sigma(C_t) = \big(\epsilon_{\sigma,\bar{q}}(\mathsf{trace}^{-1})(C_t)\big)\mathsf{Pol}(C_t)$.

Cost security policy enforcement. In Section 4.3 we showed how to calculate the expected cost of a trace of an automaton, and the expected cost of an automaton. In the previous paragraph we defined the notion of a (probabilistic) cost security policy. We will now define what does it mean for a monitor to enforce a cost security policy on a system.

Definition 23. *Given a scheduler* σ*, a cost function* cost*, a policy* Pol*, a function* $\mathbb{F}$*, a monitor* M*, and a system* S *(compatible with* M*), we say that* M $n-\text{enforces}_{\leq}$ *(resp.,* $n-\text{enforces}_{\geq}$*)* Pol *on* S *under* σ*,* $\mathbb{F}$*, and* cost *if and only if the probabilistic cost of the monitored system differs by at most* n *from the probabilistic cost that the policy assigns to the traces of the monitored system, i.e.,:*

$\Big(\mathsf{pcost}^{\mathbb{F}}_{\sigma}(M \times S)\Big) - \Big(\mathbb{F}_{t \in traces(M\times S)}\mathsf{pPol}_\sigma(C_t)\Big) \leq n$ *(resp.,* $\geq n$*), i.e.,*

$\Big(\mathbb{F}_{t \in traces(M\times S)}\mathsf{pcost}_\sigma(C_t)\Big) - \Big(\mathbb{F}_{t \in traces(M\times S)}\mathsf{pPol}_\sigma(C_t)\Big) \leq n$ *(resp.,* $\geq n$*).*

We say that a monitor M *enforces*$_{\leq}$ (resp., *enforces*$_{\geq}$) a security policy P on a system S under a function $\mathbb{F}$, a scheduler σ, and a cost function cost if and only if M $0-\text{enforces}_{\leq}$ (resp., $0-\text{enforces}_{\geq}$) P on S under $\mathbb{F}$, σ, and cost.

The definition of enforcement says that a monitor M enforces a policy Pol on a system S if the probabilistic cost of the monitored system under some scheduler σ and cost function cost is less or equal (resp. greater or equal) than the cost that the policy assigns to the behaviors that the monitored system can exhibit. We define enforcement using two comparison operators because different scenarios might assign different semantics to the meaning of enforcement: One might use a monitor to maximize the value of a monitored system with respect to some base value, e.g., in our running example, we may want to give access to as many unique clients as possible since the server is making extra money by delivering advertisements to them; thus, the monitor has motive to give priority to every new request for accessing a file. In other cases, one might use a monitor to minimize the cost of the monitored system with respect to some allowed cost, e.g., we might want to minimize the state that the monitor and the server keep to provide access to files, in which case caching might be cost-prohibitive. Without loss of generality in this thesis we focus on $\leq$; similar results hold for $\geq$.

Enforcement is defined with respect to a global function $\mathbb{F}$. $\mathbb{F}$ transforms the costs of all traces of a monitored system to a single value. As described in Section 4.3, this value could represent the minimum cost of all traces, their average, sum, etc. Thus, $\mathbb{F}$ can model situations where an individual trace might have cost that is cost-prohibited by the policy (e.g., overheating temporarily), but the monitored system as a whole is still within the acceptable range (i.e., before and after the overheating the system cools down enough).

In the previous instantiation of our running example, there might exist some trace t where $\mathsf{cost}(t) > \mathsf{Pol}(t) > -\infty$, typically when a client keeps asking for a file that is denied. Although this would intuitively mean that the cost security policy is not respected for that particular trace, it might be the case that M enforces Pol, as long as Pol is globally respected, which could happen, e.g., if the probability of t is small enough. This illustrates a strength of our framework: we can allow for some local deviations, as long as they do not impact the global properties, i.e.,

overall expected behavior, of the system. If we wish to constrain each traces, we can define *local enforcement*, which requires that the cost of *each trace* of the monitored system is below (or above) a certain threshold, as opposed to enforcement which requires that the value of some function computed over *all traces* of the monitored system is below (or above) a certain threshold. Note that local enforcement can be expressed through a function $\mathbb{F}$ that universally quantifies the cost difference from the threshold over all traces of the monitored system. Local enforcement could be useful, for example, to ensure that a system *never* overheats even momentarily, whereas enforcement would be useful if we want to have probabilistic guarantees of the system; e.g., we accept a 0.001% probability that the system will become unavailable due to overheating.

A question a security designer might have to face is whether it is possible, given a boolean security policy that describes what should not happen and a cost policy that describes the maximal/minimal allowed cost, to build a monitor that satisfies both. This problem can help illuminate a common cost/security tradeoff: the more secure a mechanism is, the more costly it usually is.

There is a close relationship between boolean security policies (e.g., [27]) and cost security policies: given a boolean security policy there exists a cost security policy such that if the cost security policy is $n-$enforceable then the boolean security policy is enforceable as well (and vice versa). Specifically, given a boolean security policy P, we write Pol_P for the function such that $\mathsf{pPol}_P(C_t) = 0$ if $P(t)$ holds, and $-\infty$ otherwise. Given a predicate P, if we instantiate function $\mathbb{F}$ with the function that returns the least element of a set and function cost with the function that maps every (trace) cone to 0, and if M $0-\text{enforces}_{\leq}$ Pol_P, then any trace belongs to P. In other words, our framework is a generalization of the traditional enforcement model.

In the other direction, since cost security policies are more expressive than boolean security policies, we need to pick a bound that will serve as a threshold to classify traces as acceptable or not. Given a probabilistic cost security policy pPol, a cost function cost, a scheduler σ and a bound $n \in \mathbb{R}$, we say that a trace *t satisfies* $\mathsf{Pol}_{\mathsf{cost},n,\sigma}$, and write $\mathsf{Pol}_{\mathsf{cost},n,\sigma}(t)$ if and only if

$\mathsf{pPol}(C_t) \geq \mathsf{pcost}_\sigma(C_t) - n$.

Expressing cost security policies as boolean security policies allows one to embed in our framework a notion of sound enforcement [30]: a monitor is a sound enforcer for a system S and security policy P if the behavior of the monitored system obeys P. As described above, one encodes P in our framework as Pol_P, which returns $-\infty$ if a trace violates P and 0 otherwise. Sound enforcement can be expressed as $0-\text{enforcement}_{\leq}$ using a global function $\mathbb{F}_P$ that assigns $-\infty$ to the cost of the automaton composition that represents the monitored system if some trace has cost $-\infty$, and 0 otherwise. Specifically, if a monitor soundly enforces P on a system, all its traces will belong to P and Pol_P will map them all to 0, which when applied to $\mathbb{F}_P$, will result in a global cost of 0. If the monitor is not sound, then the global cost will be $-\infty$. Thus, a monitor soundly enforces a boolean security policy P if and only if the monitor $0-\text{enforces}_{\leq}$ the cost security policy Pol_P under $\mathbb{F}_P$ and $\mathsf{cost}(\_) = 0$.

Transparent cost enforcement. Assume that we are given a scheduler σ, a cost function cost, a function $\mathbb{F}$, a policy Pol, and a system S and we want to $n-\text{enforce}_{\leq}$ Pol on S under σ, $\mathbb{F}$, and cost. As we explained in the introduction, different monitors may be able to achieve this goal. One such choice might be a monitor that acts as a sink: it never forwards messages between the system's components. Thus,interaction-intensive systems, i.e., systems that rely heavily on interaction to achieve their goals, will only exhibit a minimal behavior. As such, the cost of such a monitored system will be (close to) zero (assuming that components limit their non-interactive actions–cf. *quiescent forgiveness* in [36]). However, such monitors are unlikely to be useful in practice.

Previous work on run-time enforcement has identified (similar) situations where trivial monitors enforce (boolean) policies by consuming all inputs and denying all actions the target system wants to execute [30, 82, 103]. *Transparency* is one notion that aids in ruling out such uninterest-

ing cases: if the target system wants to perform an action that obeys the policy, then the monitor must allow it. Most definitions of transparency that have been introduced so far are within frameworks where policies reason only about the target's behavior [27, 30]: a policy is a predicate over traces of the target (i.e., a subset of the traces that the target might exhibit) and not over traces that the monitored target can exhibit through the interaction of the monitor with the target.

In this chapter we take a (more general) view, that has been recently introduced, which allows policies to describe how monitors are allowed to react to target's requests [36, 59] (in addition to considering policies which reason about cost). Thus, enforcement is now implicit in the definition of a policy (i.e., in the traces that the policy allows). This means that we can define transparency as a specific type of interaction between the target and the monitor.

Since our definition of policies is more expressive than previous ones with respect to the interaction between the target and the monitor, to talk about transparent enforcement we first need to encode previous definitions of policies (i.e., sets of traces by a target) in our framework (see Section 2.4.1). The main idea is as follows: given a policy that describes which target's traces are allowed, we build a policy (over the monitored target) in which every valid trace of the target is forwarded by the monitor to the environment (and vice versa). The way that the forwarding can be achieved by the monitor depends on the notion of fairness that we assume in the model: if the monitor is not allowed to finish forwarding the valid trace of the target then it will not achieve transparent enforcement, even though the same monitor under different circumstances would be transparent. Thus, we need to choose a notion of transparency depending on whether we assume fairness in our model or not (see Section 2.4.1). In the framework of this chapter (weak) fairness can be encoded by schedulers that do not assign zero probabilities to steps of an automaton from a given state (if such steps are defined in the transition relation of the automaton). Once we have encoded (boolean) policies of targets and a notion of transparent enforcement as a (boolean) policy over monitored targets, we can use the translation described previously to derive a cost

policy and talk about cost transparent enforcement. This process has some technical nuances but it is straightforward to implement and we will not pursue it more in this thesis.

