Models of Secure Software Enforcement and Development

by

Hernan M. Palombo

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy in Computer Science and Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Co-Major Professor: Jay Ligatti, Ph.D.
Co-Major Professor: Hao Zheng, Ph.D.
Xinming Ou, Ph.D.
Nasir Ghani, Ph.D.
Dmytro Savchuk, Ph.D.

Date of Approval:
March 31, 2020

Keywords: Enforceability theory, Security automata, Monitors, Policies, Safety, Liveness,
Ethnography, Secure software development lifecycle

Dedication

To all the special people that have been by my side throughout my Ph.D. journey. In particular, to my family, love partner, and friends, who have always provided invaluable support and encouragement.

Table of Contents

List of Tables

List of Figures

Abstract

Computer Security has been a pressing issue that affects our society in multiple ways. Although a plethora of security solutions have been proposed and implemented throughout the years, security continues to be a problem for at least two important reasons, (1) implementations of runtime enforcement mechanisms have not been modeled rigorously and thus may not be enforcing the policies that are expected to enforce, and (2) there are conflicting tensions in the software development process that hinder the implementation and maintenance of secure software. To investigate these issues, this dissertation is divided into two parts.

The first part of this dissertation takes the lessons learned from earlier models of runtime enforcement—developed over the past nearly twenty years—and proffers a new general model called *Stream-Monitoring Automata (SMAs)*. SMAs unify previous models and is suitable for modeling security mechanisms that operate over infinite event streams, which are now widespread and have been previously left out. SMAs enable the constraints and analyses of interest in previous models to be encoded, and overcomes several shortcomings of existing models with respect to expressiveness. Further, SMAs capture the practical abilities of mechanisms to monitor infinite event streams, execute even in the absence of event inputs, enforce non-safety policies, and operate an enforcement model in which extraneous constraints such as transparency and uncontrollable events may be specified as meta-policies.

The second part of this dissertation presents the results of an ethnographic study of secure software development processes in a software company using the anthropological research method of participant observation. Two Ph.D. students in computer science trained

in qualitative methods worked as software developers in a company for almost two years of total research time, collecting all kinds of information about the development process. The researchers participated in everyday work activities such as coding and meetings, and observed software (in)security phenomena as the software was being developed. They observed developers' reactions to several vulnerabilities that were found by code inspection and pen-testing, and further investigated the issues by interviewing participants and analyzing historical data (code repositories, ticketing system records, internal wikis, and customer-facing documentation). The study found that (1) vulnerability discoveries produce different reactions in developers, often contrary to what a security researcher would predict; (2) security vulnerabilities are sometimes introduced and/or overlooked due to the difficulty in managing the various stakeholders' responsibilities in an economic ecosystem, and cannot be simply blamed on developers' lack of knowledge or skills.

These findings highlight the nuanced nature of the root causes of software vulnerabilities and indicate the need to take into account a significant amount of contextual information to understand how and why software vulnerabilities emerge during software development. Rather than simply addressing deficits in developer knowledge or practice, this research sheds light onto at times forgotten human factors that significantly impact the security of software developed by actual companies. An analysis of the data also shows that improving software security in the development process can benefit from a co-creation model, where security experts work side by side with software developers to better identify security concerns and provide tools that are readily applicable within the specific context of the software development workflow.

Chapter 1: Introduction

This chapter[1] presents the background and the motivations for the two main contributions of this dissertation, a new general and unifying model of runtime enforcement that overcomes several limitations of previous works, and the results of a field study on the different factors that affect the development of secure software in practice.

1.1 Unifying Existing Models of Runtime Enforcement

Runtime enforcement mechanisms or *monitors* oversee untrusted systems behavior, and act on their inputs and outputs to ensure they obey desired policies. Due to the pervasiveness of security threats and attacks, security monitors have become ubiquitous. A mismatch between the security goals and the actual policies being enforced can be catastrophic [2, 3, 4]. Thus, rigorous models of monitors and analysis of the policies that they can enforce are of significant importance.

In recent years, there has been a notable shift towards a model of computation in which programs are producers and consumers of infinite data streams [5, 6]. Some example embodiments include web applications, cloud services, operating systems, and embedded microcontrollers in self-powered devices. The new paradigm's popularity is in part due to the rise of cloud infrastructure providers that offer low-cost storage and processing resources on demand and the reduced production costs of computing devices and their consequent

---

[1]Parts of this chapter are based on a paper [1] published by the author in ICSCA 2020. Permission to use these materials are provided in Appendix A.

proliferation, which are leading to the exponential increase in the amounts of digital data constantly flowing worldwide [7].

Although many models of runtime enforcement have been proposed, they sometimes omit important details or make assumptions which limit their applicability within the new computational paradigm. For example, many earlier enforcement models assume that monitors operate on possibly finite input sequences [8, 9, 10, 11, 12, 13, 14], that they are only enabled on input events [8, 9, 12, 15, 16, 17], or that they are always able to stop the underlying targets being monitored [8, 9, 12, 16]. Often, these and other constraints limit the models' enforcement power, e.g. to only enforce a subset of safety properties [8, 10, 11, 18, 15], or allow them to enforce exceptional policies in sometimes impractical scenarios [9] (as discussed in [16]). Remarkably, these limitations make many previous frameworks unsuitable for modeling monitors of black-box targets that process infinite event streams, which are now widespread.

The present work takes the lessons learned from the past and presents a new model of runtime enforcement that generalizes and unifies previous works, and is suitable for modeling mechanisms that operate on infinite event streams. In the new model, called Stream-Monitoring Automata (SMAs), monitors may transform target inputs and outputs but have no access to their program or machine code.

The SMA model overcomes several shortcomings of previous models—SMAs capture practical abilities of monitors to proceed even in the absence of input events, and lift the arguably impractical assumption that monitors can terminate the underlying targets' executions. These key differences in the way monitors are modeled have a significant impact on the policies that can be enforced. The properties that SMAs enforce are characterized, revealing that some non-safety properties that are left out of other models can be practically enforced. Further, previous analyses are unified by encoding various extraneous enforcement requirements into the model. The new model's contributions are outlined as follows.

– Runtime mechanisms are modeled as Stream-Monitoring Automata (SMAs), transition systems that mediate events in infinite input streams. A distinctive feature of SMAs is their ability to affect the targets being monitored even in the absence of input events. Another benefit is their simple operational semantics, which are defined with one rule—and thus are amenable for formal analysis.

– The properties that SMAs can soundly enforce are characterized as *game* properties, which are explained by making an analogy between property enforcement and 2-player infinite games.

– Relationships established with safety and liveness properties reveal that SMAs can enforce some non-safety properties that were previously considered unenforceable. Conversely, SMAs cannot enforce some safety properties that were previously considered enforceable. Specifically, SMAs cannot enforce some properties requiring that the monitor receives certain input events.

– The model's extensibility is shown by encoding different definitions of enforcement, e.g. transparency and uncontrollable events [13] as meta-policies, i.e. policies about policies. The main advantage of meta-policies is that special constraints about how traces must be transformed can be specified, avoiding the need to hardcode them into the model.

– The model is further extended by considering SMAs that monitor batches of concurrent events and nondeterministic SMAs. Batch and nondeterministic SMAs are useful for modeling some properties of distributed systems, and while they have slightly different transition functions, they have the same enforcement power as standard SMAs.

– The model's generality is demonstrated with simulations of alternative security automata models. Simulations also highlight the effects that different modeling assumptions have on properties' enforceability.

1.2 Understanding Secure Software Development in Practice

It has long been recognized that human factors play a dominant role in the ever-present software vulnerabilities, and substantial research has been devoted to this area [19, 20, 21, 22, 23, 24, 25, 26]. Past efforts have used a variety of research methods including surveys/interviews, controlled experiments, studying code artifacts, and analyzing data collected from secure-coding competitions. It is also understood that there is a fundamental economic problem underlying software insecurity [27], and in general there appears to be unwillingness in industry to put code security at equal importance to other business considerations, such as time to market and richness of features. It is therefore important to recognize that the (in)security of software produced by software companies are impacted not only by individual developers' knowledge and skills or the types of programming languages/environment they use, but also by the various incentives at play both at the market and the organizational levels. To produce real impact in secure software development, it is indispensable to study this problem within the context where the process happens, i.e., in the software companies.

Recent work by Sundaramurthy [28, 29] showed that by employing the anthropological research method of participant observation, researchers have successfully obtained deep insights into the challenges faced by security analysts in security operations centers (SOCs) and embeddings in the SOCs allowed researchers to produce both technical and non-technical interventions that improved SOC operations, by uncovering and addressing the pain points in the overall work process and environment. Inspired by that work, an extensive ethnographic study was conducted in a software company using the method of participant observation.

Two Ph.D. students in computer science were trained in qualitative research methods by an anthropologist and spent 23 months of total research time doing fieldwork in the company. They participated in everyday work activities in the company such as coding and meetings, and observed developers' and managers' reactions as security vulnerabilities were discovered. The discoveries produced different reactions in the development team, often times contrary to what a security researcher would predict. In addition to documenting live interactions, the researchers investigated historical data (code repositories, ticketing system records, and internal/external documentation) to gather further evidence of the motivations and challenges that could have prompted developers to introduce and in some cases keep security flaws in their code. The fieldworkers then shared their observations with a larger research team that included an anthropologist and computer science professors collaborating on this research, and correlated the data to identify adverse patterns that prevented developers from realizing and maintaining secure software.

The results in this research expose social aspects that threaten secure development. In particular, incentive structures, organizational relationships, work flow, and other contextual factors shape how security considerations come into play (or not) during the development and maintenance of software. The study's contributions can be summarized as follows.

– To the best of our knowledge, the study is the first to apply the anthropological research method of participant observation to conduct an ethnographic study of secure development practices in a software company for a total research time of 23 months. Further, this is also one of the few studies in which the embedded researchers are experienced computer scientists specialized in security and trained in qualitative research methods. The main benefits of this methodology is twofold. First, developers can be observed in a natural setting for an extended period of time, which allows the researchers to witness changes in developers behaviors as issues evolve throughout time. Second, because the

researchers are technical experts in the field, they are able to naturally establish trust relationships and quickly identify security issues.

– The study reveals that security vulnerabilities are sometimes introduced and/or over-looked due to the difficulty in managing the various stakeholders' responsibilities in an economic ecosystem, and cannot be simply blamed on developers' lack of knowledge or skills. An extended analysis of the data collected from developers' interactions, code repositories, ticketing systems, and other documentation suggests that there are adverse patterns that often threaten secure software development.

– The study also finds that the approach taken by researchers on their interventions may also affect the way that developers react and follow up on the discovered vulnerabilities. Results from the different interventions suggest that the security research community should move from an intervention model centered around developers' deficits towards a more fruitful co-creation model of security.

1.3 Outline

The first part of this dissertation focuses on Stream-Monitoring Automata (SMAs). Chapter 2 reviews related works in runtime enforcement, and classifies them according to their features and limitations. Chapter 3 defines SMAs, enforcement, and game properties, which characterize the set of soundly enforceable properties. The chapter also presents a hierarchy of properties in which game properties are compared to safety and liveness, and examples in each class are given. Chapter 4 presents several extensions to the SMA model: the formalization of extraneous constraints as meta-policies, SMAs that monitor batches of concurrent events, and non-deterministic SMAs. Chapter 5 shows how other security automata can be simulated by SMAs.

The second part of this dissertation focuses on the ethnographic study of secure software development. Chapter 6 reviews related ethnographic works on software development. Chapter 7 explains the methodology used in the study discussed here and the security vulnerabilities discovered by the researchers. Chapter 8 describes adverse patterns threatening secure software development that resulted from an extended analysis of field notes. These patterns are correlated with the historical data to explain the developers' attitudes towards the security issues found by the researchers. The chapter ends with some discussions about how security experts' interventions can be come more fruitful.

Chapter 2: Related Works In Runtime Enforcement Models

Because there is an enormous body of works about runtime enforcement, this survey is focused on what we consider highly influential works and related models with particular characteristics of interest that are in the scope of our research. More specifically, we first characterize models based on systems' semantics, monitors' semantics, and definitions of traces, policies, and enforcement. Then, we analyze previous models' provably enforceable policies with respect to several important classes that were previously defined and studied in the literature. In general, related works may be classified in many different ways; we do not claim that this is the only classification possible. However, it highlights many important characteristics of the different models, especially considering the focus of our work and the many papers that we have studied.

2.1 Systems Semantics

In this section, runtime enforcement models are analyzed based on the semantics of the systems on which the monitors operate.

2.1.1 Sequential and Monolithic Systems

A large body of works have modeled monitors that operate on sequential and monolithic systems. We call this class of enforcement mechanisms SM.

Schneider is one of the first to characterize the EM (Execution Monitoring) class of runtime mechanisms that enforce policies by monitoring execution steps of a target. A security automaton in EM receives sequential inputs from a program and terminates an

execution that is about to violate a security policy. For a policy to be enforceable by a mechanism in EM, it must be defined as a predicate on a single trace, and thus as a safety property.

Ligatti et al. proposed Edit Automata [9] as a generalization of Schneider's security automata (called truncation automata in [9]) to model mechanisms that can also *insert* or *suppress* events from traces. As in truncation automata, Edit Automata's policies are defined on programs' output traces, sequential executions are assumed, and monitors are modeled as mechanisms interposed between a program and its environment. Because Edit Automata can buffer arbitary sequences of input events, they can enforce some non-safety properties called infinite renewal.

Other works that were derived from Schneider and Ligatti et al.'s works (e.g. [10, 11, 12, 13]) also model monitors in SM, i.e. monitors operating on sequential and monolithic systems.