4.5 Cost Comparison

Given a system S, a function $\mathbb{F}$, a scheduler σ and a monitor M, $\mathsf{pcost}^{\mathbb{F}}_{\sigma}(M)$ and $\mathsf{pcost}^{\mathbb{F}}_{\sigma}(M \times S)$ are values in $[-\infty, \infty]$, and as such provide a way to compare monitors.

To meaningfully compare monitors, we need to fix the variables on which the cost of a monitor depends, i.e., functions $\mathbb{F}$ and cost, and the scheduler σ. Difficulties arise when trying to fix a scheduler for two different monitors (and thus monitored systems), even if they are defined over the same signature. States of the monitors, and thus their executions, will be syntactically different and we cannot directly define a single scheduler for both. Moreover, since schedulers assign probabilities to specific PIOA and their transitions, one scheduler cannot be defined for two different monitors.

To overcome this difficulty we rely on the abstract schedulers introduced in Section 4.2.3. Namely, to compare two monitored systems we use a single abstract scheduler which we then refine into schedulers for each monitored system.[9]

Abstract schedulers allow us to fairly compare two monitors, but additional constraints are needed to eliminate impractical corner cases. To this end we introduce *fair abstract schedulers*.

Definition 24. *An abstract scheduler τ over the signature of a class of monitored targets $\mathcal{M} \times \mathcal{S}$ is* fair *(w.r.t. comparing monitors) if and only if (1) the monitors get a chance to respond to targets' actions infinitely often (i.e., the monitors are not starved), and (2) for every trace t of a monitored target, every extension t' of t by a monitor's actions, i.e., $t' = t; a$ with $a \in$ External(M), is assigned the same probability by τ.*

Constraint (1) ensures that a fair abstract scheduler will not starve the monitor, i.e., the moni-

[9] An abstract scheduler τ also provides a meaningful way to compare monitors with different signatures: calculate the union S of the signatures of the two monitors and (1) use a τ with signature S, and (2) extend each monitor's signature to S. This is useful when comparing monitors of different capabilities, e.g., a truncation and an insertion monitor [145], where the insertion monitor might exhibit additional actions, e.g., logging.

tor will always eventually be given a chance to enforce the policy. Constraint (2) ensures that the abstract scheduler is not biased towards a specific monitoring strategy. For example, an unfair scheduler could assign zero probability to arbitrary monitoring actions (e.g., the scheduler stops insertion monitors [145]) and non-zero probability to monitors that output valid target actions verbatim (i.e., the scheduler allows suppression monitors [145]). Such a scheduler would be unlikely to be helpful in performing a realistic comparison of the costs of enforcement of an insertion and a suppression monitor. There might be scenarios where such schedulers are appropriate[10], but in this thesis we pursue only the equiprobable scenario.

Definition 25. *Given a system S, a function $\mathbb{F}$, a cost function* cost*, two monitors M_1 and M_2 with Sig$(M_1) =$ Sig(M_2), an abstract scheduler τ over Sig$(M_1 \times S)$, two schedulers σ_1 (for $M_1 \times S$) and σ_2 (for $M_2 \times S$) derivable from τ, and a well-order $\trianglelefteq$ we say that M_2 is* less costly *than a monitor M_1 and write $M_2 \trianglelefteq M_1$, if and only if $\mathsf{pcost}^{\mathbb{F}}_{\sigma_2}(M_2 \times S) \trianglelefteq \mathsf{pcost}^{\mathbb{F}}_{\sigma_1}(M_1 \times S)$.*

Definition 26. *A monitor M is* cost optimal *for a system S and a well order $\trianglelefteq$ if and only if for all monitors M' with Sig$(M) =$ Sig(M'), $M \trianglelefteq M'$.*

The next proposition states that for any system S a cost optimal monitor exists.

Proposition 4.5.1. *Given a system S there is a cost optimal monitor M (for some well-order $\trianglelefteq$).*

Proof. There are countably many monitors (since there are countably many I/O automata). Thus, there are countably many monitored systems (for the given system S). Moreover, it is known (from set theory) that every countable set can be well-ordered by some well-order $\trianglelefteq$. By definition of well-orders every subset of monitored systems will have a least element under $\trianglelefteq$. Thus, the least element $M \times S$ of the set of all possible monitors for S exists, and M is cost optimal for S.

□

[10]This is a similar situation with having various definitions for fairness [63].

Note that although for every system S we can find a cost optimal monitor M, M is cost optimal for a specific well-order $\trianglelefteq$ which might not be the standard well-order $\leq$. For example, if the expected costs of monitored systems take values from the natural numbers then there exists an optimal monitor under $\leq$. On the other hand, if the expected costs of monitored systems range over the integers, or real numbers, then there might not exist a cost optimal monitor under $\leq$ (although a cost optimal monitor might exist for a different well order $\trianglelefteq$[11]).

Proposition 4.5.1 guarantees that a cost-optimal monitor exists (for given system S, function $\mathbb{F}$, cost function cost, and abstract scheduler τ). However, we might not be able to find or construct such a cost-optimal monitor. One reason for this inability can be, for example, that $\mathbb{F}$ is not be differentiable. However, for many practical purposes functions $\mathbb{F}$ for which we can find optimal monitors will be used. One such class of functions is *monotone* functions. Monotone functions map arguments to values in such a way that they preserve some order, i.e., if two arguments are ordered under some ordering, then so are the corresponding values when the function is applied on the arguments. For instance, the function that maps traces of a system to CPU cycles or power consumption is monotone: the more time the system runs, i.e., the longer the trace of the system, the more CPU cycles and power are consumed. Another example is the function that maps valid executions of a monitor that mediates communication between a client and a server to profits for the server: the longer a valid execution, the more profit the server will make. This function is relevant in scenarios where some Service Level Agreement exists between a client and a server, and we want to maximize the well-behaved interaction amongst them (cf. transparency–Section 4.4). Next we formally define monotone functions $\mathbb{F}$ (remember that we assume that $\mathbb{F}$'s arguments are sets of pairs $\langle t, c\rangle$, where the first component t is a trace and the second component c is (typically) the expected cost of t).

[11]For instance, since there are countably many monitors, one can use a bijection that maps monitors to natural numbers and the standard well-ordering of the natural numbers.

Definition 27. *Given two sets X, Y of pairs of traces and real numbers, i.e., $X, Y \in 2^{(\Sigma)^\infty \times \mathbb{R}}$, we write $X \sqsubseteq Y$ if and only $\forall \langle t_1, c_1 \rangle \in X : \langle t_1, c_1 \rangle \in Y$, and $\exists \langle t_2, c_2 \rangle \in Y : \forall \langle t_3, c_3 \rangle \in X : \langle t_3, c_3 \rangle \sqsubseteq \langle t_2, c_2 \rangle$, where $\langle t_i, c_i \rangle \sqsubseteq \langle t_j, c_j \rangle$ if and only if $t_i \preceq t_j$ and $c_i \trianglelefteq c_j$.*

We say that a function $\mathbb{F} : 2^{(\Sigma)^\infty \times \mathbb{R}} \to \mathbb{R}$ is monotone *if and only if it is monotone under the ordering $\sqsubseteq$, i.e., if $X \sqsubseteq Y$ then $\mathbb{F}(X) \trianglelefteq \mathbb{F}(Y)$.*

The next theorem formalizes the intuition that when dealing with monotone functions we can exploit knowledge about the scheduler and the cost function to build cost optimal monitors.

Theorem 4.5.1. *Given:*

1. *a finite-state system S,*
2. *a cost assignment* map *of each action of S to a real value such that for every input action i of $\mathsf{map}(i) = 0$,*
3. *a cost function* cost *that is defined recursively based on* map *such that the* cost *of a trace t is the sum of the values of the actions that appear on t,*
4. *a weakly fair*[12] *abstract scheduler τ, and*
5. *a function $\mathbb{F}$ that is monotone and continuous (i.e., it preserves limits),*

we can construct a cost optimal monitor for the standard ordering $\leq$ of real numbers.

Proof. First we define the signature of M to contain (1) as input actions the output actions of S, and (2) as output actions the input actions of S. Since S has finitely many states we can detect (1) whether it contains any cycles, and (2) the cost of each cycle. Assuming that S contains cycles, we construct a monitor that complements every positive-value cycle k of S, i.e., the transition relation of M produces all possible cycles k that have a positive sum of costs. Since M and S contain complementary input and output actions, the composition $M \times S$ will exhibit all those cycles.

[12] As described in Section 4.4, weakly fair schedulers are the schedulers that do not assign zero probabilities to steps of an automaton from a given state (if such steps are defined in the transition relation of the automaton).