2.1.2   Concurrent Events and Networked Components

Although many works on runtime enforcement assume that monitors receive serialized input events from a single program/system being monitored, event serialization and single-target enforcement is not always possible or desirable. For example, a security monitor of a system-on-chip implementation may observe signals on different wires concurrently, or a network intrusion-detection system may need to oversee parallel activities on different hosts to detect distributed denial-of-service attacks. We call the class of mechanisms that can observe concurrent events on a network of components CN.

Concurrent events can be usually encoded in models that consider enforcement of hyperproperties (e.g. [15, 30, 31]) but in general these models were developed for monitors that act as program wrappers (single-target enforcement) or for monitors that operate in be-

tween two targets, e.g. a target application and an operating system. Monitors of distributed systems must usually oversee many targets and links.

Networked targets can be naturally encoded as graphs with multiple links. Eearly models and derived works ([8, 9, 10, 13, 17, 12, 11]) did not consider multi-link systems. Recently, MRAs [16] incorporated pairs of links between a monitor and an application (operating system) to model mandatory-results scenarios. However, MRAs do not model distributed systems (at most two links between two targets can be monitored). Distributed operations on multiple links can be encoded in Mallios et al.'s I/O automata-based enforcement model [18] but monitors' inputs are always serial. Yet, other important model features were left out in their work; e.g. flexible definitions of enforcement or non-safety property enforcement.

Several authors have addressed runtime enforcement for networked targets or components (e.g. [32, 33, 34, 35]). Yet, their works cannot be fully generalized; often monitors have limited enforcement power ([32, 35]) or operate on specific architectures ([33, 34]).

2.2 Monitors' Semantics

In this section, runtime enforcement models are analyzed for specific constraints in the semantics of monitors that have an impact on which policies are enforceable at runtime.

2.2.1 Delayed Outputs Disallowed

In general, runtime-enforcement requires that monitors must output some event from a set of possible events to get the next input event (e.g. [8, 16, 15]). When events that delay enforcement decisions (e.g. empty outputs) are not included in the set of possible outputs, monitors must ensure the validity of their trace(s) after every output. We call the class of mechanisms for which delayed outputs are disallowed DD. Notice that in practice some monitors may be able to delay outputs and wait for new inputs before making an enforcement decision because new inputs will always eventually arrive. This scenario is

possible when enforcing policies on programs for which new inputs can be certified, e.g., due to some environment invariant or the monitor's ability to read possibly-emtpy inputs at-will.

2.2.2 Security-Relevant Input Enabledness

Many mechanisms (e.g. [8, 9, 12, 15, 16]) also have a constraint that prevents them from executing in the absence of security-relevant input events. We call this class of mechanisms SR. This constraint may not reflect how many monitors behave in practice. For instance, a monitor might occasionally probe a network to detect which hosts are listening on a specific port, despite any other observed security-relevant activity. In general, monitors in SR are limited to enforcing safety properties because new security-relevant events may not be guaranteed to arrive in the future, so the monitor must always ensure that its output is valid.

2.2.3 Bounded-buffering

Some previous works have studied mechanism with bounded-buffering capabilities. We call the class of mechanisms with bounded-buffering capabilities BB. Talhi et al. [11] introduced Bounded History Automata, execution monitors constrained by memory limitations. Viswanathan [36] and Kim et al. [37] established tighter bounds on the power of runtime monitors by adding computability constraints to the definition of the automata, and Viswanathan [36] also showed that these computable safety properties are equivalent to CoRE properties. Certainly, bounded-buffering models are useful in many practical settings. Fong [10] has shown that mechanisms with a shallow-history are able to enforce many practical properties. Yet, bounded-buffering monitors add a restriction that may leave out other interesting scenarios (e.g. monitoring as a service).

2.2.4 Passive Mechanisms

Some mechanisms monitor executions to detect policy violations but do not edit traces. We call this class of mechanisms PM. In general, these mechanisms (e.g. [38, 31]) have the enforcement power of Truncation Automata because they can only output a yes/no (or maybe [38]) answer.

2.2.5 Monitors with Static Information

Some works have analyzed enforcement for monitors with some static information. We call this class of mechanisms SI.

Pinisetty et al. proposed Predictive Runtime Enforcement (PRE) [17], a model of monitors that have some a priori knowledge of the system and can therefore anticipate their behavior. PRE monitors can enforce regular properties, i.e. languages accepted by an automaton, and satisfy monotonicity and urgency constraints. Monotonicity means that the enforcer cannot undo what is already released as output during the incremental computation. The urgency constraint is related to releasing events as output as soon as possible. PRE extends other previous works on gray-box enforcement (e.g. [39]) by showing that runtime mechanisms can enforce some non-safety properties but is limited to properties that are defined over output traces only, i.e. that are not monitor-centric.

Imanimehr and Fallah [40] studied the enforcement power of runtime monitors with static information. They modeled monitors that have a possibly inaccurate approximation of the target's possible executions, and extend the definitions of effective and precise enforcement given by Ligatti et al. [9] with a parameterized upper-bound of the distance between the approximation and the exact traces generated by the program.

Ligatti et al. [41] analyzed the enforcement power of monitors when the set of possible executions is nonuniform, i.e. a subset of all possible inputs. Nonuniformity models the

scenarios in which a monitor may know *a priori* that it will never be given as input certain executions, e.g., because some other security mechanism such as a type checker has already ruled out those executions. In general, nonuniformity extends the set of properties that can be enforced by a mechanism. For instance, if all input executions are guaranteed to be nonterminating, a truncation automaton may be able to enforce some nonsafety properties. E.g., an otherwise unenforceable nontermination property that requires that every execution not terminate could be enforced by accepting all actions.

2.2.6 Distributed Enforcement

Some authors have modeled mechanisms that enforce policies distributedly. We call this class of mechanisms DM.

Service automata [42] is a framework to implement monitors in distributed networks. Each monitor must implement four components: an interceptor, a coordinator, an enforcer, and a local policy module. The framework defines the behavior of the interceptor and coordinator, and it only specifies the interface for the enforcer and local policy module, which have to be instantiated by an implementation.

RVDist is an implementation of a runtime verification framework for component-based systems in which controllers are attached to local schedulers and events are then then sent to a global observer that reconstructs traces and evaluates them [35]. RVDist has the enforcement power of truncation automata.

Pinisetty and Tripakis [43] analyzed how to compose distributed runtime enforcement mechanisms. They studied two monitor composition schemes, serial and parallel composition, and showed that, while enforcement under these schemes is generally not compositional, it is for certain subclasses of regular properties.

2.3 Definitions of Traces, Policies, and Enforcement

Definitions of traces, policies, and enforcement can have a significant impact on the capabilities of a mechanism. Next, we analyze related works based on how traces were defined (with respect to inputs and outputs), and how extraneous constraints (such as transparency and uncontrollable events) are incorporated into the models.

2.3.1 Output-only Traces

In most previous models, traces are defined only on monitors' outputs (e.g [8, 9]). We call this class of mechanisms OO. These mechanisms often enforce properties that define requirements for inputs and outputs of programs, i.e. program centric policies. However, policies that specify requirements about how monitors may transform inputs into outputs— i.e. monitor-centric policies—cannot be specified in mechanisms in OO.

2.3.2 Input/output Traces

Some models also incorporate monitor inputs into the traces (e.g. [18, 16, 15]). Therefore, monitor-centric policies can be specified. Typical monitor-centric policies include those specifying filtering requirements since they often specify relationships between inputs and outputs. We call the class of mechanisms that can enforce monitor-centric policies IO.

Ligatti et al. [16] modeled runtime mechanisms as Mandatory Results Automata (MRAs). MRAs operate on systems consisting of a target application and an executing system. The target application sends actions to the executing system and the executing system sends results for those actions back to the target application. One of MRAs most significant contributions was that by including inputs to the monitor in traces, input-output relationships such as observable and partially controllable events could be encoded into the model.

Mallios et al. [18] modeled runtime enforcement mechanisms as I/O automata [44], a labeled transition model for concurrent systems. I/O automata make a distinction between input, output, and internal actions which allows them to enforce monitor-centric policies (i.e. those that define relationships between inputs and outputs). I/O automata can encode distributed systems by defining *tasks*, an equivalence relation partitioning the set of local (output and internal) actions into a countable number of equivalence classes. Yet, the I/O automata monitors in [18] can enforce only prefix-closed properties, e.g. safety.

Ngo et al.'s modeled mechanisms on black-box reactive programs (which we call RMs), and their traces include monitor inputs and outputs. RMs can enforce monitor-centric policies; yet they cannot delay outputs so they are limited to safety enforcement.

2.3.3   Hard-coded Constraints

Some previous models were proposed based on extraneous constraints on the traces that monitors could produce, and these constraints were usually hard-coded in the definitions of enforcement. We call the class of mechanisms that are bounded by extraneous constraints HC.

For example, Basin et al. [13] proposed a distinction between system actions that are controllable by an enforcement mechanism and those that are only observable. Khoury et al. [14] further distinguished sets of events that can only be partially controlled, i.e. events that can only be inserted or suppressed. Renard et al. [45] extended work on uncontrollable events by studying enforcement of timed properties, and showed how to synthesize runtime enforcement mechanisms at two levels of abstraction to facilitate their design and implementation. Their work claims that, in the timed setting, policies should also satisfy a monotonicity constraint reflecting the streaming of events: the output sequence can only be modified by appending new events to its tail [46].

2.3.4 Extensible Constraints

Some models have attempted to generalize extraneous constraints and incorporate them into the definitions of policies. We call the class of mechanisms that may be flexibly bounded by arbitrary constraints CE.

For example, Khoury and Tawbi [47] analyze monitors that preserve some equivalence relation between its inputs and outputs, give examples of meaningful relations and identify the policies that are enforceable with their use. Further, they investigate how a priori knowledge of the target's program behavior would increase the monitor's enforcement power. Dolzhenko et al. [16] demonstrated that constraints such as transparency and uncontrollable events define subsets of the policies that are enforceable by Mandatory-Results Automata.

2.4 Provably Enforceable Policies

This section juxtaposes the related works reviewed in Sections 2.1, 2.2, and 2.3 with respect to the sets of policies that are provably enforceable in each model. In particular, it focuses on enforcement of non-safety properties because they have been left out in many models despite their commonality, and non-interference, due to the continued interest from the research community.

2.4.1 Non-safety Properties

Some works have modeled mechanisms that can enforce non-safety properties (e.g. [9, 41, 48, 17]). We call the class of mechanisms that can enforce non-safety properties NS.

Ligatti et al.'s edit automata can enforce infinite renewal properties [9, 41]. A renewal property is one in which every valid infinite-length sequence has infinitely many valid prefixes, and conversely, every invalid infinite- length sequence has only finitely many valid prefixes.

Falcone et al. analyzed rutnime verification and enforcement of properties in the safety-progress classification [49, 48]. This classification is an alternative to the classical safety-liveness view, and includes four classes of properties. Precisely, a property P is:

– safety if all the traces in P have all their prefixes in P,

– guarantee if all the traces in P have some prefix in P,

– response if all the traces in P have an infinite number of prefixes in P, and

– persistence if the traces in P have a finite number of prefixes not in P [50].

Pinisetty et al. developed a model of mechanisms that can enforce regular properties of systems for which some information is known a priori [17]. A regular property is a language accepted by an automaton, and mechanisms in this model can output some events immediately, satisfying soundness, transparency, monotonicity, and urgency constraints.

Note that neither edit automata, nor Falcone's, nor Pinisetty's models are suitable for encoding distributed systems or monitor-centric policy enforcement. Models that include monitor inputs in their traces (e.g. [18, 16]) could possibly enforce some non-safety properties if some system assumptions were modified (e.g. that new inputs will always eventually arrive). Moreover, it could be argued that exchange-safety properties enforceable by MRAs [16] were a superset of traditional safety properties (as defined by Lamport [51]) because they include inputs to the monitor in their traces, and thus MRAs enforced some non-(traditional) safety properties. However, these conjectures are only speculations and would require further formal analysis to be confirmed.

2.4.2 Non-interference

Within enforcement of hyperproperties, there has been particular interest in non-interference properties. Noninterference is a confidentiality policy which requires that ex-

ecutions from users (or processes) with high security clearances (high executions) have no effects on executions from those with low clearances (low executions). We call the class of mechanisms that can provably enforce noninterference NI.

Devriese and Piessens [52] introduced secure multi-execution, a non-interference enforcement technique that executes a program multiple times, once for each security level, using special rules for I/O operations. Inputs are replaced by default inputs except in executions linked to their security level or higher. Input side effects are supported by making higher-security-level executions reuse inputs obtained in lower-security-level threads.

Askarov et al. [53] proposed a monitoring framework to enforce progress-sensitive information-flow control on concurrent programs. A program is progress-sensitive if confidential information influences whether it terminates. Thus, observing termination or non-termination of such a program can leak confidential information. Their monitors are hybrid: they combine dynamic and static program analysis to enforce security both at a local-thread and global program level.

2.5 Summary

Table 2.1 summarizes the related works reviewed in the previous sections. Despite many models of runtime enforcement have been proposed in the nearly twenty years, previous models have not been unified or generalized. There are still many questions that remain unanswered. For example, *can any policies that were left out of previous models be practically enforced at runtime?* Clearly, the answer to this question depends on the framework being used for analysis, i.e. it relies on the definitions of systems, monitors, policies, and enforcement. A unified model that incorporates all the features of previous models is expected to bring light on an answer. Further, a precise analysis of the policies that are enforceable will be required. A follow up question is, *how would existing models compare to the unified model?* Results in this area demand simulations. And finally, *how can previous*

Table 2.1: Classification of Related Works.