Since the abstract scheduler is weakly fair, it is easy to see that this will also hold for the refined scheduler σ for $(M \times S)$ (by definition of refinement). This guarantees that $(M \times S)$ will contain infinite traces. In fact, $M \times S$ will contain all possible infinite traces that S can produce and whose costs will diverge to $+\infty$.

Finally, every other monitor M' will either (1) contain less cycles with positive cost than M, and thus will diverge (if they do) slower, or (2) contain negative cycles in addition to the positive ones which will have the same effect as the previous point, or (3) may contain additional self loops of output actions since they are under the monitor's control: by the second and third constraint of the theorem the only actions that have a cost are the outputs of the system, i.e., the outputs of any M' will have a zero impact on the cost of a trace. Thus, in conclusion, by the monotonicity of $\mathbb{F}$, M will be optimal as compared to every other M'.

□

Thm. 4.5.1 provides a generic description of the conditions sufficient for constructing a cost-optimal monitor. In the constructive proof of Thm. 4.5.1 we try to find a monitor M such that the (finite-state) monitored system $(M \times S)$ contains as many cycles as possible. The cycles together with the constraint that the scheduler is weakly fair guarantee the existence of infinite traces. The monotonicity of $\mathbb{F}$ and the recursive additive definition of the cost function cost guarantee that the monitor that contains the most highest-value cycles will be cost optimal. One might argue that Thm. 4.5.1 contains many (severe) restrictions. However, this is something unavoidable. Cost-enforcement and cost-optimality depend on many functions that need to be optimized in synchrony: there is no generic algorithm that will work for every (arbitrary) choice of functions. On the other hand, a positive interpretation of Thm. 4.5.1, based on the previous discussion about monotone functions, is that in (many) practical applications we have to deal with simple enough functions, and thus we can construct (verifiable) optimal monitors.

Running example. Typically, when a monitor modifies the behavior of the system some cost is incurred (e.g., the usability of the system decreases, computational resources are consumed). For instance, in the running example, one way monitors can modify the behavior of the system is by denying an access to a client. If we assume that each deny action incurs a cost of 1, then we can define a function cost_D that associates with each trace the cost n, where n is the number of denies that appear in the trace.

Moreover, let us assume that (1) $\mathbb{F}$ is Sup, and (2) the abstract scheduler τ follows the pattern $\Big([C_1, \ldots, C_n];\ M^*;\ S;\ M^*\Big)^\infty$ as described in Section 4.2.4. Assuming we have two clients C_1 and C_2, our monitored system is $\Pi = C_1 \times C_2 \times M \times S$. If M is M_{DENY}, then we refine τ to the scheduler $\sigma_{M_{DENY}}$; dually, the scheduler for M_{PROB} will be $\sigma_{M_{PROB}}$. The probabilistic cost of the monitored system with M_{DENY} is $\mathsf{sup}_{t \in traces(\Pi_{M_{DENY}})}(\mathsf{pcost}_{\sigma_{M_{DENY}}}(C_t))$, and similarly for M_{PROB}.

We first observe that with such a cost function, the maximal (i.e., best) reachable cost is 0, meaning that no deny action is returned. It follows that the cost-optimal monitor never denies any action, and, clearly, this monitor does not generally respect the requirement that at most one client at a time should have access to a particular file.

Second, we observe that if we assume that C_1 and C_2 ask for a file after a denied request with probability p_1 and p_2 respectively, with $p_1 < p_2$, then C_1 is less likely to ask again for a file which has been denied. In this case, it is better to deny an access to C_1 rather than to C_2, in order to limit the number of deny actions. Hence, with such a system, we have $M_{PROB} \leq M_{DENY}$.

Finally, observe that the last result is sound only under the assumption that schedulers $\sigma_{M_{DENY}}$ and $\sigma_{M_{PROB}}$ are compatible with τ. If that was not the case, then $\sigma_{M_{DENY}}$ could starve C_2 (or $\sigma_{M_{PROB}}$ could starve C_1). This would give M_{DENY} an unfair advantage over M_{PROB}, and we would have as a result that $M_{DENY} \leq M_{PROB}$. Such unfair comparisons are ruled out by requiring schedulers to be compatible.

$N-$step optimality. In practice, the risk (cost-benefit) analysis is a dynamic process: probabilities for attacks to happen (arrival of inputs) are re-evaluated based on the current state of the system (e.g., sudden publicity, or increase in the value of the company), costs might have changed (e.g., the state of the art cryptographic protocols might require more computational resources), and such changes affect the optimality of an enforcement system. Thm. 4.5.1 gives us a generic description of what conditions are sufficient for constructing a cost-optimal monitor.

However, the assumption of having schedulers and cost functions that have valid information about the future might be too ambitious. Next, we present a special case of the theorem where we only have information about $n-$steps ahead in the future, where $n \in \mathbb{N}$. Before we state the modified theorem, we need to first adjust our basic definitions so they reason about the partial knowledge that we may have.

Given a scheduler σ intuitively we construct a $n-$step scheduler σ_n by restricting the knowledge that σ may have about the probabilities of actions to happen after an execution that has length at most n. More formally:

Definition 28. *Given a scheduler σ, an $n-$step scheduler σ_n is a function σ_n : execs$^*(P) \to$ $\mathsf{SubDisc}($trans$(P))$ defined as:*

- $\sigma_n(e) = \sigma(e)$, *if* $|$*traces*$(e)| \leq n$,
- $\mathsf{supp}(\sigma_n(e)) = \emptyset$, *otherwise.*

Note that a $n-$step scheduler σ_n is still a scheduler but with less information than the original scheduler σ. Thus, the definition of a measure of cone and trace remain the same as in Section 4.2.2.

The definition of an $n-$step abstract scheduler follows that of a $n-$step scheduler:

Definition 29. *Given an abstract scheduler τ over a signature S, an $n-$step abstract scheduler τ_n is a function τ_n : $($External$(S))^* \to \mathsf{SubDisc}($External$(S))$ defined as:*

- $\tau_n(t) = \tau(t)$, *if* $|t| \leq n$,

- $\mathsf{supp}(\tau_n(t)) = \emptyset$, *otherwise.*

Given a cost function cost, a $n-$step cost function $\mathsf{cost_n}$assigns a real number to every trace over a signature S that has length at most n, and 0 otherwise. More formally:

Definition 30. *Given a cost function* cost *over a signature* S*, a* $n-$*step cost function* $\mathsf{cost_n}$ *over* S *is defined as*

- $\mathsf{cost_n}(t) = \mathsf{cost}(t)$*, if* $|t| \leq n$*,*
- $\mathsf{cost_n}(t) = 0$*, otherwise.*

We calculate the $n-$step probabilistic cost of a trace as described in Section 4.3, i.e., by multiplying the probability of the trace with its cost. Note that when a trace has length larger than n then its $n-$step probabilistic cost will be zero. Although we could have defined the (expected) cost of such a trace to be undefined (e.g., $\perp$) rather than a concrete value, we set up our definitions in such a way that this value does not cause any conflicts when the expected costs of traces with length less than n is also zero. In the alternative case we would have to adjust many definitions and calculations, e.g., those of measures, with significant notational overhead.

Definition 31. *Given a function* $\mathbb{F}$*, we define the* $n-$*step restriction of* $\mathbb{F}$ *to be the restriction of* $\mathbb{F}$ *to the traces that have length at most* n*, and write* $\mathbb{F} \restriction_n$.

Definition 32. *Given a system* S*, a function* $\mathbb{F}$*, a* $n-$*step cost function* $\mathsf{cost_n}$*, two monitors* M_1 *and* M_2 *with* $\mathit{Sig}(M_1) = \mathit{Sig}(M_2)$*, an* $n-$*step abstract scheduler* τ_n *over* $\mathit{Sig}(M_1 \times S)$*, two* $n-$*step schedulers* σ_1 *(for* $M_1 \times S$*) and* σ_2 *(for* $M_2 \times S$*) derivable from* τ_n*, and a well-order* $\trianglelefteq$ *we say that* M_2 *is* $n-$step less costly *than a monitor* M_1 *and write* $M_2 \trianglelefteq_n M_1$*, if and only if* $\mathsf{pcost}^{\mathbb{F}\restriction_n}_{\sigma_2}(M_2 \times S) \trianglelefteq \mathsf{pcost}^{\mathbb{F}\restriction_n}_{\sigma_1}(M_1 \times S)$.

Definition 33. *A monitor* M *is* $n-$step cost optimal *for a system* S *and a well order* $\trianglelefteq$ *if and only if for all monitors* M' *with* $\mathit{Sig}(M) = \mathit{Sig}(M')$*,* $M \trianglelefteq_n M'$.

Now we are ready to state the equivalent of Thm. 4.5.1 for the case where our information

and knowledge is limited for just $n-$steps:

Theorem 4.5.2. *Given a number $n \in \mathbb{N}$ and:*

1. *a finite-state system S,*
2. *a $n-step$ cost function $\mathsf{cost_n}$,*
3. *a $n-$step abstract scheduler τ_n, and*
4. *a function $\mathbb{F}$,*

we can construct a $n-$step cost optimal monitor for the standard ordering $\leq$ of real numbers.

Proof. One can (inefficiently) enumerate all possible values for traces that the system can exhibit for $n-$steps and find the actions that the monitor needs to exhibit in order to be cost optimal. Since the traces that the monitors needs to exhibit to be $n-$step optimal are finite (both in length and cardinality), one can easily build a monitor that exhibits those (finitely many) traces by interleaving fresh states amongst each of those traces.

□

Note that a lot of the constraints of Thm. 4.5.1 have been relaxed. The lack of knowledge arbitrarily far in the future can actually help us to constrain and enumerate the variables that affect cost optimality and thus build an optimal monitor.

Although $n-$step cost-optimal monitors are practically relevant, they are limited in their capability to optimize the cost of the monitored system on a longer interval than they are supposed to. For example one cannot iterate an $n-$step cost-optimal monitor and expect to have the same results as using a (infinite-horizon) cost-optimal monitor.

4.6 Related Work

In Section 2.7 we discussed several previous formal models of run-time monitors that build on the original model of run-time monitors, i.e., security automata [27]. Although these models are richer and orthogonal revisions to security automata and related computational and operational extensions, they maintain the same view of (enforceable) security policies: binary predicates over sets of executions. In this chapter we take a richer view of security policies than binary predicates, assigning costs and probabilities to traces and define cost-security policies and cost-enforcement, which as shown in Section 4.4 is a strict extension of binary security policies and enforcement.

Ray and Ligatti introduced the notion of gray security policies [160]. Gray security policies are similar to our cost security policies: they are extensions of security policies from predicates on individual executions, to functions that assign real numbers to executions (and sets of executions). Thus, similarly to our work, gray security policies allow for the quantification of security policies. The main difference between our work and the work on gray security policies is the focus of the meta-theoretical analysis. The work on gray security policies focuses on casting previous well-known results in security policies (e.g., the decomposition of properties to safety properties and liveness properties) in the context of the new quantitative notion of security policies. In contrast, our work focuses on comparing the expected cost of different monitoring designs. Thus, besides using measure theory as the basis of our framework (which is necessary for assigning probabilities to uncountable spaces [155]), we introduce concepts, such as abstract schedulers (Section 4.2.3), that allow us to compare monitoring designs in a fair manner.

Drábik et al. introduced the notion of calculating the cost of an enforcement mechanism based on a relatively simple enforcement model which does not include input/output actions or a detailed calculation of the execution probabilities [157]. To some extent, the notion of cost security policy defines a threshold characterising the maximal/minimal cost reachable, while taking the

probability of reaching this threshold into account. Such a notion of threshold is also used by Cheng et al. where accesses are associated with a level of risk, and decisions are made according to some predefined risk thresholds, without detailing how such policies can be enforced at runtime [161]. In the context of runtime enforcement, Bielova and Massacci proposed to apply a distance metrics to capture the similarity between traces, and we could consider the cost required to obtain one trace from another as a distance metrics [162]. An important aspect of this work is to consider that a property might not be locally respected, i.e., for a particular execution, as long as the property holds globally. This possibility is also considered by Drabik et al. which quantifies the tradeoff correctness/transparency for non-safety boolean properties [163]. Caravagna et al. introduced the notion of lazy controllers, which use a probabilistic modelling of the system in order to minimize the number of times when a system must be controlled, without considering input/output interactions between the target and the environment as we do [164]. These lines of work are in the scope of going towards a notion of quantitative enforcement where enforcement mechanisms are quantitatively evaluated and decisions made using quantitative analysis, rather than the binary adherence to a policy, in the context of process algebra [165].

Finally, the idea of optimal monitor is considered by Easwaran et al. in the context of software monitoring, where correcting actions are associated with rewards and penalties within a Directed Acyclic Graph, using dynamic programming to find the optimal solution [166]. Similarly, Markov Decision Process (MDPs) can be used to model access control systems [167], and the optimal policy can be derived by solving the corresponding optimisation problem. A potential lead for future work would therefore be to focus on Probabilistic I/O Automata (PIOA) that correspond to MDPs (since PIOA subsume MDPs [168]), in order to reuse the computation of optimal policy of the latter within the expressive framework of the former.

4.7 Conclusion

In this chapter we introduced a formal framework based on probabilistic I/O automata to model and reason about interactive run-time monitors. In our framework we can formally reason about probabilistic knowledge monitors have about their environment and combine it with cost information to minimize the overall cost of the monitored system. We have used this framework to (1) calculate *expected costs of monitors* (Section 4.3), (2) define *cost security policies* and *cost enforcement*, richer notions of traditional definitions of security policies and enforcement [27] (Section 4.4), and (3) order monitors according to their expected cost and show how to build an optimal one (Section 4.5).

Chapter 5

Conclusion

Enforcing security policies and detecting attacks in practice requires the collaborative use of a variety of security mechanisms such as firewalls, network-based intrusion detection systems, host-based intrusion detection systems, intrusion prevention systems, netflow data analysis, spam filters, antivirus software, and audit logs and tools [50]. Formal frameworks for modeling and analyzing such security mechanisms are of great importance for both theoretical and practical reasons. From a theoretical perspective, there are two main advantages. First, formal frameworks for security mechanisms can be used to analyze the intrinsic limits of different types of security mechanisms by characterizing the class of security policies that these mechanisms can enforce. Such a characterization can lead to a taxonomy of security policies based on mechanisms' formal semantics [27]. Second, formal frameworks can be used to provide formal semantics to policy specification languages that can be used to define our security goals and describe what are the behaviors of the system, or the network, that are, and are not, allowed. From a practical perspective there are two main advantages. First, the knowledge of the classes of security policies that different mechanisms can enforce can help security engineers to make sound and efficient design decisions. For example a security engineer can choose the appropriate type of mechanisms to

enforce a given security policy, or reevaluate the given security goals if a policy is not enforceable with the security mechanisms available to the engineer. Second, formal frameworks can be used in the verification of systems' designs, and the formal semantics of security policy specification languages can be used in the building of certified compilers which can generate efficient mechanisms that provably enforce the given security policies.

Previous work introduced several formal frameworks for security mechanisms and characterized the policies enforceable by mechanisms modeled in these frameworks [27, 28, 29, 30, 120]. In this thesis we took some additional steps towards the characterization of security policies that practical security mechanisms can enforce. We summarize our key contributions in Section 5.1. However, the space of security policies and security mechanisms is very large and there are practical scenarios that cannot be modeled in existing frameworks, including the one presented in this thesis. In Section 5.2 we discuss some potential future steps that can be taken towards expanding the currently available characterization of clases of enforceable security policies.

5.1 Summary of Contributions

As outlined in Section 1, in this thesis we have made the following principal contributions.

In Chapter 2 we introduced a (basic) formal framework based on Input/Output automata that can be used to model target applications, various types of monitors (e.g., partially mediating ones), and the environment that the monitored targets operate. We also illustrated how different monitors can be modeled in this framework. In our framework we extended previous definitions of security policies to support more fine-grained reasoning of policy enforcement (e.g., enforcement that requires monitors to enforce the security policy by modifying the intercepted traffic) and we identified a set of upper bounds of policies that are enforceable by monitors that can be modeled in our framework, namely input forgiveness, safety, and quiescent forgiveness. Fi-

nally, we demonstrated how to use our framework to derive new meta-theoretical results, such as the comparison of enforcement capabilities of monitors with different monitoring interfaces and the characterization of the class of security policies that monitors can enforce regardless of their monitoring interface.

In Chapter 3 we extended the previous basic framework by introducing a framework that allows modeling distributed systems with arbitrary architectures and distributed monitors. We used our framework to characterize the security policies that are enforceable in asynchronous and synchronous distributed systems. To characterize these policies we analyzed which centralized monitors can be *simulated* by distributed monitors. Specifically, we provided a blueprint for decomposing centrally specified monitors to monitors that are distributed over a message-passing network and enforce the same policies as the original centralized monitors. To illustrate how the blueprint can be used to design such decomposition algorithms we described two different algorithms that can be used to instantiate the blueprint. These two algorithms correspond to two widely implemented types of distributed enforcement: *centralized enforcement* (where a single monitored node in the distributed system makes all the enforcement decisions) and *decentralized enforcement* (where multiple nodes in the distributed system are responsible for making enforcement decisions). We then provided a characterization of the security policies that are enforceable by monitors that operate in a hierarchical manner. Finally, we identified the constraints under which monitors with simple capabilities (e.g., truncation automata) can be used in a cooperative manner to simulate monitors with seemingly more capabilities (e.g., suppression automata).

Finally, in Chapter 4 we introduced a formal framework based on Probabilistic Input/Output automata that enabled us to formally reason about the expected cost of different monitoring designs. One of our key contributions was the concept of *abstract schedulers* which allows fair comparison of systems, where a policy is enforced on a target by different monitors. Another contribution was the definition of cost security policies and cost enforcement, richer notions of

(boolean) security policies and enforcement. Cost security policies assign a cost to each trace allowing a richer classification of traces than just good or bad. We also showed how to encode boolean security policies as cost security policies. Finally, we showed how to use our framework to compare monitor's implementations and we identified the sufficient conditions for constructing cost-optimal monitors.

5.2 Future Work

This section describes some possibilities for extending our work in the future.