		Class	Related Works
System Semantics	Sequential and monolithic	SM	[8, 9, 10, 11, 12, 13, 14]
	Concurrent and networked	CN	[32, 33, 34, 35]
Monitor Semantics	Delayed outputs disallowed	DD	[8, 16]
	Security-relevant enabledness	SR	[8, 9, 12, 15, 16]
	Bounded buffering	BB	[11, 10, 15]
	Passive mechanisms	PM	[38, 31]
	Static information	SI	[54, 55, 53, 40, 39, 17]
	Distributed enforcement	DM	[42, 43]
Traces, Policies, and Enforcement	Hard-coded constraints	HC	[13, 14, 45]
	Extensible Constraints	CE	[47, 16]
	Output-only traces	OO	[8, 9, 12, 10, 17, 13, 11]
	Input-output traces	IO	[18, 16, 15]
Provably Enforceable Properties	Nonsafety properties	NS	[9, 48, 17]
	Noninterference	NI	[52, 53, 15]

requirements/constraints be generalized such that the new model is simple yet extensible to accommodate specific needs? Arbitrary enforcement definitions and restrictions about how monitors are able to produce traces can potentially be placed as policy constraints. The focus of the remaining chapters is to make progress on finding answers to these and other related questions in this exciting field.

Chapter 3: Stream-Monitoring Automata

This chapter[2] presents a model of runtime enforcement mechanisms that mediate infinite streams of events generated by untrusted blackbox targets.

3.1   Background and Notation

SMAs mediate input and output events of untrusted black-box targets. There is no restriction about what constitutes a target. For example, a target can be a program, a hardware component inside a chip, or a host connected to the Internet.

Every target is associated with a countable set of events $E = E' \cup \{\epsilon\}$, where E' are the target's possible inputs and outputs, and ϵ is a special nil event. The definition of what constitutes an event depends on the context. Events can be thought of as words, strings, commands, or actions/results in reactive systems.

A key difference between SMAs and previous models is the explicit characterization of nil events. Nil events constitute a void result from a monitor's proactive sampling of inputs. In other words, a nil event is an empty or blank observation. In all previous security automata models of which we are aware, monitors are reactive, i.e. the monitors' computations are blocked when the monitored target provides no inputs to the monitor. In contrast, SMAs are proactive, sampling inputs and possibly making transitions even on nil events.

A stream is an infinite sequence of events $e_0 e_1 e_2...$ such that $\forall i \in \mathbb{N} : e_i \in E$. The set of streams is denoted E^ω. Similarly, the set of finite (finite or infinite) sequences of events is

---

[2]Parts of this chapter are based on a paper [1] published by the author in ICSCA 2020. Permission to use these materials are provided in Appendix A.

denoted E^* (E^∞), i.e. $E^\infty = E^* \cup E^\omega$. For any set S, we write S_f to mean that the set has a finite number of elements (cardinality). Metavariable i (o) ranges over E and is used to denote an event that is input to (output from) the monitor. The wildcard symbol $\_$ is used to signify any event in E that is not bound to a variable.

Events are an abstraction of a target's runtime activity as observed by the monitor. An underlying monitoring infrastructure is assumed to be permanently available, yet there is no requirement that the target is active (e.g. an observed event can be nil, i.e. ϵ). Permanently-available infrastructure is nowadays common in practice; modern systems are often designed and effectively implemented to be available and fault-tolerant. For example, surge protectors, UPS devices, and power generators prevent power spikes and outages, RAIDs prevent data loss from disk failures, etc.

3.2 Stream-Monitoring Automata

This section defines SMAs and their operational semantics.

Definition 1. *A Stream-Monitoring Automaton (SMA) M is a tuple (E, Q, q_0, δ), where E is a recursively enumerable set of events such that $\epsilon \in E$, Q is a recursively enumerable set of automaton states, $q_0 \in Q$ is an initial state, and $\delta : Q \times E \to Q \times E$ is a computable and total transition function.*

An important feature that distinguishes SMAs from other automata-based transducers (e.g. Mealy machines [56]) is that SMAs may have a countably infinite set of states. It is common for security automata frameworks to allow states to be infinite [8, 9, 16]. Yet, some previous works have considered the effects of limiting the number of states in security automata [10, 11].

Example 1 (Filter-and-sanitize SMA). *Consider an SMA M_1 that monitors a web server receiving queries from clients. M_1 filters queries from bad sources according to the computable*

function allow : $E \to \{0, 1\}$, *and sanitizes the contents of the queries using the computable function sanitize* : $E \to E$. M_1's *transition function is defined by* $\delta(q, i) = (q, sanitize(i))$ *if allow*(i); *otherwise* $\delta(q, i) = (q, \epsilon)$.

Example 2 (Log SMA). *Consider an SMA* M_2 *that logs all activity observed by issuing a store operation* $st(f) \in E$ *where* f *is a file generated using function log* : $E^* \to \{f \mid f$ *is a binary file*$\}$. M_2's *set of states is* $Q = \{v \mid v \in E^*\}$, *and its initial state is the empty sequence. Let* $|v|$ *be the length of sequence* v *and* k *a user-defined parameter that specifies a threshold size for logging.* M_2's *transition function is defined by* $\delta(v, i) = (vi, \epsilon)$ *if* $|vi| < k$; *otherwise* $\delta(v, i) = (i, st(log(v)))$. M_2 *buffers input events until threshold* k *is met, then creates and stores the log.*

3.2.1 Operational Semantics

An *exchange* is a pair of events $\langle i, o \rangle$, consisting of an input event $i \in E$ (i.e. the event input to the SMA) and an output event $o \in E$ (i.e. the event output from the SMA). The set of exchanges over the input and output events is denoted by $\mathcal{E}$, i.e. $\mathcal{E} \subseteq E \times E$. The operational semantics of SMAs is defined with a labeled single-step judgment whose form is $q \xrightarrow{\langle i, o \rangle}_M q'$, shown in Figure 3.1, where $q, q' \in Q$ are configurations (i.e. states) of an SMA. This judgment indicates that the SMA M takes a single step from configuration q to q' while producing the exchange $\langle i, o \rangle$. Event i is an input event as observed by the monitor, and there are no explicit assumptions on it. In other words, a target can produce any $i \neq \epsilon$ at any time-step of the target's execution, or $i = \epsilon$. By default, SMAs can step on nil input events

$$\boxed{q \xrightarrow{\langle i, o \rangle}_M q'} \qquad \frac{\delta(q, i) = (q', o)}{q \xrightarrow{\langle i, o \rangle}_M q'}$$

Figure 3.1: SMAs' operational semantics. Symbol i (o) denotes an input (output) event, and q, q' are states of SMA M.

and output nil events. More restrictive SMAs can be specified with meta-policies (Section 4.1).

An *execution* or *trace* is an infinite sequence of exchanges, i.e. $x \in \mathcal{E}^\omega$, where $\mathcal{E}^\omega$ is the set of all executions. An SMA M *produces* a trace x if x is in the language of M.

Definition 2. *Given an SMA $M = (E, Q, q_0, \delta)$, the language of M is*

$$\mathcal{L}(M) = \{\langle i_0, o_0 \rangle \langle i_1, o_1 \rangle ... \mid \exists q_1, q_2, ... \in Q : \forall j \in \mathbb{N} : q_j \xrightarrow{\langle i_j, o_j \rangle}_M q_{j+1}\}.$$

The operational semantics of SMAs have three noteworthy features.

– Because SMAs' transition functions are total and input events may be nil (i.e. ϵ), SMAs can output events even when the monitored target is inactive. Formally, this means that for any state q, $\delta(q, \epsilon)$ is defined. SMAs that produce outputs in the absence of inputs model monitors that initiate communications, which are common in practice (e.g. monitors may publish information to other nodes on a network), yet this capability was not considered in previous frameworks [8, 9, 16, 12, 15, 10, 17, 13, 11].

– Because SMAs read inputs and produce outputs incessantly (since their transition function is computable), SMAs do not have the power to stop traces. This model of traces is reasonable in the context of stream-generating black-box targets where enforcement mechanisms may not be able to stop the executions of the targets being monitored and may always continue to receive inputs. In contrast, previous models assumed that monitors had the power to stop the targets being monitored [8, 9, 12, 16].

– As SMA traces include monitor inputs (as similarly done in [16, 18, 15]), SMAs are useful to reason about security policies that specify constraints between monitor inputs and outputs. Because many previous models did not include monitor inputs on traces

[8, 9, 12, 10, 17, 13, 11], requirements about how monitors transformed certain inputs, e.g. filtering policies, were often left out of previous analyses.

Finally, the combination of nil events and total transition functions enable SMAs to simulate monitors with slightly different semantics. For example, monitors that are only enabled on non-nil input events can be encoded by SMAs that define $\delta(q, \epsilon) = (q, \epsilon)$ for any $q \in Q$, and monitors that halt their activities can be encoded by SMAs that output nil events ad infinitum.

3.3 SMA-based Enforcement

This section presents a formal definition of properties, defines what it means for an SMA to enforce a property, and then discusses the properties that are enforceable by SMAs.

3.3.1 Properties

This paper adopts standard definitions of policies and properties [8, 9, 57]. A *policy* is a predicate over sets of executions. Some policies are properties.

Definition 3. *A policy P is a property if there exists a predicate p over $\mathcal{E}^\omega$ such that $\forall X \subseteq \mathcal{E}^\omega : (P(X) \iff \forall x \in X : p(x))$.*

For each property P, let $S_p = \{x \in \mathcal{E}^\omega \mid p(x)\}$ be the set of executions that satisfy predicate p. Note that there is a one-to-one correspondence between a property P, its predicate p, and the set S_p, so the rest of this paper uses P unambiguously to refer to any of the three depending on the context.

3.3.2 Enforcement

Property enforcement is defined in terms of the standard principle of soundness, which is a widely accepted requirement across the security and formal methods communities. SMA M is *sound* with respect to property P when M only produces traces satisfying P.

Definition 4. *An SMA M soundly enforces a property P if $\mathcal{L}(M) \subseteq P$.*

Definition 4 is significantly simpler and more flexible than previous definitions of enforcement because it does not hardcode specific requirements or constraints (e.g. transparency, uncontrollable events), which can be encoded as meta-policies (Section 4.1, [16]).

Example 3 (Filter-and-sanitize Property). *SMA M_1 soundly enforces a filter-and-sanitize property that asserts that only sanitized queries from allowed sources are output, i.e.*

$$P_1 = \{\langle i, o \rangle \mid (\neg allow(i) \lor o = sanitize(i)) \land (allow(i) \lor o = \epsilon)\}^\omega.$$

Example 4 (Log Property). *SMA M_2 soundly enforces a log property which asserts that input events are eventually logged, i.e.*

$$P_2 = \{\langle i_0, o_0 \rangle \langle i_1, o_1 \rangle ... \mid \forall n \in \mathbb{N} : \exists m \geq n : o_m = st(log(...i_n...))\}.$$

3.3.3 Enforceable Game Properties

The set of properties that are soundly enforceable by SMAs can be characterized by new definitions that draw connections to game theory. Intuitively, property enforcement can be interpreted as a two-player game, where the first player is a target and the second player is a monitor, played in a sequence of rounds (traces). Each round has an infinite sequence of hands (exchanges) that are the concatenation of player one's hand (target events that are input to the monitor) and player two's hand (monitor outputs). A property describes the

"winning rules" for the monitor. That is, a property specifies all possible rounds (traces) which dictate that player two wins the game (produces satisfying traces). A property contains a winning-strategy if a plan exists for a monitor to win no matter how the target plays. Precisely, let $i, o \in E$ denote the input and output of an exchange $\langle i, o \rangle$ in an execution $x \in \mathcal{E}^\omega$. A property P contains a winning strategy if $\forall i_0 : \exists o_0 : \forall i_1 : \exists o_1 : ... : p(\langle i_0, o_0 \rangle \langle i_1, o_1 \rangle ...)$. Properties with winning strategies can be enforced if there exists a function that can compute a valid output at every exchange.

Definition 5. *A property P is a game property if there exists a computable function $f : E^* \to E$ such that $\forall i_0, i_1, ... \in E : p(\langle i_0, f(i_0) \rangle \langle i_1, f(i_0 i_1) \rangle ...)$.*

The set of all game properties is denoted GP. An interesting observation is that game properties are closed under superset.

Proposition 1. $\forall P, P' \subseteq \mathcal{E}^\omega : ((P \subseteq P') \wedge (P \in GP)) \implies P' \in GP$.

Proof. Let P be a game property, i.e. $P \in GP$, and $P \subseteq P'$. Because $P \in GP$, we know that there exists a computable function $f : E^* \to E$ such that $\forall i_0, i_1, ... \in E : p(\langle i_0, f(i_0) \rangle \langle i_1, f(i_0 i_1) \rangle ...)$. Since all traces in P are also in P', there exists a computable function $f' : E^* \to E$, precisely $f' = f$, such that $\forall i_0, i_1, ... \in E : p'(\langle i_0, f'(i_0) \rangle \langle i_1, f'(i_0 i_1) \rangle ...)$, meaning P' is also in GP.

$\square$

Game properties characterize the set of properties that are enforceable by SMAs.

Theorem 1. *A property P is soundly enforceable by an SMA iff P is game.*

Proof. Let M be an SMA that soundly enforces some property P, i.e. $\mathcal{L}(M) \subseteq P$. Because M's transition function is total and computable, it is defined for all possible inputs and will always output some event (possibly ϵ). Let $f : E^* \to E$ be a function that takes

any finite sequence of inputs and runs M with those inputs to compute a valid output. Because the traces that form by starting from the empty sequence and concatenating every pair $\langle i_0, f(i_0) \rangle, \langle i_1, f(i_0 i_1) \rangle$... ad infinitum satisfy P (because $\mathcal{L}(M) \subseteq P$), $\mathcal{L}(M)$ is a game property. Further, because $\mathcal{L}(M) \subseteq P$, by Proposition 1, P is a game property, i.e. $P \in GP$.

Let P be a game property, i.e. $P \in GP$, and f a function that computes the outputs of traces in P (which must exist because $P \in GP$). Construct an SMA M such that $Q = \{v \mid v \in E^*\}$, q_0 is the empty sequence, and δ is defined as follows. Given a state v and an input event i, $\delta(v, i) = (vi, f(vi))$. Because every trace that M produces is in P, $\mathcal{L}(M) \subseteq P$.

$\square$

3.4 A Hierarchy of Properties with Examples

Relationships can be established among game and traditional formulations of safety and liveness properties. Recall that safety properties, as first defined by Lamport [51] and later by Alpern and Schneider [8], proscribe "bad things". Let $\prec$ be the strict prefix operator, i.e. $\forall x \in \mathcal{E}^* : \forall y \in \mathcal{E}^\omega : x \prec y \iff \exists x' \in \mathcal{E}^\omega : xx' = y$. The $\succ$ operator is defined symmetrically. A property P is safety if

$$\forall x \in \mathcal{E}^\omega : \neg p(x) \implies \exists x' \prec x : \forall y \succ x' : \neg p(y).$$

On the other hand, liveness properties prescribe "good things", i.e. a property P is liveness [58] if
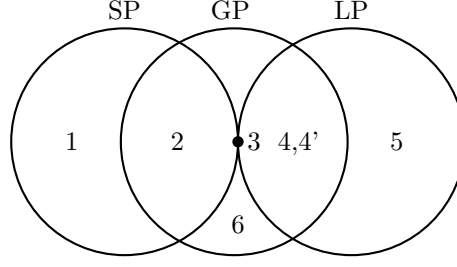
$$\forall x \in \mathcal{E}^* : \exists y \succ x : p(y).$$

The set of all safety (liveness) properties is denoted SP (LP).

Figure 3.2 shows a hierarchy of properties that includes safety, liveness, and game properties, and examples in each class. As expected, SMAs can enforce a myriad of safety properties that are enforceable in other models, including those for integrity and access-control (e.g. classical *no-send-after-read* [8] and *authenticated-login* [9] properties), and filtering properties that specify requirements between monitor inputs and outputs (P_1 in Example 3) [16].

Further, SMAs can soundly enforce some additional properties.

– SMAs can enforce liveness properties that specify requirements about a monitor's eventual obligation or sequence of obligations, e.g. the *log* property P_2 in Example 4, the *eventual-monitor-activity* property $\mathcal{E}^\omega \setminus \langle \_, \epsilon \rangle^\omega$, or the *eventual-monitor-inactivity* property $...\langle \_, \epsilon \rangle^\omega$. Note that, in many cases, there may be multiple ways to enforce non-safety properties. For example, for *eventual-obligation* properties such as $...\langle \_, o \rangle...$ (where o is some arbitrary output event), a monitor may choose to output o right away or after some number of exchanges. The alternatives should be evaluated in context, and the choice may be intentionally left to the implementer. On the other hand, this flexibility can be removed by incorporating specific requirements as meta-policies that, e.g., require obligations to be output within a certain number of exchanges.

– SMAs can enforce some non-safety, non-liveness properties that are the conjunction of some safety and liveness. For example, consider a garbage collector in a programming language's runtime system. Garbage collection demands that all objects that are not referenced by other objects are eventually collected (liveness requirement), and that objects that are referenced by other objects are not collected (safety requirement). Let $collect(r) \in E$ be a memory collection event that effectively frees memory reference r in a set of references R, and $collectable : E^* \to 2_f^R$ a computable function that takes an input sequence and computes a finite set of references that are out of scope and can

Example Properties

1. always-nil-inputs — $\langle \epsilon, \_ \rangle^\omega$
2. filter-and-sanitize — P_1 in Example 3
3. allow-all — $\mathcal{E}^\omega$
4. log — P_2 in Example 4;
4'. eventual-monitor-activity — $\mathcal{E}^\omega \setminus \langle \_, \epsilon \rangle^\omega$
5. target-inactivity — $... \langle \epsilon, \_ \rangle^\omega$
6. garbage-collection — $\{ \langle i_0, o_0 \rangle \langle i_1, o_1 \rangle ... \mid$
$(\forall n : o_n = collect(r) \Rightarrow \exists m \le n : r \in collectable(i_m...i_n)) \wedge$
$(\forall m, n : \forall r \in collectable(i_m...i_n) : \exists p \ge n : o_p = collect(r)) \}$

Figure 3.2: A hierarchy of properties with examples. SMAs can enforce properties in GP. Variables $m, n, p \in \mathbb{N}$.

be marked for collection. The *garbage-collection* property is

$$\{ \langle i_0, o_0 \rangle \langle i_1, o_1 \rangle ... \mid$$

$$(\forall n \in \mathbb{N} : o_n = collect(r) \implies \exists m \le n : r \in collectable(i_m...i_n)) \wedge$$

$$(\forall m, n \in \mathbb{N} : \forall r \in collectable(i_m...i_n) : \exists p \ge n : o_p = collect(r)) \}.$$

Lastly, some properties with specific input constraints are not enforceable, regardless of being safety, liveness, or neither. Broadly speaking, safety properties that only allow some inputs to appear on traces (e.g. the *always-nil-inputs* property $\langle \epsilon, \_ \rangle^\omega$) cannot be enforced because monitors have no control over the inputs received. These properties are enforceable

in previous models that rely on the assumption that monitors are always able to stop target executions, but that assumption is impractical in the context of monitors of black-box targets operating on streams of events. Further, liveness properties that require particular monitor inputs cannot be enforced either because those inputs may not ever come. For example, SMAs cannot guarantee that targets will eventually cease all activity (i.e. ϵ monitor inputs ad infinitum) or conversely always be active. Thus, these *target-activity* properties are not enforceable by SMAs.

Chapter 4: Extensions

This chapter[3] elaborates different extensions to the SMA model. First, extraneous enforcement constraints are considered. These constraints have been traditionally hardcoded into the definitions of enforcement, limiting the models' applicability. Second, SMAs are generalized to model monitors that oversee multiple streams that may generate batches of concurrent events. This generalization enables analyzing property enforcement for many practical mechanisms that oversee parallel executions of untrusted targets. Lastly, nondeterministic SMAs are considered. Nondeterminism can be useful to describe monitoring algorithms more abstractly, e.g. when implementation details are unavailable.

4.1 Extraneous Enforcement Constraints

Models of runtime monitors often need to accommodate extraneous enforcement constraints to be applicable in practical scenarios, e.g. monitors may be required to enforce properties transparently, i.e. by outputting valid inputs verbatim, or deal with uncontrollable events that cannot be modified. SMAs adopt a definition of enforcement that is simpler than those considered by other models and generalize extraneous constraints as meta-policies, i.e. policies about policies (as similarly done in [16]). Traditionally, these constraints have been hard-coded into the definitions of enforcement, limiting the models' applicability. Because meta-policies can specify arbitrary constraints about input and output events, they

---

[3]Parts of this chapter are based on a paper [1] published by the author in ICSCA 2020. Permission to use these materials are provided in Appendix A.

are more fine-grained than previous encodings. Some commonly studied constraints are discussed next.

4.1.1 Transparency Policies

First, consider transparency [41], which requires that valid sequences input to a monitor not be modified. A transparency requirement may be specified to constrain the enforcement power of a monitor so that the input events from the target are output verbatim if they do not violate the property under consideration.

In general, a transparency requirement can be formalized as follows. Let $i_0; i_1; \ldots \in E^\omega$ be an input stream, and $V \subseteq E^\omega$ a set of input streams that are considered valid in the context of transparency. The set of traces that satisfy the transparency requirement, denoted $\mathcal{E}_t^\omega$, is defined by traces with input sequences not in V, or with input sequences in V and outputs that are a verbatim expansion of inputs, i.e.

$$
\begin{aligned}
\mathcal{E}_t^\omega = &\{\langle i_0, o_0 \rangle \langle i_1, o_1 \rangle \ldots \mid i_0; i_1; \ldots \notin V\} \cup \\
&\{\langle i_0, i_0 \rangle \langle i_1, i_1 \rangle \ldots \mid i_0; i_1; \ldots \in V\}.
\end{aligned}
$$

The transparency requirement can be easily incorporated into the SMA model.

Corollary 1. *Let TP be the set of properties that are enforceable by SMAs and satisfy the transparency requirement. Then, $TP = GP \cap \{P \mid P \subseteq \mathcal{E}_t^\omega\}$.*

Proof. ($\subseteq$) Let M be an SMA that enforces a property P that satisfies the transparency requirement for the set of input sequences V, i.e. $\mathcal{L}(M) \subseteq P$ and $\forall x \in P : x \in \mathcal{E}_t^\omega$. Theorem 1 states that enforceable properties are game, so $P \in GP$. Because all the traces in P are in $\mathcal{E}_t^\omega$, $P \subseteq \mathcal{E}_t^\omega$. Thus, $P \in GP \cap \{P \mid P \subseteq \mathcal{E}_t^\omega\}$.

($\supseteq$) Let P be a property in $GP \cap \{P \mid P \subseteq \mathcal{E}_t^\omega\}$. Theorem 1 shows how to construct an SMA M that enforces any property in GP, including P. □

32

4.1.2 Uncontrollable-Events Policies

Uncontrollable or *only observable* events cannot be mediated by monitors [13]; a typical example are clock ticks. On the other hand, events that can be inserted or suppressed by monitors are called *controllable*. Let (E_u, E_c) be a partition of events where E_u, E_c are sets of uncontrollable and controllable events respectively. The uncontrollable events constraint requires that events in E_u that are input to the monitor are not suppressed, and that events in E_u are not output if they are not in the input. Let $\mathcal{E}_u^\omega$ be the set of all traces constructed as the sequences of exchanges that satisfy the uncontrollable events constraint, i.e. $\mathcal{E}_u = \{\langle i, o \rangle \mid (i \in E_u \vee o \in E_u) \implies i = o\}$. Properties about traces with uncontrollable events that are enforceable by SMAs are exactly the game properties that satisfy the uncontrollable events constraint.

Corollary 2. *Let UP be the set of properties about traces with uncontrollable events that are enforceable by SMAs. Then, $UP = GP \cap \{P \mid P \subseteq \mathcal{E}_u^\omega\}$.*

Proof. ($\subseteq$) Let M be an SMA that enforces a property $P \in UP$ that satisfies the uncontrollable events constraint, i.e. $\mathcal{L}(M) \subseteq P$ and $\forall x \in P : x \in \mathcal{E}_u^\omega$. By Theorem 1, $P \in GP$. Because all the traces in P are in $\mathcal{E}_u^\omega$, $P \subseteq \mathcal{E}_u^\omega$. Thus, $P \in GP \cap \{P \mid P \subseteq \mathcal{E}_u^\omega\}$.

($\supseteq$) Let P be a property in $GP \cap \{P \mid P \subseteq \mathcal{E}_u^\omega\}$. Theorem 1 shows how to construct an SMA M that enforces any property in GP, including P.

$\square$

4.1.3 Partially Controllable-Events Policies

Events are called partially controllable when they can be inserted but not suppressed, or suppressed but not inserted [14]. For example, a monitor may be able to drop but not inject events encrypted with a key that is unknown to the monitor. To encode partially controllable events, let (E_u, E_d, E_i, E_c) be a partition of E such that:

- E_u contains uncontrollable events,

- E_d contains events that can be suppressed but not inserted,

- E_i contains events that can be inserted but not suppressed,

- E_c contains events that can be fully controlled (i.e. suppressed *and* inserted).

In addition to the uncontrollable events requirement, the partially controllable events constraint allows events in E_d to be output only if they are input, and requires that events in E_i that are input are not suppressed.

Let $\mathcal{E}_{pc}^{\omega}$ be the set of all traces constructed as sequences of exchanges in which the partially controllable events constraint is satisfied, i.e.

$$\mathcal{E}_{pc} = \{\langle i, o \rangle \mid (i \in E_u \cup E_i \vee o \in E_u \cup E_d) \implies i = o\}.$$

Properties about traces with partially controllable events that are enforceable by SMAs are exactly the game properties that satisfy the partially controllable-events constraint.

Corollary 3. *Let PCP be the set of properties about traces with partially controllable events that are enforceable by SMAs. Then, $PCP = GP \cap \{P \mid P \subseteq \mathcal{E}_{pc}^{\omega}\}$.*

Proof. ($\subseteq$) Let M be an SMA that enforces a property $P \in PCP$ that satisfies the partially-controllable events constraint, i.e. $\mathcal{L}(M) \subseteq P$ and $\forall x \in P : x \in \mathcal{E}_{pc}^{\omega}$. By Theorem 1, $P \in GP$. Because all the traces in P are in $\mathcal{E}_{pc}^{\omega}$, $P \subseteq \mathcal{E}_{pc}^{\omega}$. Thus, $P \in GP \cap \{P \mid P \subseteq \mathcal{E}_{pc}^{\omega}\}$.