Models of systems. Previous formal frameworks focused on monitors and systems that could be expressed in centralized models [27, 28, 29, 30]. Centralized models (e.g., Turing machines) are not expressive enough to model and reason about key characteristics of distributed systems such as interaction, concurrency and parallelism. In this thesis we extended previous work by introducing a framework that allows expressing distributed systems and monitors. However, there are systems whose properties of interest cannot be expressed in our framework because distributed models are not sufficient to model these systems' key characteristics. One such example is time: real-time systems and monitors, e.g., security monitors for vehicles, need more expressive frameworks than ours, e.g., frameworks based on Timed I/O Automata [33]. Another example of a more expressive model that could serve as the basis for a formal framework of enforcement mechanisms is Dynamic I/O Automata (DIOA) [146]. As we mentioned in Section 3.7, DIOA could be used to model monitors that have the ability to generate new monitors (i.e., automata) at run-time. This would be useful to model distributed enforcement mechanisms that can dynamically generate new monitors in order to more efficiently handle large amounts of traffic.

Finally, as we mentioned in Section 3.2, one aspect of distributed systems that we did not investigate in this thesis is failures. There are different types of failures both for nodes (e.g., stopping and byzantine failures) and communication channels (e.g., message loss, message duplication, and message reordering). Similarly to applications making different assumptions about the underlying timing model (i.e., synchronous or asynchronous – see Section 3.5) and thus corresponding to different class of enforceable policies, applications may also make different assumptions about failures and thus it is important to have results for these different types of models as well.

Models of enforcement mechanisms. Orthogonal to the modeling of systems is the modeling of security mechanisms with different monitoring and enforcement *abilities*. Although we demonstrated how we can formalize some common security mechanisms (e.g., resource usage monitors and intrusion detection systems) in our frameworks (see Chapter 2), there are security mechanisms that cannot be (directly) expressed in our framework. One such example is inlined reference monitors [82, 103, 104]. Inlined reference monitors are run-time monitors that are inlined in the target application and have direct access to the state of the target application. Although one could model such mechanisms in the shared memory model (see Section 3.4.6.1) as two processes having access to a shared variable, where the shared variable represents the memory of the monitored application, there are details that one needs to be careful, and formally reason, about. For instance, if the part of the shared memory that the monitor keeps its state at is accessible by the target application, then the target application could corrupt this state and circumvent the monitor. Such a model would need (new) composition operators to define not only how to compose a monitor with a target application (modeling the inlining process), but also how to compose multiple monitored applications together. Moreover, it would have to be proven that these composition operators preserve the *non-circumvention* property that a sound

inlining technique should have [103]. Work that is related to inlining models includes (1) *monitoring algorithms* that have been used for, e.g., taking consistent global snapshots and detecting stable properties in distributed systems [1], and (2) the *superposition* operation in the Unity programming language [169].

Security policies' representations. In Chapter 2 we formally defined security policies as sets of schedule modules, i.e., sets of sets of traces [27, 159]. In Chapters 3 and 4 we focused on a specific class of policies, namely *propreties* [27, 30, 159]. Previous work has identified that *properties* are not expressive enough to model some common security policies such as information flow and real time availability [27, 159]. Thus, in addition to more expressive frameworks for modeling systems and monitors, future work can focus on characterizing the enforcement of such additional security policies by, e.g., distributed monitors.

Security policy specification languages. In Section 3.3.1 we mentioned that previous work has introduced a large number of specification languages that can be used to express (centralized) security policies using preconditions and postconditions or, equivalently, centralized automata (i.e., global monitors) [49, 95, 100, 103, 104, 105, 106, 107, 108, 109].

In this thesis, we used I/O automata (and PIOA) to model targets, monitors, and security policies. There is a straightforward translation from the semantic framework of I/O automata to the description of distributed algorithms using preconditions and effects (i.e, postconditions). In fact, it is typical to use precondition and effects to describe I/O automata without distinguishing between the two levels of description (i.e, semantic and syntactic) [1, 112]. Due to this fact, we mentioned that our framework can serve as the basis for an attack specification language with formally specified semantics. Although specification languages for I/O automata have already been introduced in the literature [112, 113, 114], an interesting avenue for future research would

be to formalize such an attack specification language in a verification system (e.g., COQ [170] or Isabelle/HOL [171]) so that properties of this language can be formally proven, and certified compilers that generate efficient mechanisms that provably enforce a given policy can be built.

Bibliography

[1] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996. (document), 2.1, 2.2, 2.3.1, 2.5.1, 2.5.1, 2, 3.2, 3.2, 3.2, 3.3.1, 3.4.1.1, 3.4.1.3, 2, 3.4.2, 3.4.3.1, 3.4.3.2, 3.4.4, 3.4.6, 3.4.6.1, 3.16, 3.17, 3.4.6.1, 3.4.6.1, 3.4.6.1, 3.4.6.1, 3.4.7, 3.4.7.1, 3.4.7.1, 3.4.7.1, 17, 3.4.6, 3.4.7.1, 3.4.7.3, 3.4.7.3, 18, 20, 3.5, 3.5.1, 3.5.1, 3.6, 3.8, 5.2, 5.2

[2] Seth Fiegerman. Yahoo says 500 million accounts stolen. CNNMoney, September 2016. http://money.cnn.com/2016/09/22/technology/yahoo-data-breach/. 1

[3] Jon M. Chang. Passwords and email addresses leaked in Kickstarter hack attack. ABC News, February 2014. http://abcnews.go.com/Technology/passwords-email-addresses-leakedkickstarter-hack/story?id=22553952. 1

[4] Joseph Bonneau. The Gawker hack: How a million passwords were lost. Light Blue Touchpaper Blog, December 2010. http://www.lightbluetouchpaper.org/2010/12/15/the-gawkerhack-how-a-million-passwords-were-lost/. 1

[5] Jose Pagliery. Cyber thieves siphon tax forms from ADP payroll data. CNNMoney, May 2016. http://money.cnn.com/2016/05/03/technology/adp-w2-forms-stolen/. 1

[6] Vodafone customers' bank details 'accessed in hack' company says. The Gurdian, October 2015. https://www.theguardian.com/business/2015/oct/31/vodafone-

customers-bank-details-accessed-in-hack-company-says. 1

[7] Gregory Wallace. Target credit card hack: What you need to know. CNNMoney, December 2013. http://money.cnn.com/2013/12/22/news/companies/target-credit-card-hack/. 1

[8] Kevin McCoy. Cyber hack got access to over 700000 IRS accounts. USA Today, February 2016. http://www.usatoday.com/story/money/2016/02/26/cyber-hack-gained-access-more-than-700000-irs-accounts/80992822/. 1

[9] Elizabeth Snell. Centene Healthcare Data Breach Affects 950K Patients. HealthITSecurity, January 2016. http://healthitsecurity.com/news/centene-healthcare-data-breach-affects-950k-patients. 1

[10] Tim Greene. Hackers compromise 1.8 million medical records from healthcare provider Premera. Networked World, March 2015. http://www.networkworld.com/article/2898497/security0/hackers-compromise-18-million-medical-records-from-healthcare-provider-premera.html. 1

[11] Frank Gluck. 21st Century Oncology data breach prompts multiple lawsuits. news-press.com, July 2016. http://www.news-press.com/story/news/2016/07/22/21st-century-oncology-data-breach-prompts-multiple-lawsuits/87386068/. 1

[12] Sam Frizell. What Leaked Emails Reveal About Hillary Clinton's Campaign. Time, October 2016. http://time.com/4523749/hillary-clinton-wikileaks-leaked-emails-john-podesta/. 1

[13] Polly Mosendz. Wikileaks Continues Publicizing Emails Of CIA Chief John Brennan. Newsweek, October 2015. http://www.newsweek.com/wikileaks-continues-publicizing-emails-cia-chief-john-brennan-387316. 1

[14] Sam Biddle. More Embarrassing Emails: The Sony Hack B-Sides. Gawker,

April 2015. http://gawker.com/more-embarrassing-emails-the-sony-hack-b-sides-1698557943. 1

[15] Andy Greenberg. Hackers Remotely Kill a Jeep on the Highway — With Me in It. Wired, July 2015. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/. 1

[16] Lucas Mearian. Update: Chrysler recalls 1.4M vehicles after Jeep hack. Computerworld, July 2015. http://www.computerworld.com/article/2952186/mobile-security/chrysler-recalls-14m-vehicles-after-jeep-hack.html. 1

[17] Joe Davidson. Next cyberattack front could be your car. Washington Post, May 2016. https://www.washingtonpost.com/news/powerpost/wp/2016/05/18/next-cyberattack-front-could-be-your-car/. 1

[18] Christoph Steitz and Eric Auchard. German nuclear plant infected with computer viruses operator says. Reuters, April 2016. http://www.reuters.com/article/us-nuclearpower-cyber-germany-idUSKCN0XN2OS. 1

[19] Eduard Kovacs. Attackers Alter Water Treatment Systems in Utility Hack: Report. Securityweek, March 2016. http://www.securityweek.com/attackers-alter-water-treatment-systems-utility-hack-report. 1

[20] John Leyden. Water treatment plant hacked chemical mix changed for tap supplies. The Register, March 2016. http://www.theregister.co.uk/2016/03/24/water_utility_hacked/. 1

[21] Michael B Kelley. The Stuxnet Attack On Iran's Nuclear Plant Was 'Far More Dangerous' Than Previously Thought. Business Insider, November 2013. http://www.businessinsider.com/stuxnet-was-far-more-dangerous-than-previous-thought-2013-11. 1

[22] Kim Willsher. French fighter planes grounded by computer virus. The Telegraph, February