($\supseteq$) Let P be a property in $GP \cap \{P \mid P \subseteq \mathcal{E}_{pc}^{\omega}\}$. Theorem 1 shows how to construct an SMA M that enforces any property in GP, including P.

$\square$

4.2 SMAs that Monitor Batches of Concurrent Events

SMAs can be generalized to model monitors that oversee multiple streams of events generated by concurrently executing targets. The generalization is achieved by formalizing batches of events, which can be regarded as "global" events. This formalization enables modeling many practical implementations, e.g. network routers, intrusion detection systems, and system-on-chip interconnects, yet it assumes that concurrent events can be collected/distributed for enforcement.

A batch is a tuple of events that may be observed on a finite vector of monitorable streams. A stream is monitorable if it can be associated with a recursively enumerable set of events (that can be monitored by an SMA). The set of batches associated with a vector of streams is the Cartesian product of the events associated with each stream component. For example, given a vector of streams $s_1, s_2, ..., s_n$, the set of batches is $E_1 \times E_2 \times ... \times E_n$ such that E_i is the set of events associated with stream s_i (for all $i \in 1...n$). Because the vector of monitored streams is finite and the sets of events associated with each stream component are recursively enumerable, the set of batches is also recursively enumerable (and thus the vector of streams is monitorable).

Example 5 (Load-balancing SMA). *Let M_3 be an SMA that acts as a load-balancer by distributing requests from a stream s_x to two servers y and z that accept requests on streams s_y and s_z, respectively. The set of M_3's batches is $E = E_x \times E_y \times E_z$, such that $\epsilon \in E_x$ and $E_x \subseteq E_y \cap E_z$. M_3 is defined as (E, Q, q_0, δ), where $Q = \{y, z\}, q_0 = y$, and δ is defined as follows. Given a state $q \in \{y, z\}$ and an input batch $(e, e', e'') \in E$,*

$$\delta(q, (e, e', e'')) = \begin{cases} (q, (e, e', e'')) & \text{if } e = \epsilon \\ (z, (\epsilon, e, e'')) & \text{if } q = y, \\ (y, (\epsilon, e', e)) & \text{otherwise} \end{cases}$$

In words, M_3 does not intervene when input events on stream s_x are nil. Else, M_3 takes non-nil inputs from stream s_x and outputs them on streams s_y or s_z, alternating the output stream on each intervention (and removing the events from stream s_x).

4.3   Nondeterministic SMAs

SMAs can be generalized to allow nondeterminism by changing an SMA's transition function to a relation. Nondeterminism may be useful for describing monitoring algorithms more abstractly, e.g. to avoid cluttering algorithm descriptions or when implementation details are unavailable.

Definition 6. *A nondeterministic Stream-Monitoring Automaton (NSMA) N is a tuple (E, Q, I, δ), where E is a recursively enumerable set of events such that $\epsilon \in E$, Q is a recursively enumerable set of automaton states, $I \subseteq Q$ is a set of initial states, and $\delta \subseteq Q \times E \times Q \times E$ is a computable and total transition relation.*

Interestingly, game properties also characterize the set of properties that are enforceable by NSMAs.

Theorem 2. *A property P is soundly enforceable by an NSMA iff P is game.*

Proof. Let N be an NSMA that soundly enforces some property P. Since N's transition relation is total and computable, it is defined for all possible inputs and it will always output some event (possibly ϵ). Let $f : E^* \to E$ be a function that takes any finite sequence of

inputs and runs N with those inputs to compute a valid output. Because the traces that form by starting from the empty sequence and concatenating every pair $\langle i_0, f(i_0) \rangle, \langle i_1, f(i_0 i_1) \rangle...$ ad infinitum satisfy P (because $\mathcal{L}(N) \subseteq P$), $\mathcal{L}(N)$ is a game property. Further, because $\mathcal{L}(N) \subseteq P$, by Proposition 1, P is a game property, i.e. $P \in GP$.

Theorem 1 shows how to construct an SMA M that enforces any property in GP, including P. If an SMA can be constructed to enforce P then so can an NSMA. □

Chapter 5: Simulations

This chapter shows that SMAs can simulate other previously defined security automata. Simulations are important because they demonstrate the generality of the SMA model. For simplicity we only compare deterministic versions of SMAs and other security automata. As is common in other models that represent traces as infinite sequences (e.g. [15]), finite traces are encoded as infinite sequences that end in stuttering events *ad infinitum* (here, the nil event).

5.1 Truncation Automata

A truncation automaton [8, 9] is a tuple (Q, q_0, I, γ) where Q is a countable set of states, q_0 is an initial state, I is a countable set of input symbols, and $\gamma : (Q \times I) \to Q$ a transition function. If $\gamma(q, i)$ is ever undefined (for any $q \in Q$ and $i \in I$), the input is rejected (in the SMA model, a ϵ output); otherwise the input is accepted (outputted verbatim). The following simulation assumes that the input symbols and states of truncation automata are recursively enumerable sets.

Let $I(x)$ ($O(x)$) denote the sequence of inputs (outputs) in a sequence of exchanges $x \in \mathcal{E}^\infty$ and the ϵ-stuttering equivalence operator $\stackrel{\epsilon}{=}$ defined as follows. $\forall v \in E^\omega : \forall v' \in E^\infty : v \stackrel{\epsilon}{=} v' \iff v = v' \ \vee \ v = v' \epsilon\epsilon\epsilon....$

Theorem 3. *For every truncation automaton A, there exists an SMA M such that for every input sequence σ that is accepted by A, there exists an execution $x \in \mathcal{L}(M)$ with $I(x) = \sigma...$ (where $\sigma...$ is an input sequence with the σ prefix) and $O(x) \stackrel{\epsilon}{=} \sigma$.*

Proof. Given a truncation automaton $A = (Q, q_0, I, \gamma)$, consider an SMA $M = (E, Q, q_0, \delta)$ such that $E = I \cup \{\epsilon\}$ and δ is defined as follows. If $\gamma(q, i) = q'$ is defined (for some states $q, q' \in Q$ and input $i \in I$), then define $\delta(q, i) = (q', i)$. Otherwise, $\delta(q, i) = (q, \epsilon)$. Thus, for every execution σ that A accepts, there exists an execution $x \in \mathcal{L}(M)$ with $I(x) = \sigma..., O(x) \stackrel{\epsilon}{=} \sigma$. $\qquad\square$

5.2 Edit Automata

An edit automaton [9] is a tuple (Q, q_0, γ) defined with respect to a set of actions $\mathcal{A}$, where Q is a set of states, q_0 an initial state, and $\gamma : A \times Q \to Q$ is a transition function that either inserts an action or suppresses the input[4]. The single-step semantics of an execution has the form $(\sigma, q) \xrightarrow{\alpha}_{EA} (\sigma', q')$ where σ denotes the sequence of actions that the target program wants to execute; q denotes the current state of the edit automaton; σ' and q' denote the action sequence and state after the automaton takes a single step; and α denotes either an action produced by the automaton or no output.

Theorem 4. *For every edit-automaton A defined with respect to a set of actions $\mathcal{A}$, there exists an SMA M such that for every execution σ, if A transforms σ to τ, then there exists an execution $x \in \mathcal{L}(M)$ with $I(x) \stackrel{\epsilon}{=} \sigma$ and $O(x) \stackrel{\epsilon}{=} \tau$.*

Proof. Given an edit-automaton $A = (Q, q_0, \gamma)$ defined with respect to a set of actions $\mathcal{A}$, consider an SMA $M = (E, Q', (\tau_e, q_0), \delta)$, where $E = \mathcal{A}, Q' = \mathcal{A}^* \times Q$, τ_e is the initial sequence of actions pending to be executed, and δ is defined as follows.

$$\delta((x, q), a) = \begin{cases} ((xa, q'), a') & \text{if } (xa, q) \xrightarrow{a'}_{EA} (xa, q'), \\ ((x, q'), \epsilon) & \text{if } (xa, q) \xrightarrow{\cdot}_{EA} (x, q'). \end{cases}$$

---

[4]The original definition of edit automata also includes operational rules for single-action acceptance and insertion of multiple actions in one step. These additional features are omitted here because they can be simulated by single-action insertion and suppression [9].

In words, δ either outputs event (action) a' if there is a step in A that inserts a', or outputs ϵ if there is a step in A that suppresses input a. Thus, for every execution σ that A transforms to τ, there exists an execution $x \in \mathcal{L}(M)$ with $I(x) \stackrel{\epsilon}{=} \sigma$ and $O(x) \stackrel{\epsilon}{=} \tau$. $\qquad\square$

5.3 Mandatory-Results Automata

Mandatory Results Automata [16] (MRAs) mediate events between a target application and an executing system. The target application sends actions to the executing system and the executing system sends results for those actions back to the target application. Let $\mathcal{A}$ and $\mathcal{R}$ denote the set of actions and responses that are mediated by an MRA, respectively. An MRA is a tuple (E, Q, q_0, γ) where $E = \mathcal{A} \cup \mathcal{R}$ is the event set over which the MRA operates, Q is a computably enumerable set of automaton states, q_0 is an initial state, and $\gamma : Q \times E \to Q \times E$ is a transition function, which takes the current state and input event and returns the next state and output event. SMAs can simulate MRAs by producing the same actions and responses.

Theorem 5. *For every MRA A, there exists an SMA M such that for every trace $x \in \mathcal{L}(A)$, there exists a trace $x' \in \mathcal{L}(M)$ with $I(x) \stackrel{\epsilon}{=} I(x')$ and $O(x) \stackrel{\epsilon}{=} O(x')$.*

Proof. Given an MRA $A = (E, Q, q_0, \gamma)$ interposed between an application and an executing system, consider an SMA $M = (E, Q, q_0, \delta)$, where δ is defined as follows.

$$\delta(q, e) = \begin{cases} (q', e') & \text{if } \gamma(q, e) = (q', e'), \\ (q, \epsilon) & \text{if } \gamma(q, e) \text{ is undefined.} \end{cases}$$

Observe that δ outputs the same state and event as γ for all the inputs that are defined in γ, and outputs ϵ for all the inputs that are not defined in γ. Thus, for every trace $x \in \mathcal{L}(A)$, there exists a trace $x' \in \mathcal{L}(M)$ with $I(x) \stackrel{\epsilon}{=} I(x')$ and $O(x) \stackrel{\epsilon}{=} O(x')$. $\qquad\square$

5.4 Input-Output Automata

Input-output (I/O) automata [44] have been used to model concurrent systems, and more recently runtime enforcement mechanisms [18]. An I/O automaton A is a tuple $(\mathcal{A}, Q, q_0, \gamma, R)$, where each of its components is defined as follows. $\mathcal{A}$ is an action signature defined as a triple (I, O, Int) of disjoint sets denoting input, output, and internal actions respectively; Q is a possibly infinite set of states; q_0 is an initial state; $\gamma \subseteq Q \times \mathcal{A} \times Q$ is a transition function with the *input-enabledness* property, i.e. for every state q and input action a there is a transition $(q, a, q') \in \delta$; and R is an equivalence relation partitioning the set $O \cup Int$ into a countable number of equivalence classes. The set $O \cup Int$ specifies the actions that are locally controllable by the automaton, and the relation R is used to specify fairness constraints for the automaton's executions.

For every I/O automaton A having some internal actions, a new I/O automaton A' can be defined that has no internal actions and outputs the same traces as A. Because I/O automata that have no internal actions are as expressive as I/O automata that have internal actions, only I/O automata with external actions are considered for simulation. Thus, an I/O automaton's trace is a sequence of input and output actions. Let $x_A = a_0; a_1; \ldots$ be a trace produced by an I/O automaton A, and $\langle i_0, o_0 \rangle \langle i_1, o_1 \rangle \ldots$ be a trace produced by an SMA M. Then,

$$a_0; a_1; \ldots \cong \langle i_0, o_0 \rangle \langle i_1, o_1 \rangle \ldots \iff \forall j \in \mathbb{N} : \langle i_j, o_j \rangle = \begin{cases} \langle a_j, \epsilon \rangle, & \text{if } a_j \in I \\ \langle \epsilon, a_j \rangle, & \text{if } a_j \in O \end{cases}$$

Trace equivalence is extended to sets of traces, i.e.

$$X \cong Y \iff \forall x \in X : \exists y \in Y : x \cong y.$$

Theorem 6. *For every I/O automaton A, there exists an SMA M such that $\mathcal{L}(A) \cong \mathcal{L}(M)$.*

Proof. Let $A = (\mathcal{A}, Q, q_0, \gamma, R)$ be an I/O automaton (with only external actions), consider an SMA $M = (E, Q, q_0, \delta)$, where $E = I \cup O$ and δ is defined as follows. For every transition $(s, a, t) \in \gamma$, if a is an input action, add (s, a, t, ϵ) to δ. Otherwise, add (s, ϵ, t, a). The traces produced by M will be equivalent to the traces produced by A. Therefore, $\mathcal{L}(A) \cong \mathcal{L}(M)$. $\square$

Chapter 6: Related Works on Ethnographic Studies of Secure Software Development

It has long been recognized that human factors play a dominant role in the ever-present software vulnerabilities, and substantial research has been devoted to this area [19, 20, 21, 22, 23, 24, 25, 26]. Past efforts have used a variety of research methods including surveys/interviews, controlled experiments, studying code artifacts, and analyzing data collected from secure-coding competitions. It is also understood that there is a fundamental economic problem underlying software insecurity [27], and in general there appears to be unwillingness in industry to put code security at equal importance to other business considerations, such as time to market and richness of features. Next, each of the related works are reviewed and compared to the ethographic study presented in the following chapters of this dissertation.

Assal and Chiasson [19] performed an online survey to explore the interplay between developers and software security processes. Their research found that developers were motivated to develop secure code, but were often hindered by a mismanaged organizational process. The authors advocated looking beyond developers and examining broader organizational factors that may impact the security of the developed software. The present study is one such attempt, through an extensive ethnographic study in a software company. Many of our findings confirm the analysis results from Assal and Chiasson's work. The present work also reveals some deeper insights into the reasons behind software (in)security, as well as a co-creation model to address them.

Ruef, et al. [20] and Votipka, et al. [21] conducted a series of studies based on data collected from the Build It, Break It, Fix It (BIBIFI) contests. A number of patterns of developer mistakes leading to vulnerabilities were analyzed. The present work examines software development process in a real company. This in-depth ethnographic study is complementary to the analysis based on large-scale competition data. One possible cross over between the two types of studies is that the insights from one can be used to drive the analysis in the other. For example, an observed real-world phenomenon that has significant security impact could be replicated in the BIBIFI contest to further examine a hypothesis on a much larger and more diverse population.

Oorschot and Wurster [22] posited that developers have different skills which often do not include security and suggest that the focus should be on those who design APIs, because it is unrealistic to expect all developers be taught sufficient security. The present study poses a similar question that emerges from the ethnographic data, regarding how much security knowledge developers can realistically master, and whether a co-creation model where security experts and developers closely collaborate would be a more effective approach.

Green and Smith [23] discussed that developers are not the problem for insecure code. The focus should be on creating more developer-friendly and developer-centric approaches and supporting them when they are dealing with the security tasks. The ethnographic data in the present study supports this conclusion. Moreover, the fieldwork also resulted in a co-creation model that achieved the goal of providing the needed support to developers for writing more secure code.

In addition to the works mentioned above, the research community has explored this area through a number of angles. Oliveira et al. [24] conducted surveys to understand developers' attitudes toward security which leads to understanding that APIs and tools can be improved significantly. Votipka et al. [25] performed semi-structured interviews to

compare how hackers and testers find vulnerabilities. Stransky et al. [26] designed an online platform to conduct online secure-programming studies with remote developer participants.

To the best of our knowledge, the present ethographic study is the first work that uses participant observation and long-term ethnography to study secure software development in a real company. The data and findings presented here serve to complement the efforts discussed above, and often times reveal deep insights not obtainable through other approaches.

Chapter 7: Discovering and Fixing Security Issues in Practice

This chapter describes the discoveries of two security researchers who were embedded in a company as software developers for almost two years (23 months) of total research time. The chapter starts by describing the study's methodology, research ethics, and the company's software development process. It continues by discussing with the recount of two types of security issues found: code injection vulnerabilities and authentication flaws.

7.1   Methods

The main method utilized in the research was participant observation. This method was developed by anthropologists and sociologists as an effective way to study human behaviors and cultures through participating in daily activities and observing people's behaviors through long-term study (typically more than a year). These activities help researchers obtain a solid understanding of a particular culture and gain insights into subjects' activities, knowledge, and habits.

In this research, the participant observation method was adapted to work within a software security company. The participant observers (POs) were two computer science Ph.D. students, each of whom underwent systematic training in qualitative research method under the guidance of an anthropologist. In addition, the research team included computer science professors specialized in programming languages and security. Hereafter, the terms R1 and R2 are used whenever necessary to differentiate the contributions of each PO, where R1 is the author of this dissertation.

The particular methodology adopted in this study had several advantages.

- The POs were experienced programmers and knowledgeable in security, which enabled them to quickly get immersed into the company's software development process and gain the trust of other developers.

- POs were actively involved in the software development process, which enabled them to closely observe software (in)security phenomena from within the context where it developed.

- POs not only acted as passive observers but also as advocates of software security inside the company. This approach enabled the team to observe how the various stakeholders reacted to discoveries of security vulnerabilities, providing valuable insights into why those vulnerabilities were introduced in the first place and the constraints under which they had to be fixed (or not).

- Having access to company's internal records enabled the POs to observe not only contemporary events as they unfolded, but also the historical evolution of problems that were archived on code repositories, ticketing systems, and other registries.

- Participant observation is often a solo affair in the social sciences; having two embedded POs permitted the examination of the company from two different but complementary perspectives. The POs were assigned different tasks, had slightly different hours at the company, and developed relationships with company personnel at different points of time.

- The research team consisted of experts in engineering and social science. This multidisciplinary team participated with the embedded researchers in developing the analysis over months, permitting the identification of themes and ideas that crosscut disciplines and had both theoretical and applied dimensions. This team-based approach to both

data collection and analysis is a significant contribution to how this type of research can be done effectively.

Each PO worked in the company 20 hours a week spread on three week days. R1 worked for 17 months and R2 for 6 months. In general, the POs' tasks included debugging existing implementations to find bugs' root causes, writing code fixes or implementing new features, performing code reviews, and software quality assurance. The POs took field notes about their observations, including both security issues found in the software and everyday interactions with developers and other employees involved in the development process. Notes had two forms: descriptive and insightful. Descriptive notes were intended to be as informative as possible, avoiding personal judgments or opinions. Insightful notes aimed to capture "ah-ha" moments and provide reflective analysis of the situations experienced by the observers.

To derive research insights into the raw notes, the general inductive approach [59] was applied, augmented by specific techniques for qualitative data analysis [60]. The initial step was to find patterns that emerged directly from the data themselves. This process happened via weekly meetings of the entire research team, where comparisons could be made across researchers, discussions could address both the human and technical dimensions of software development in a company, and plans made for further exploration of interesting topics. Identifying themes and links between ideas proved central to the inductive analysis, as well as developing contextual analysis around key examples. Data analysis continued through the coding of field notes based on identified themes. These codes included themes related to software security, human elements of the work, important explanatory concepts that emerged during the research, and data linked to the key examples. Research meetings then shifted to further developing a joint understanding of the data and identifying ways to explain the observed patterns, as well as potential solutions to how human and technical factors combined to shape (in)security.

7.2 Research Ethics

In this research, the employees of the company (developers, support techs, and managers) were considered human subjects. The study was reviewed and approved by the Institutional Review Board (IRB). The approval letter is shown in Appendix B. Researchers explained the study goals to participants and obtained verbal informed consent from participants. Field notes were anonymized, as well as discussions during weekly research meetings. This presentation follows that same anonymization approach. Accordingly, the term *application under study (AUS)* is used hereafter to refer to an application in the company's product suite.

One ethical dilemma that emerged during the research was what to do when security vulnerabilities were discovered. Given ethical standards among cybersecurity professionals, we made the decision to present these discoveries to the software development team. This process proved crucial to the further development of the research. Rather than simply observing what happened while continuing to work at the company, the researchers raised these security concerns, and where directed, actively worked on addressing them. This active engagement led the research team to a co-creation model, where research, programming, and security were all ongoing parts of what happened during the fieldwork.

7.3 Development Process

The development team held a scrum meeting every morning that lasted 15-30 minutes. In this meeting, each developer briefly commented about any progress accomplished or roadblocks encountered the day before and discussed the plan-of-work for the current day. This was an opportunity for developers and managers to give and receive feedback from each other.

Work was organized, prioritized, assigned, and tracked using ticketing and code management systems. In general, tickets were generated by developers, support techs, or customer-facing specialists, ranked in prioritization meetings held by development managers (including a dev team lead), and assigned and tracked by the dev team lead. After implementation, tasks were moved into the peer-review stage in which at least two other developers reviewed any code changes, added pending tasks if necessary, and finally approved merge requests. After code changes were approved by all reviewers, tickets were reassigned for quality assurance and integration testing, which was often done by both developers and support/customer-facing specialists. When all tests had been passed, tickets were marked as "done" and merged into the code repository's development branch. When the set of target features for a release had been implemented, the team lead created a release candidate branch. Every release candidate was tested in-house one last time before being finally moved into release and installed on customer environments.

7.4   Code Injection Vulnerabilities

This section describes several code injection vulnerabilities discovered by pen-testing specific areas of the product suite, and the reactions of developers after being exposed to such vulnerabilities.

7.4.1   Specific Methods Adopted in this Study

Penetration testing, also known as pen-testing or ethical hacking, is an authorized simulated cyber-attack process against a computer system to reveal security flaws and holes. The goal is to identify weaknesses which might provide a passage for unauthorized users to gain access and alter the integrity of the computer system. Pen-testing efforts were focused on code injection because it is one of the most common vulnerabilities found on web

applications [61]. A code injection attack allows attackers to inject malicious codes into a computer system and change the course of execution.

7.4.2 Vulnerabilities Found

Two types of code injection vulnerabilities were found during the pen-testing process: cross-site scripting (XSS) and shellcode injection via file upload.

– To exploit XSS vulnerabilities, attackers typically use the input fields of web applications to insert malicious scripts that are then executed by the browser on other users' sessions. These malicious scripts may read sensitive information (e.g. passwords or session tokens) and transmit it to the attacker.

– Shellcode injection via file upload allows attackers to inject malicious code into the web application's back-end server by uploading a crafted file. When the file is executed by the server (e.g. to display the contents of that file), the malicious script opens a port on which the attacker can connect and obtain a system shell.

7.4.2.1 XSS Injections

On the first day of pen-testing, R2 found an XSS vulnerability on a third-party component of the AUS. A potential XSS attack was demonstrated to the developers, who initially showed some interest in the finding and blamed the third-party application developers. One participant said:

> "This vulnerability belongs to our third-party application, and we did not develop this part. It is better to upgrade the software and see if we will still have the issue,"

They also mentioned that it would be more interesting if researchers could find any vulnerability in the company's code.

Within a few minutes, R1 found several other areas in the AUS that had the same XSS problems. This time, the code was written in-house and the developers were more concerned about the new findings. R1 documented both vulnerabilities on their ticketing system.

Remediation of the XSS injection was split into two separate development tickets that were assigned to R2. R2 fixed the XSS in the AUS by writing new library functions to add the missing input validation. R2 spent a few days investigating the XSS in the third-party component. Fixing just the XSS didn't seem like a good solution because the component was no longer being maintained and contained other vulnerabilities. Updating the component to its latest version would break other parts of the software that the developers seemed to be unfamiliar with, and neither the component nor the existing AUS were documented, so figuring out a solution would probably require some time. A few days later, R2 was told,

"We have some other issues that need more immediate attention. We might have to put that project on hold and move on to something that is more urgent."

7.4.2.2 Shellcode Injection

On the same day, R2 found another vulnerability on the AUS; this time it was a shellcode injection. The vulnerability allowed attackers to inject their customized shellcode into a valid file input and upload it to the server to get backdoor access with a privileged user account on the server. The attacker must be someone who already had a regular account inside the AUS web application. The finding was interesting to the researchers and the developers for different reasons. For the research team, this was a critical vulnerability—clients should never have escalated access to the server. They should only be able to perform some limited commands on the operating system such as changing the network IP address. This essentially allowed their customers to jail-break out of the sandbox set up on the server.

Developers were interested in understanding how access had been gained. They asked how the researcher was able to gain access, and what was obtained user privilege on the server. They also mentioned that they didn't believe to have any important information on that server.

They quickly realized that they had hardcoded credentials stored on the server. These credentials would allow a customer who successfully exploited the shellcode injection vulnerability to see other customers' information. Faced with this fact, the developers indicated that this vulnerability was bigger than what they thought, and that they should take imminent action on it. However, that did not come to fruition at the next group meeting, where they continued to talk about these vulnerabilities but took no action. They agreed that an attack on this vulnerability could have some serious impact, but they downplayed the possibility of being attacked. One participant said,

"It's unlikely that someone will find this vulnerability."

At the same time, another participant said that the team should focus on developing new features. The participant added,

"If we tried to fix every bug in our system, we would quickly go out of business."

7.5 Authentication Flaws

This section describes several instances of a security flaw (or feature) found by R1 in an application dedicated to providing network access control (henceforth the AUS). The AUS was designed to be configured by network operators enforcing access-control policies on end users of the networks they managed. The feature, called here *silently allow failed authentication (SAFA)*, enabled untrusted end users to quietly bypass authentication and access protected network resources without authorization. In contrast to the code injection vulnerabilities described earlier, the code that enabled SAFA was originally introduced

(intentionally) as a feature, and its usage had evolved over time to cover different failure scenarios. Studying the origin and evolving usage of SAFA provided a lens through which to observe a number of dilemmas the developers and company had to deal with.

SAFA is described next, followed by the specific methods adopted for its study, and four scenarios in which SAFA was used.

7.5.1 Silently Allow Failed Authentication (SAFA)

In the AUS, the process of authenticating users and devices into the network involved assigning an authentication state to every authentication attempt and subsequent authentication queries. Authentication queries were self-triggered by the AUS and configurable, i.e., AUS operators could specify how frequently users and devices must authenticate and the AUS executed the corresponding assessments by querying authentication servers, the operating system on the client's device, or sometimes prompting the user directly through a web browser. The authentication states were: pass, fail, unreachable server, unknown username, and a *silently-allow (SA)* state. Once an authentication flow entered the SA state, it acquired an SA role, which was associated with a built-in policy that granted full access to the protected network. From a security perspective, SA was a dangerous authentication bypass mode, providing access without full authentication (hence, silently-allow). Specifically, an authentication attempt assigned to an SA state was treated as successful and granted full access to the network that the AUS was expected to protect. Further, the SA state was relatively persistent. Once a device acquired an SA role, it remained in this state until an administrator manually added policies that would forcefully change the role assigned to the device.