2009. http://www.telegraph.co.uk/news/worldnews/europe/france/4547649/French-fighter-planes-grounded-by-computer-virus.html. 1

[23] Robert A. Guth, Daniel Machalaba. Computer Viruses Disrupt Railroad, and Air Traffic. The Wall Street Journal, August 2003. http://www.wsj.com/articles/SB106140797740336000. 1

[24] Mary Kay Mallonee. Hackers publish contact info of 20000 FBI employees. CNN, February 2016. http://www.cnn.com/2016/02/08/politics/hackers-fbi-employee-info/. 1

[25] Leith Huffadine. Australian government told to 'harden up' as it's revealed Chinese hackers target the defence force and other departments 'on a daily basis'. Daily Mail Australia, August 2016. http://www.dailymail.co.uk/news/article-3762761/Austrade-Defence-Science-Technology-Group-hacked-China-cyber-attacks-sponsored-Beijing.html. 1

[26] Matt Bishop. *Computer Security: Art and Science*. Addison-Wesley Professional, 2002. 1, 2.1, 3.3.2

[27] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, February 2000. 1, 1, 1, 1, 2, 2.1, 2.3.1, 2.3.2, 2.3.3, 2.3.4, 1, 9, 2.4.1, 2.6, 2.7, 2.8, 3.3.2, 3, 3.4.2, 1, 3.7, 3.8, 3.9, 3, 4.4, 4.4, 4.4, 4.6, 4.7, 5, 5.2, 5.2

[28] M. Viswanathan and M. Kim. Foundations for the run-time monitoring of reactive systems: Fundamentals of the MaC language. In *International Conference on Theoretical Aspects of Computing (ICTAC)*, volume 3407 of *LNCS*, pages 543–556, Guiyang, China, September 20-24 2004. 1, 5, 5.2

[29] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006. 1, 2.4.1, 2.5.2, 2.6, 2.7, 3.3.2, 3.4.2, 1, 3.4.6.3, 5, 5.2

[30] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3):19:1–19:41, January 2009. 1, 1, 1, 2, 2.1, 2.3.1, 2.3.4, 2.3.4, 2.3.5, 1, 2.4.1, 9, 2.4.1, 2.5.2, 2.5.3, 2.6, 2.6, 2.7, 2.7, 2.8, 3.3.2, 3, 3.4.2, 1, 3.9, 4.4, 4.4, 5, 5.2, 5.2

[31] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996. 1, 3.2

[32] Christos M. Papadimitriou. *Computational complexity*. Addison-Wesley, Reading, Massachusetts, 1994. 1

[33] Dilsun K. Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. *The Theory of Timed I/O Automata (Synthesis Lectures in Computer Science)*. Morgan & Claypool Publishers, 2006. 3, 5.2

[34] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *European Symposium on Research in Computer Security (ESORICS)*, volume 3679, pages 355–373, 2005. 1, 2, 2.3.2, 2.6

[35] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Air Force Electronic Systems Division, 1972. 1

[36] Yannis Mallios, Lujo Bauer, Dilsun Kaynar, and Jay Ligatti. Enforcing more with less: Formalizing target-aware run-time monitors. In *Proceedings of the 8th International Workshop on Security and Trust Management*, volume 7783 of *Lecture Notes in Computer Science*, pages 17–32, 2013. 2, 1.1, 4.1, 7, 4.4, 4.4

[37] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 2–16, 1999. 1, 2.1

[38] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable,

untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009. 1

[39] William Enck, Machigar Ongtang, Patrick Drew McDaniel, et al. Understanding android security. *IEEE security & privacy*, 7(1):50–57, 2009. 1

[40] G. Robert Malan, David Watson, Farnam Jahanian, and Paul Howell. Transport and application protocol scrubbing. In *Proceedings of INFOCOM 2000*, pages 1381–1390, 2000. 1, 4.1, 4.1

[41] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 257–272, Berkeley, CA, USA, 2003. USENIX Association. 1

[42] William R Cheswick, Steven M Bellovin, and Aviel D Rubin. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley Longman Publishing Co., Inc., 2003. 1

[43] Elizabeth D Zwicky, Simon Cooper, and D Brent Chapman. *Building internet firewalls*. O'Reilly Media, Inc., 2000. 1

[44] New Open Source Intrusion Detector Suricata Released. Slashdot, December 2009. `https://linux.slashdot.org/story/09/12/31/2143250/New-Open-Source-Intrusion-Detector-Suricata-Released`. 1

[45] Gene H Kim and Eugene H Spafford. The design and implementation of tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29. ACM, 1994. 1

[46] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the 13th USENIX Conference on System Administration*, LISA '99, pages 229–238, Berkeley, CA, USA, 1999. USENIX Association. 1

[47] Rick Hofstede, Pavel Čeleda, Brian Trammell, Idilio Drago, Ramin Sadre, Anna Sperotto,

and Aiko Pras. Flow monitoring explained: from packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials*, 16(4):2037–2064, 2014. 1

[48] Justin Mason. Filtering spam with spamassassin. In *HEANet Annual Conference*, page 103, 2002. 1

[49] Giovanni Vigna, William Robertson, Vishal Kher, and Richard A. Kemmerer. A stateful intrusion detection system for world-wide web servers. In *Proceedings of the 19th Annual Computer Security Applications Conference*, ACSAC '03, pages 34–43, Washington, DC, USA, 2003. IEEE Computer Society. 1, 3.3.1, 5.2

[50] Michael Collins. *Network Security Through Data Analysis*. O'Reilly Media, 2014. 1, 2.3.2, 3.3, 2, 3.3.2, 3.4.3.2, 5

[51] Joshua Haines, Dorene Kewley Ryder, Laura Tinnel, and Stephen Taylor. Validation of sensor alert correlators. *IEEE Security and Privacy*, 1(1):46–56, January 2003. 1, 3.3, 2

[52] Cristina Abad, Jed Taylor, Cigdem Sengul, William Yurcik, Yuanyuan Zhou, and Ken Rowe. Log correlation for intrusion detection: A proof of concept. In *Computer Security Applications Conference*, pages 255–264. IEEE, 2003. 1, 3.1, 3.8

[53] Yannis Mallios, Lujo Bauer, Dilsun Kirli Kaynar, Fabio Martinelli, and Charles Morisset. Probabilistic cost enforcement of security policies. In *Proceedings of the 9th International Workshop on Security and Trust Management*, volume 8203 of *Lecture Notes in Computer Science*, pages 144–159, 2013. 1.1

[54] Yannis Mallios, Lujo Bauer, Dilsun Kaynar, Fabio Martinelli, and Charles Morisset. Probabilistic cost enforcement of security policies. *Journal of Computer Security*, 23(6):759–787, 2015. 1.1

[55] David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report UCB/CSD-99-1056, EECS, University of California, Berkeley, 1999. 2.1

[56] Tal Garfinkel. Traps and pitfalls: Practical problems in system call interposition based

security tools. In *Network and Distributed Systems Security Symposium*, volume 3, pages 163–176, 2003. 2.1, 2.1

[57] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *Proceedings of the First USENIX Workshop on Offensive Technologies*, WOOT '07, pages 2:1–2:8, 2007. 2.1

[58] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996. 2.1

[59] Jay Ligatti and Srikar Reddy. A theory of runtime enforcement, with results. In *Computer Security - ESORICS 2010*, volume 6345 of *Lecture Notes in Computer Science*, pages 87–100, 2010. 2.1, 2.3.1, 1, 2.4.1, 2.6, 2.7, 4.4

[60] Richard Gay, Heiko Mantel, and Barbara Sprick. Service automata. In *Proceedings of the 8th international conference on Formal Aspects of Security and Trust*, pages 148–163, 2012. 2.1, 2.3.1, 2.7

[61] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings ACM Symposium on Principles of Distributed Computing*, pages 137–151, New York, NY, USA, 1987. ACM. 2.1, 2.2, 2.3.1

[62] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. 2.1

[63] M.Z. Kwiatkowska. Survey of fairness notions. *Information and Software Technology*, 31(7):371 – 386, 1989. 2.2, 2.5.3, 10

[64] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1982. 2.2, 2.7

[65] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *IEEE Computer Security Foundations Symposium*, pages 51–65, 2008. 2.3.3

[66] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European software engineering conference*, pages 109–120, 2001. 2.5.1

[67] Hugues Chabot, Raphael Khoury, and Nadia Tawbi. Extending the enforcement power of truncation monitors using static analysis. *Computers and Security*, 30(4):194 – 207, 2011. 2.5.2, 2.7

[68] Philip W. L. Fong. Access control by tracking shallow execution history. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pages 43–55, 2004. 2.7

[69] Nataliia Bielova and Fabio Massacci. Do you really mean what you actually enforced? *International Journal of Information Security*, 10(4):239–254, 2011. 2.7

[70] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31:560–599, June 1984. 2.7

[71] Egor Dolzhenko, Jay Ligatti, and Srikar Reddy. Modeling runtime enforcement with mandatory results automata. *International Journal of Information Security*, 14(1):47–60, February 2015. 2.7

[72] Fabio Martinelli and Ilaria Matteucci. Through modeling to synthesis of security automata. *Electron. Notes Theor. Comput. Sci.*, 179:31–46, July 2007. 2.7