7.5.2 Specific Methods Adopted in this Study

SAFA was discovered by R1 by inspection of code repositories, ticketing systems, internal wiki articles, and user-facing documentation. After the initial SAFA instance was discovered, R1 investigated all other parts of code that could lead to the SA state. To do this, he first searched for specific terms in the code repositories and ticketing systems. Because the AUS code structure was distributed among many projects, this task was non-trivial and time consuming. In the ticketing system, all instances were located where comments were made about SAFA. Developers rarely used SAFA-related terms in the tickets' descriptions, so uncovering the different instances of SAFA required R1 to correlate ticket information with the implementation code on the different project repositories. To further understand the different ways in which SAFA could be used, R1 formalized the authentication flows and set up a lab for testing the different scenarios. These steps led to the discovery of other SAFA instances not mentioned in tickets.

7.5.3 Observed Instances of SAFA

The AUS had five configurable authentication back-ends: three databases (one legacy implementation, a second one for network guests, and a third one for admin user accounts) and two integration components (one connecting to LDAP servers, and a second connecting to SAML identity providers). The SA state could be triggered in each of these configurations, under one of the following scenarios: 1) the AUS failed to communicate with some authentication server, 2) the AUS was misconfigured, 3) an unexpected runtime exception occurred, or 4) a back-end implementation was lacking.

7.5.3.1   SAFA 1: Broken Integration

Broken integration with a customer's backend authentication server was originally the only intended enabler of SAFA. In one of the authentication flows, the AUS communicated with Active Directory (AD) servers. The connection parameters must be pre-configured by network operators and stored in the AUS's internal database. At runtime, these parameters were pulled from the database and LDAP search queries were executed.

If the network connection to the AD server failed (e.g., due to misconfiguration, a physical link failure, DNS down, or time outs due to request overloads), no domain entries were found in the AD server, or any other runtime exception occured (e.g., invalid credentials or duplicate entries in the AUS database), then the device was assigned the SA role and obtained full access to the network.

Most developers and support techs were aware of this SAFA instance and often framed it as a feature that alleviated the configuration burden for network operators. It was often justified by explaining that network operators were more interested in not blocking legitimate users into the network than in protecting their networks from intruders with stricter policies that could affect network usability. For instance, two support techs commented that there had been scenarios in which customers became very frustrated because policy changes had forced most users out of the network and required them to reauthenticate. According to study participants, network operators would see these scenarios as "network outages" that would cause a high load of help-desk support calls for them, which were not well-received.

It remains unclear how many customers adhered to this instance of SAFA. Written documentation suggested that some of the developers' rationalization of SAFA was missing some important details. A support ticket generated from a customer call dated four years before the study showed that SAFA was not always well-received by customers. This ticket stated:

"Customer doesn't want failed authentication attempts due to LDAP errors to fall into SAFA but to try another server."

Further, there was scarce evidence on customer-facing documentation to support the claim that customers were well aware of SAFA authentication. SAFA was only mentioned as a footnote in the release notes of a version of the product that had been released five years before the study. The footnote read:

"A system failure will not cause a network-wide outage and will silently-allow authentication for existing and new users attempting network access."

This single appearance of SAFA in customer-facing documentation suggested that SAFA was not well-known and understood by customers. When R1 raised the concern that some customers may not be fully aware of SAFA, the participants explained that customers would be informed if they asked about the SA role (which was visible on a secondary page on the UI). Since this SAFA instance was not seen as a security vulnerability but as a feature that benefited most users, no code changes were made to remediate the issue.

7.5.3.2   SAFA 2: Misconfiguration

The second and most evident way in which SAFA would give full access to the network was when the AUS was misconfigured. Specifically, a dropdown menu in the UI for policy creation in the administrative portal allowed operators to select an SQL authentication option that mapped to a non-functional legacy database. This would result in assigning an SA state to all authentication attempts, regardless of which credentials were entered by users. Developers explained:

"This authentication method is probably broken. I believe it has been deprecated a long time ago."

When asked about why the UI was still showing this option, they said,

> *"I don't know why but it should not be there."*

We were unable to find any written documentation other than the code to confirm if and when this authentication method had been officially deprecated. An internal testing ticket from three years back suggested that this authentication method had already been deprecated, but some customer-facing documentation still listed SQL authentication as a possible authentication method. The issue was documented by R1 and eventually prioritized for development.

7.5.3.3   SAFA 3: Unexpected Runtime Errors Due to Implementation Bugs

SAFA could also be triggered by unexpected runtime errors. Several instances of authentication code were surrounded in try-catch blocks that would catch SQL and other runtime exceptions and set authentication state directly to SA. SQL exceptions were somewhat common across the AUS and could cause it to halt operation. Some SQL exceptions occurred after upgrades that resulted in tables with missing attributes, or because the AUS had incorrect database permissions. Other runtime exceptions included null pointer exceptions and out-of-bound array access.

Like the other cases, this instance of SAFA was an intentional choice. The code that implemented SQL server authentication was added more than 15 years ago and since then had been revisited a few times. Although not explicitly stated, it is possible that this SAFA instance was an ad-hoc solution for dealing with code complexity and legacy implementations, allowing the AUS to continue execution and hide any incomprehensible bugs. The code preceding the catch blocks looked complicated (with several sections commented out). From reading the code, it was hard to tell what code paths could be executed in each scenario.

7.5.3.4 SAFA 4: Unimplemented Protocol Flows

Perhaps the most critical SAFA instance was an unimplemented SAML authentication flow that allowed users directly into the network without even checking the credentials against the configured SAML identity provider. Unlike previous SAFA instances which were somewhat acknowledged by developers, developers explained that they were unaware of this SAFA instance and blamed previous developers who no longer worked at the company for the wrongdoing. When asked about the impact of the problem, some developers downplayed its potential negative effects by suggesting that

"Very few customers are probably using SAML authentication on their networks."

Yet, in a separate conversation, a support tech disputed the developers' claim by saying that they knew of at least ten customers who were using SAML for authentication. This SAFA instance was fixed by R1 who implemented the missing SAML authentication flow.

Chapter 8: Understanding Developers' Attitudes towards (In)Security

This chapter describes the emerging patterns and themes that resulted from R1's analysis of his field notes. Based on these themes, the developers' reactions to the security issues found by the researchers are explained, and several explanations of their sometimes inconsistent narratives are conjectured. The chapter ends with a discussion about how security experts' interventions can become more fruitful.

8.1   Emerging Patterns from Field Notes

Notes were collected from daily interactions with developers. Each note was treated as a data unit and contained records of relevant discussions held during meetings and informal conversations. To derive research insights into the raw notes, the general inductive approach [59] was applied, augmented by specific techniques for qualitative data analysis [60].

The core meanings of observations was first captured by assigning codes to raw notes, the inductive component. The coding process was carried out through multiple readings and interpretations of notes. Not all notes were assigned codes, and some notes were assigned more than one code. Codes were re-evaluated throughout many iterations. At each iteration, codes were added, merged, or removed to reduce overlap and redundancy. Codes were then grouped into themes which identified patterns that were most relevant to the research objective.

The data gathering and coding process happened in parallel with the weekly research meetings held with the entire research team. During these meetings, insights were shared across researchers, human and technical dimensions of software development were discussed,
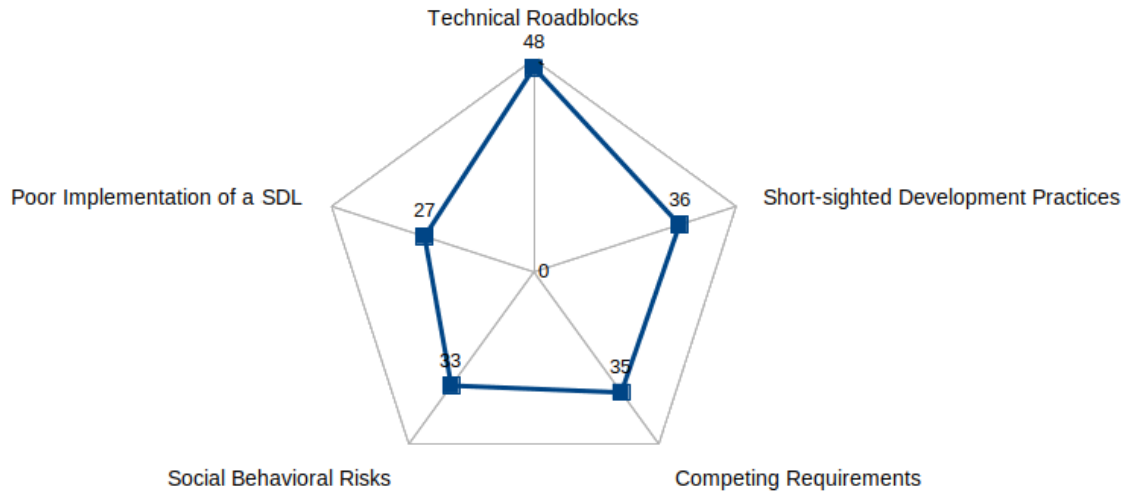
Figure 8.1: Secure Software Development Threats. The distance of each point from the center represents the number of notes that were tagged with codes grouped by the respective theme labeling each axis.

and plans were made for further exploration of interesting topics. These insights were incorporated into the coding process and helped come up with new codes, themes, and hypotheses that were then tested in the field. Identifying themes and links between ideas proved central to the inductive analysis, as well as developing contextual analysis around key examples.

The common threats (i.e. themes) that emerged from the codes are shown in Figure 8.1. The center represents the ultimate goal, i.e. development of secure software and each axis represents a threat. The distance of each point from the center represents the number of notes that were tagged with codes grouped by the respective threat. For example, there were 48 notes tagged with codes that were grouped in the threat *Technical Roadblocks*. The meanings of each threat are explained next.

8.1.1 Technical Roadblocks

This category includes mostly notes about conversations in which developers described the time they spent debugging existing implementations, understanding legacy code, and configuring systems. It also includes observations about the problems that developers faced when trying to bring dependent libraries and third-party components up to date.

The time spent on these tasks was significant and directly impacted their ability to focus on developing secure software. Understanding the root cause of problems in customer deployments was challenging because developers were often operating with incomplete information. Managing updates was also challenging because of the complex relationships among the many components of their software and the dependencies on third-party libraries. For this reason, developers were often hesitant to make code modifications or update libraries if the changes were not strictly required.

8.1.2 Poor Implementation of a Secure Development Lifecycle

This category includes codes that expose deficits in their implementation of secure software development practices. Software development practices that are good for security usually include security training, documentation, testing, risk analysis, and development of an incident response plan [62, 63].

The development team's security awareness evolved through the study. In the beginning, there were not many conversations about security or security practices in place. Over time, as researchers demonstrated security issues and actively evangelized about security, the developers started to become more security-aware and added some security to their development process. For example, toward the end of the study developers incorporated security discussions into their design meetings. Still, the researchers observed that in general, de-

velopers were not trained to think as attackers, and sometimes missed important security considerations.

8.1.3 Competing Customer Requirements and Conflicting Business Strategy

This category includes observations about developers' efforts to meet requirements that often went against the development of secure software. There were two main sub-themes in this category: the business need to remain profitable, and customers' requirements to improve features like usability that were sometimes hard to achieve without compromising security. For example, occasionally developers were required to work on certain features because they were necessary to close certain deals with customers.

Customers' requirements were ever changing and seldom demanded security improvements. Sometimes developers would provide minimal implementations of certain features to satisfy these requirements. These solutions were often put in place as temporary, but ended up remaining in the code because future requirements prevented developers from ever having the time to revisit the original implementations. Other times legacy features were not removed because some important customers were still using them, even if those features were no longer being maintained and could contain security bugs.

8.1.4 Short-sighted Development Practices

A common theme that emerged from observations was that in general developers didn't spend much time planning, documenting, or testing. The development culture was focused on getting rapid-prototypes of products available to customers, and then keep building products ad-hoc. This highly pragmatic approach was sometimes detrimental for security, as there was no time for designing features with security in mind or performing security testing. These practices are called "short-sighted" because sometimes developers would reflect on past experiences and realize that they could have benefited if they had looked further into

the future and acted with more careful consideration before releasing new products and features to customers.

8.1.5 Social Behavioral Risks

This category includes observations about situations in which developers expressed ideas that did not correspond with their actions. For example, some developers would claim that testing was an important part of the development process, but then would write trivial tests designed only to pass an automated tool's minimum code coverage requirements. Similarly, they sometimes had a positive bias, i.e. they believed that their software was unlikely to have vulnerabilities or be attacked. Other observations in this category captured how frictions across organizational teams occasionally jeopardized new initiatives to improve the software or the development process, and participants' difficulties to be self accountable for their past mistakes.

8.2 Explaining Developers' Attitudes to Security Issues

This section describes R1's attempt to interpret the developers' attitudes towards the security issues described in Chapter 7. This interpretation is based on a correlation of the documented evidence collected from the different sources within the company (code repositories, ticketing systems, internal wikis, and customer-facing documentation) and the threats that emerged from analyzing the field notes.

8.2.1 Reactions to Code Injection Vulnerabilities

When developers were faced with the code injection attacks demonstrated by the researchers, they were surprised and expressed interest in learning how these attacks worked and how their application and customers could be impacted. Yet, their follow up actions diverged.

While the XSS in the AUS was fixed immediately, the XSS in the third-party component was backlogged and never addressed again (at least until the time of this writing). This difference in the approach seemed to be a consequence of a problem that came up quite often during developers' discussions, i.e. the challenge of keeping dependencies up to date in all of their projects. A participant once explained that updating third-party components was sometimes not cost-effective because the costs to update (measured in development efforts) outweighed the benefits of the update. Waiting too long was often also a problem because the component could stop being maintained, which could increase its risk of having broken functionality and security vulnerabilities. Thus, the difficulty of the issue relied on finding the right time to update, a problem for which developers had not yet found an effective solution.