[73] David Basin, Ernst-Ruediger Olderog, and Paul E. Sevinc. Specifying and analyzing security automata using CSP-OZ. In *Proceedings ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 70–81, 2007. 2.7

[74] David Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zălinescu. Enforceable security policies revisited. *ACM Trans. Inf. Syst. Secur.*, 16(1):3:1–3:26, June 2013. 2.7

[75] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983. 2.7

[76] Dimitra Giannakopoulou, Corina S Pasareanu, and Jamieson M Cobleigh. Assume-guarantee verification of source code with design-level assumptions. In *Proceedings of the 26th international conference on software engineering*, pages 211–220. IEEE Com-

puter Society, 2004. 2.7

[77] Anupam Datta, Ante Derek, John C Mitchell, and Arnab Roy. Protocol composition logic (pcl). *Electronic Notes in Theoretical Computer Science*, 172:311–358, 2007. 2.7

[78] Limin Jia, Shayak Sen, Deepak Garg, and Anupam Datta. A logic of programs with interface-confined code. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 512–525. IEEE, 2015. 2.7

[79] Jamieson M Cobleigh, George S Avrunin, and Lori A Clarke. Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 97–108. ACM, 2006. 2.7

[80] John Rushby. Formal verification of mcmillan's compositional assume-guarantee rule. *Computer Science Laboratory SRI International*, 2001. 2.7

[81] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: a retrospective. In *Workshop on New security paradigms*, pages 87–95, New York, NY, USA, 2000. ACM. 2.8, 3.9

[82] Kevin Hamlen. *Security policy enforcement by automated program-rewriting*. PhD thesis, Cornell University, 2006. 2.8, 3.9, 4.4, 5.2

[83] Steven R. Snapp, James Brentano, Gihan V. Dias, Terrance L. Goan, L. Todd Heberlein, Che lin Ho, Karl N. Levitt, Biswanath Mukherjee, Stephen E. Smaha, Tim Grance, Daniel M. Teal, and Doug Mansur. Dids (distributed intrusion detection system) - motivation, architecture, and an early prototype. In *In Proceedings of the 14th National Computer Security Conference*, pages 167–176, 1991. 3.1, 3.3, 3.3, 3.8

[84] JR Winkler. A unix prototype for intrusion and anomaly detection in secure networks. In *Proceedings of the 13th National Computer Security Conference*, pages 115–124, 1990. 3.1, 3.8

[85] Harold S Javitz and Alfonso Valdes. The sri ides statistical anomaly detector. In *Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on*, pages 316–326. IEEE, 1991. 3.1, 3.8

[86] Kathleen A Jackson, David H DuBois, and Cathy A Stallings. An expert system application for network intrusion detection. Technical report, Los Alamos National Lab., NM (United States), 1991. 3.1, 3.8

[87] Stuart Staniford-Chen, Steven Cheung, Richard Crawford, Mark Dilger, Jeremy Frank, James Hoagland, Karl Levitt, Christopher Wee, Raymond Yip, and Dan Zerkle. Grids-a graph based intrusion detection system for large networks. In *Proceedings of the 19th national information systems security conference*, volume 1, pages 361–370. Baltimore, 1996. 3.1, 3.3, 3.3, 3.3, 3.6, 3.8

[88] Giovanni Vigna and Richard A Kemmerer. Netstat: A network-based intrusion detection system. *Journal of computer security*, 7(1):37–71, 1999. 3.1, 3.6, 3.8

[89] Hervé Debar and Andreas Wespi. Aggregation and correlation of intrusion-detection alerts. In *International Workshop on Recent Advances in Intrusion Detection*, pages 85–103. Springer, 2001. 3.1, 3.6, 3.8

[90] Gregory B White, Eric A Fisch, and Udo W Pooch. Cooperating security managers: A peer-based intrusion detection system. *IEEE network*, 10(1):20–23, 1996. 3.1, 3.8

[91] Jiahai Yang, Peng Ning, X Sean Wang, and Sushil Jajodia. Cards: A distributed system for detecting coordinated attacks. In *Information Security for Global Information Infrastructures*, pages 171–180. Springer, 2000. 3.1, 3.4.3.1, 3.4.6.2, 3.5, 3.6, 3.8

[92] Peng Ning, Sushil Jajodia, and Xiaoyang Sean Wang. Abstraction-based intrusion detection in distributed environments. *ACM Transactions on Information and System Security (TISSEC)*, 4(4):407–452, 2001. 3.1, 1, 2, 4, 3.4.3.1, 3.4.6.2, 3.5, 3.5.2, 2, 3.6, 3.8

[93] Peng Ning, Sushil Jajodia, and Xiaoyang Sean Wang. Design and implementation of a de-

centralized prototype system for detecting distributed attacks. *Computer Communications*, 25(15):1374–1391, 2002. 3.1, 3.5, 3.6, 3.8

[94] Christopher Kruegel and Thomas Toth. Distributed pattern detection for intrusion detection. In *Proceedings of the Network and Distributed System Security Symposium*. Citeseer, 2002. 3.1, 3.3, 3.3, 3.3, 3.5, 3.6, 3.8

[95] Christopher Kruegel, Fredrik Valeur, and Giovanni Vigna. *Intrusion detection and correlation: challenges and solutions*, volume 14. Springer Science & Business Media, 2004. 3.1, 3.3, 3.3, 3.3, 3.3.1, 3.3.1, 1, 3, 3.5, 3.5.2, 2, 3.6, 3.8, 5.2

[96] Steven M Bellovin. Distributed firewalls. *Journal of Login*, 24(5):37–39, 1999. 3.1, 3.6, 3.8

[97] Sotiris Ioannidis, Angelos D Keromytis, Steve M Bellovin, and Jonathan M Smith. Implementing a distributed firewall. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 190–199. ACM, 2000. 3.1, 3.8

[98] David Peleg. Distributed computing. *SIAM Monographs on discrete mathematics and applications*, 5, 2000. 2, 3.2, 3.2, 3.2, 3.6

[99] Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004. 3.2, **??**, 3.4.4, 12, 3.4.7.3, 3.6

[100] Koral Ilgun, Richard A Kemmerer, and Phillip A Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE transactions on software engineering*, 21(3):181–199, 1995. 3.3.1, 3.3.1, 1, 2, 5.2

[101] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990. 3.3.1

[102] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991. 3.3.1

[103] Úlfar Erlingsson. *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Cornell University, Ithaca, NY, USA, 2004. 3.3.1, 2, 3.4.6.3, 4.4, 5.2, 5.2

[104] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–314, New York, NY, USA, 2005. ACM. 3.3.1, 3.4.1.3, 2, 3.4.6.3, 3.9, 5.2, 5.2

[105] Irem Aktug and Katsiaryna Naliuka. Conspec – a formal language for policy specification. *Science of Computer Programming*, 74(1-2):2 – 12, 2008. Special Issue on Security and Trust. 3.3.1, 2, 3.9, 5.2

[106] Peng Ning, Yun Cui, and Douglas S Reeves. Constructing attack scenarios through correlation of intrusion alerts. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 245–254. ACM, 2002. 3.3.1, 5.2

[107] Frédéric Cuppens and Alexandre Miege. Alert correlation in a cooperative intrusion detection framework. In *IEEE Symposium on Security and Privacy*, pages 202–215. IEEE, 2002. 3.3.1, 5.2

[108] Giovanni Vigna, Fredrik Valeur, and Richard A Kemmerer. Designing and implementing a family of intrusion detection systems. *ACM SIGSOFT Software Engineering Notes*, 28(5):88–97, 2003. 3.3.1, 2, 5.2

[109] Fredrik Valeur, Giovanni Vigna, Christopher Kruegel, and Richard A Kemmerer. Comprehensive approach to intrusion detection alert correlation. *IEEE Transactions on dependable and secure computing*, 1(3):146–169, 2004. 3.3.1, 3.4.3.2, 3.5.2, 2, 5.2

[110] R Sekar, Ajay Gupta, James Frullo, Tushar Shanbhag, Abhishek Tiwari, Henglin Yang, and Sheng Zhou. Specification-based anomaly detection: a new approach for detecting network intrusions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 265–274. ACM, 2002. 3.3.1, 3, 3.3.2, 4.1

[111] Calvin Ko, Manfred Ruschitzka, and Karl Levitt. Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *IEEE Symposium on Security and Privacy*, pages 175–187. IEEE, 1997. 3.3.1

[112] Leen Helmink, Martin Paul Alexander Sellink, and Frits W Vaandrager. Proof-checking a data link protocol. In *International Workshop on Types for Proofs and Programs*, pages 127–165. Springer, 1993. 3.3.1, 5.2

[113] Stephen J Garland and Nancy A Lynch. Using i/o automata for developing distributed systems. *Foundations of Component-Based Systems*, 13:285–312, 2000. 3.3.1, 5.2

[114] Myla Archer, Hongping Lim, Nancy Lynch, Sayan Mitra, and Shinya Umeno. Specifying and proving properties of timed i/o automata using tempo. *Design Automation for Embedded Systems*, 12(1-2):139–170, 2008. 3.3.1, 5.2

[115] Sandeep Kumar. *Classification and detection of computer intrusions*. PhD thesis, Purdue University, 1995. 1

[116] Prem Uppuluri and R Sekar. Experiences with specification-based intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 172–189. Springer, 2001. 3, 3.3.2, 4.1

[117] Tao Song, Calvin Ko, Chinyang Henry Tseng, Poornima Balasubramanyam, Anant Chaudhary, and Karl N Levitt. Formal reasoning about a specification-based intrusion detection for dynamic auto-configuration protocols in ad hoc networks. In *International Workshop on Formal Aspects in Security and Trust*, pages 16–33. Springer, 2005. 3, 3.3.2

[118] Robin Berthier and William H Sanders. Specification-based intrusion detection for advanced metering infrastructures. In *Dependable Computing (PRDC), 2011 IEEE 17th Pacific Rim International Symposium on*, pages 184–193. IEEE, 2011. 3, 3.3.2, 4.1

[119] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M Frans Kaashoek, and Zheng Zhang. D3s: Debugging deployed distributed systems. In

NSDI, volume 8, pages 423–437, 2008. 4, 3.4.1.3, 2

[120] Fred Cohen. Computational aspects of computer viruses. *Computers & Security*, 8(4):297–298, 1989. 3.3.2, 5

[121] Steven T Eckmann, Giovanni Vigna, and Richard A Kemmerer. Statl: An attack language for state-based intrusion detection. *Journal of computer security*, 10(1, 2):71–103, 2002. 2

[122] Dingbang Xu and Peng Ning. Privacy-preserving alert correlation: A concept hierarchy based approach. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 537–546, Washington, DC, USA, 2005. IEEE Computer Society. 3.4.3.2

[123] Michael E Locasto, Janak J Parekh, Angelos D Keromytis, and Salvatore J Stolfo. Towards collaborative security and p2p intrusion detection. In *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, pages 333–339. IEEE, 2005. 3.4.3.2

[124] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984. 3.4.4, 3.4.7.3, 3.6

[125] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990. 3.4.4, 3.4.7.3

[126] Nancy A Lynch, Michael Merritt, and Ronald R Yager. *Atomic transactions: in concurrent and distributed systems*. Morgan Kaufmann Publishers Inc., 1993. 3.5

[127] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991. 3.5

[128] Andreas Bauer and Ylies Falcone. Decentralised ltl monitoring. In *International Symposium on Formal Methods*, pages 85–100. Springer, 2012. 3.5.2, 3.8

[129] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed

computations: In search of the holy grail. *Distributed computing*, 7(3):149–174, 1994. 3.5.2, 3.6, 3.8

[130] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006. 1, 3.6

[131] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, 2007. 1, 3.6

[132] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project voldemort. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pages 18–18. USENIX Association, 2012. 1, 3.6

[133] Aanchal Malhotra, Isaac E Cohen, Erik Brakke, and Sharon Goldberg. Attacking the network time protocol. In *Network and Distributed System Security Symposium*, 2016. 2, 3.6

[134] Aanchal Malhotra and Sharon Goldberg. Attacking ntp's authenticated broadcast mode. *ACM SIGCOMM Computer Communication Review*, 46(1):12–17, 2016. 2, 3.6

[135] Richard P Lippmann, David J Fried, Isaac Graf, Joshua W Haines, Kristopher R Kendall, David McClung, Dan Weber, Seth E Webster, Dan Wyschogrod, Robert K Cunningham, et al. Evaluating intrusion detection systems: The 1998 darpa off-line intrusion detection evaluation. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 12–26. IEEE, 2000. 2

[136] R Durst, T Champion, E Miller, L Spagnuolo, and B Witten. Testing and evaluating computer intrusion detection systems. *Communications of the ACM*, 42(9):15–15, 1999. 2

[137] Phillip A Porras and Peter G Neumann. Emerald: Event monitoring enabling response to anomalous live disturbances. In *Proceedings of the 20th national information systems security conference*, pages 353–365, 1997. 3.6

[138] Ravi Prakash and Mukesh Singhal. Dependency sequences and hierarchical clocks: efficient alternatives to vector clocks for mobile computing systems. *Wireless Networks*, 3(5):349–360, 1997. 3.6

[139] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, 1991. 3.6

[140] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989. 3.6

[141] Colin J Fidge. *Timestamps in message-passing systems that preserve the partial ordering*. 1988. 3.6

[142] Dana Angluin. Local and global properties in networks of processors. In *Proceedings of the twelfth annual ACM symposium on Theory of computing*, pages 82–93. ACM, 1980. 3.6

[143] Jukka Suomela. Survey of local algorithms. *ACM Computing Surveys (CSUR)*, 45(2):24, 2013. 3.6

[144] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011. 3.6

[145] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, 2005. 3.7, 3.7, 9, 4.5

[146] Paul C. Attie and Nancy A. Lynch. Dynamic input/output automata, a formal model for dynamic systems. In *ACM Symposium on Principles of Distributed Computing*, pages

314–316, New York, NY, USA, 2001. ACM. 23, 5.2

[147] Thierry Massart and Cédric Meuter. Efficient online monitoring of ltl properties for asynchronous distributed systems. *Université Libre de Bruxelles, Tech. Rep*, 2006. 3.8

[148] Koushik Sen, Grigore Roşu, and Gul Agha. Online efficient predictive safety analysis of multithreaded programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 123–138. Springer, 2004. 3.8

[149] Koushik Sen, Abhay Vardhan, Gul Agha, and Grigore Rosu. Efficient decentralized monitoring of safety in distributed systems. In *Proceedings of the 26th International Conference on Software Engineering*, pages 418–427. IEEE Computer Society, 2004. 3.8

[150] Ylies Falcone, Tom Cornebize, and Jean-Claude Fernandez. Efficient and generalized decentralized monitoring of regular languages. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 66–83. Springer, 2014. 3.8

[151] Christian Colombo and Yliès Falcone. Organising ltl monitors over distributed systems with a global clock. In *International Conference on Runtime Verification*, pages 140–155. Springer, 2014. 3.8

[152] Menna Mostafa and Borzoo Bonakdarpour. Decentralized runtime verification of ltl specifications in distributed systems. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 494–503. IEEE, 2015. 3.8

[153] Joseph Y Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM (JACM)*, 37(3):549–587, 1990. 3.8

[154] Ran Canetti, Ling Cheung, Dilsun Kaynar, Moses Liskov, Nancy Lynch, Olivier Pereira, and Roberto Segala. Task-structured probabilistic I/O automata. Technical Report MIT-CSAIL-TR-2006-060, MIT, 2006. 4, 1, 4.2, 4.2.2, 4.2.2, 3, 7

[155] R. Canetti, Ling Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, and R. Segala.

Task-structured probabilistic i/o automata. In *Proceedings of 8th International Workshop on Discrete Event Systems*, pages 207–214, 2006. 4, 1, 4.2, 7, 4.6

[156] Thomas Ptacek, Timothy Newsham, and Homer J. Simpson. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, DTIC Document, 1998. 4.1

[157] Peter Drábik, Fabio Martinelli, and Charles Morisset. Cost-aware runtime enforcement of security policies. In *Proceedings of the 8th International Workshop on Security and Trust Management (STM 2012)*, pages 1–16, 2013. 4.1, 4.3, 4.6

[158] Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Quantitative languages. In *Proceedings of the 17th International Conference on Computer Science Logic (CSL)*, pages 385–400, 2008. 4.3

[159] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010. 4.4, 5.2

[160] Donald Ray and Jay Ligatti. A theory of gray security policies. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, September 2015. 4.6

[161] Pau-Chen Cheng, Pankaj Rohatgi, Claudia Keser, Paul A. Karger, Grant M. Wagner, and Angela Schuett Reninger. Fuzzy multi-level security: An experiment on quantified risk-adaptive access control. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 222–230, 2007. 4.6

[162] N. Bielova and F. Massacci. Predictability of enforcement. In *Proceedings of the International Symposium on Engineering Secure Software and Systems*, pages 73–86, 2011. 4.6

[163] Peter Drábik, Fabio Martinelli, and Charles Morisset. A quantitative approach for inexact enforcement of security policies. In *Proceedings of the 15th international conference on*

Information Security, ISC'12, pages 306–321, 2012. 4.6

[164] Giulio Caravagna, Gabriele Costa, and Giovanni Pardini. Lazy security controllers. In *Proceedings of the 8th International Workshop on Security and Trust Management (STM 2012)*, pages 33–48, 2013. 4.6

[165] Fabio Martinelli, Ilaria Matteucci, and Charles Morisset. From qualitative to quantitative enforcement of security policy. In *Proceedings of the 6th international conference on Mathematical Methods, Models and Architectures for Computer Network Security: computer network security*, MMM-ACNS'12, pages 22–35, 2012. 4.6

[166] Arvind Easwaran, Sampath Kannan, and Insup Lee. Optimal control of software ensuring safety and functionality. Technical Report MS-CIS-05-20, University of Pennsylvania, 2005. 4.6

[167] Fabio Martinelli and Charles Morisset. Quantitative access control with partially-observable markov decision processes. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, CODASPY '12, pages 169–180, 2012. 4.6

[168] Mariëlle Stoelinga. An introduction to probabilistic automata. *Bulletin of the EATCS*, 78(176-198):2, 2002. 4.6

[169] Jayadev Misra and KM Chandy. Parallel program design: a foundation. *Addison-W esley*, 1988. 5.2

[170] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013. 5.2

[171] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002. 5.2