Like the third-party XSS, the shellcode injection was not fixed either. Unlike the third-party XSS, the code that enabled the shellcode injection was written by the developers. The developers agreed with the researchers that the impact of an attack could have very serious consequences. Yet, they believed that an attack was highly unlikely. Developers perceived this attack as difficult to achieve. Because the attack model assumed the attacker was an administrator user (usually a network administrator), they mentioned that it was unlikely that these users possessed the skills necessary to perform the attack. Further, they explained that because they were shipping the machine to their customers, there were potentially other ways to break into the system, and thus fixing the current issue would not have that much value after all. Finally, they considered that fixing this issue would require a significant amount of work, presumably because the fix could impact many customers in the field.

Developers often faced similar dilemmas, i.e. whether to release software patches based on their potential impacts on existing deployments. Because the company had a small team and limited resources, they sometimes chose to postpone (and sometimes even never release)

certain features or fixes to prevent problems that could arise after the software had already been released.

8.2.2 Reactions to SAFA

According to members of the support team, SAFA was an "effective" solution to reduce customers' frustration for the number of help-desk calls that customers received when end-users experienced network interruptions. SAFA addressed customers' requirements for a product that would deliver an easy-to-use and frustration-free experience for both network operators and end-users. However, in general developers were somewhat hesitant to talk about SAFA, possibly because they did not understand how that part of the software worked or it was code that they were not proud of.

Attempts to explain the security implications of SAFA to the developers and make them fixed was not a straightforward process. A common perception is that security vulnerabilities are introduced in code because developers are not aware of the security issues involved, and explicit exposure to the issues would allow them to understand and take immediate corrective actions to remediate their software. This was not the researchers' experience in the case of SAFA. First, for three out of the four SAFA instances, some developers were aware of it. And for all the SAFA instances, they acknowledged that it was problematic. However, even after the researchers brought these issues to their attention, the developers still failed to implement the expected security fixes. Moreover, reactions after being exposed to the insecure code were not always consistent with subsequent behaviors, which suggested that there were other reasons why SAFA persisted in the code.

8.2.3 An Interpretation of Inconsistent Narratives

By correlating the documentation found about historical security issues and the insights that emerged from the analysis of R1's field notes, the following conjectures are made

66

about the motivations for introducing or not fixing the different security issues found by the researchers.

Conjecture 1. *Developers introduced SAFA 1 and 3 for usability and business reasons, expecting to hide errors and reduce the number of customer support tickets.*

One of the effects of SAFA was that it would make runtime errors unnoticeable by customers. Hiding runtime errors from customers reduced the chances that customers would complain that the AUS was not working correctly. When developers talked about these complaints, they implied that customers would blame the company if the AUS could not communicate with other servers, even if the problem was extraneous to the AUS. As one developer said:

> *"If the system breaks because we followed the specification and the system cannot talk to another server because they are not following the spec, we are probably going to lose money. So we need to code to prevent that."*

Hiding errors from customers would imply that fewer integration-support tickets would be generated, reducing the chances that some of those tickets would be escalated to the development team. Because developers often complained about how much time they spent debugging issues reported by customers (documented on integration-support tickets), possibly developers have introduced SAFA to alleviate their job.

Integration-support tickets were created by support techs assisting customers in integrating AUS with their other network products (e.g., routers and switches). Whenever support techs were unable to resolve tickets in this category they would escalate them to the development team for further investigation and potential development of bug fixes or custom integration code. Although developers understood that assisting support techs was necessary, they appeared to be more interested in developing new features than fixing bugs or writing custom integration code (which would later require more effort to be maintained).

Further, debugging the issues described on the tickets was challenging because it often required setting up environments that were similar to their customers', which was difficult because of the diversity of network device vendors. In this context, SAFA reduced the number of integration-support tickets, so developers would spend less time debugging integration problems and have more time to develop new features.

Conjecture 2. *Some security issues were not fixed because fixing them was not considered profitable for the business or the developers.*

Of the four instances of SAFA, the researchers were only assigned to fix two, SAFA 2 (misconfiguration, Section 7.5.3.2) and SAFA 4 (unimplemented SAML flow, Section 7.5.3.4). One of the possible reasons why SAFA 4 was fixed was that at the time the company was negotiating a deal with a customer that wanted the feature. Similarly, SAFA 2's fix was selected for development as part of a product merge plan that would allow the AUS to be sold to a significantly larger market, so managers likely saw the fix as a profitable improvement. Because there would be no profit gained for fixing the other instances, those tasks were backlogged. The XSS injections in the AUS were likely fixed because the owner of the project that was affected by those vulnerabilities was particularly interested in showing management that his product did not have any vulnerabilities. In contrast, other vulnerabilities were not fixed because no one was pushing for that to happen. For instance, recall that some developers were aware of SAFA 2 (misconfiguration, Section 7.5.3.2), acknowledged that it was problematic and that *"it shouldn't be there"*, yet they took no action to remediate the issue, possibly because they saw no benefit in making the extra effort.

Conjecture 3. *Developers maintained SAFA to avoid breaking existing implementations, even if it implied a security risk.*

Another reason why SAFA was still in production code was probably that developers didn't want to fix it to avoid introducing other issues with the fix. For instance, when asked

why SAFA 2 (misconfiguration, Section 7.5.3.2) was still there, the developers' explanation was simply that nobody took the time to remove it. In fact, removing SAFA implied a risk, i.e., some other part of the software could break on production systems, thus developers did not want to take this risk, especially because there was no need for it (because customers were not demanding it). In summary, there were likely just not enough incentives to remove it.

8.3   Reflections about the Intervention Model

The intervention approach used throughout the study evolved as new insights emerged from field observations. The initial assumptions were that (a) developers were not aware of any of the vulnerabilities discovered by researchers, and (b) as soon as developers learned about the newly discovered vulnerabilities, they would go ahead and fix the issues right away. In other words, researchers were operating a deficit-based intervention approach, in which the subjects were expected to take corrective actions upon being informed about their "deficits" in security.

The reactions of the developers after being exposed to the different security issues demonstrated that the assumptions made by the researchers did not always hold. The developers decided not to fix several security issues found by the researchers. Further, they described being aware of SAFA (in three out of four instances), but took no actions to remediate it. The development team consisted of highly skilled engineers that knew a fair bit about security, but their justifications for SAFA were at the least controversial from a security perspective. Evidently, there were other non-technical factors that were not initially considered by the researchers that motivated the developers' to ignore some of the issues. Some potential reasons that might have motivated developers to introduce or keep vulnerabilities in their code were outlined in the previous section. Still, a question that came up during research discussions was if researchers, or security experts in general, could

do anything better to help developers implement more effective solutions to their security problems.

In this particular study, a more empathetic understanding of developers was key to provide better interventions. Overcoming the deficit model in the researchers' own thinking helped them to better interpret why participants responded or not to security issues and to recognize how security concerns existed alongside other factors that shaped their work. This understanding motivated the researchers to try a co-creation model of secure development. Instead of focusing on deficits, the researchers collaborated with the developers on ideas and processes to improve software security together. This approach worked well, specially towards the end of the research, where developers became more open to implementing changes to improve their secure development process, and more security issues got fixed. For this model to be effective, the researchers had to first build rapport and gain the developers' trust. Part of that process meant that the researchers did not work exclusively on security but dealt with different tickets. These collaborations helped the developers see the researchers as proficient programmers that could work well within the development team.

Chapter 9: Conclusions

Computer security continues to be a pressing issue at least for the following two reasons: (1) implementations of runtime security mechanisms have not been rigorously analyzed and thus are not enforcing the policies that are expected to enforce, and (2) there are conflicting tensions in the software development process that hinder the implementation and maintainance of secure software. To investigate these problems, this dissertation provides two main contributions: a new general and unifying model of runtime enforcement that overcomes several limitations of previous works, and an ethnographic study on the software development process that aims to understand the various factors that hinder the implementation of effective security solutions. The implications of this work and some directions for future research are discussed next.

9.1   Stream-Monitoring Automata

Models of runtime monitors are fundamental to understand what policies can or cannot be ultimately enforced. Inconsistencies among specifications of security policies, models of monitors, and the actual properties being enforced by practical mechanisms can have critical consequences. Previous frameworks are not suitable for modeling monitors that operate over infinite event streams because they often omit important details or make assumptions that result in an imprecise characterization of enforceable properties.

The SMA model overcomes several limitations of previous works with respect to expressiveness and enables interesting analyses considering different enforcement scenarios. SMAs model practical abilities of security mechanisms monitoring streams that were not previously

considered, e.g. their ability to act even in the absence of input events and enforce some non-safety properties. SMAs adopt a simple enforcement model in terms of soundness, a widely accepted requirement across the security and formal methods communities. Interesting connections are drawn between runtime enforcement and game theory, and the exact set of enforceable properties is characterized by a new definition of game properties. The model's extensibility is demonstrated with adaptable meta-policies that embody extraneous enforcement constraints.

There are interesting open questions regarding extensions to the present work. For example, an analysis of which properties are enforceable by SMAs with memory constraints, compositional enforcement techniques, or quantitative evaluation of policies are all relevant topics with practical applications that may be developed in future work. Finally, SMAs provide a formal basis for analyzing many practical runtime enforcement mechanisms that are left out of previous models. Thus, the model may be used in the future for proving the correctness of monitoring algorithms that fit well into the SMA model.

9.2   Ethnographic Study of the Software Development Process

It has long been recognized that human factors play a dominant role in the ever-present software security issues. The ethnographic study presented in this dissertation aims to explain some of the misunderstood human factors that shape the development of secure software. The study spanned almost two years and studied secure software development processes using the method of participant observation. Two researchers were embedded as software developers in a company, where they participated in everyday work activities such as coding and meetings, and observed and provided intervention on software (in)security phenomena as it unfolded. They observed the developers' reactions to the discoveries of several security issues and their evolving attitudes towards them. The study found that (1) vulnerability discoveries produce different reactions in developers, often contrary to what a

security researcher would predict, and (2) security vulnerabilities are sometimes introduced and/or overlooked due to the difficulty in managing the various stakeholders' responsibilities in an economic ecosystem, and cannot be simply blamed on developers' lack of knowledge or skills.

It appears that the common wisdom, that software security problems need to be addressed by better education and training of developers, may not be targeting the core of the problem. This ethnographic study found that different factors may threaten developers' abilities to develop and maintain secure solutions, and that software (in)security does not happen solely because of deficits in the software and/or in developers' knowledge or efforts. The people in this company were trained professionals acting in good faith to create successful products. They faced dilemmas that can be common in the software industry and aimed to resolve them in ways that produced a viable product, established good relationships with customers, maintained profitability, and enhanced usability and security. Security issues were not directly attributable to deficits in knowledge, but rather emerged and sometimes persisted as a result of how developers dealt with both technical and human demands. The adverse patterns identified in this research are an attempt to understand the dynamics that drive developers away from implementing secure software effectively. Finally, the results of this study's interventions suggest that future attempts to improve software security might be more effective if they apply the lessons learned in this study, i.e. if they move away from focusing on developers' deficits towards a more understanding and fruitful model in which security is co-created between security experts and software developers.

References

[1] Hernan Palombo, Egor Dolzhenko, Jay Ligatti, and Hao Zheng. Stream-monitoring automata. In *Proceedings of the 2020 9th International Conference on Software and Computer Applications*, 2020.

[2] Patrick McDaniel and Stephen McLaughlin. Security and privacy challenges in the smart grid. *IEEE Security & Privacy*, 7(3), 2009.

[3] Sathish Alampalayam Kumar, Tyler Vealey, and Harshit Srivastava. Security in internet of things: Challenges, solutions and future directions. In *System Sciences (HICSS), 2016 49th Hawaii International Conference on*, pages 5772–5781. IEEE, 2016.

[4] Paul German. Face the facts–your organisation will be breached. *Network Security*, 2016(8):9–10, 2016.

[5] Serverless. The Serverless application framework powered by AWS Lambda, API Gateway, and more. https://serverless.com/, 2018.

[6] Shanmugavelayutham Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

[7] Serverless architecture market size, share. Industry report, 2018-2025. https://www.grandviewresearch.com/industry-analysis/serverless-architecture-market, 2018.

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

[9] 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.

[10] Philip WL Fong. Access control by tracking shallow execution history. In *Security and Privacy, 2004. Proceedings. 2004 IEEE Symposium on*, pages 43–55. IEEE, 2004.

[11] Chamseddine Talhi, Nadia Tawbi, and Mourad Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Information and Computation*, 206(2):158–184, 2008.

[12] Nataliia Bielova and Fabio Massacci. Do you really mean what you actually enforced? *International Journal of Information Security*, 10(4):239–254, 2011.

[13] David Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zălinescu. Enforceable security policies revisited. *ACM Transactions on Information and System Security (TISSEC)*, 16(1):3, 2013.

[14] Raphaël Khoury and Sylvain Hallé. Runtime enforcement with partial control. In *International Symposium on Foundations and Practice of Security*, pages 102–116. Springer, 2015.

[15] Minh Ngo, Fabio Massacci, Dimiter Milushev, and Frank Piessens. Runtime enforcement of security policies on black box reactive programs. In *ACM SIGPLAN Notices*, volume 50, pages 43–54. ACM, 2015.

[16] Egor Dolzhenko, Jay Ligatti, and Srikar Reddy. Modeling runtime enforcement with mandatory results automata. *International Journal of Information Security*, 14(1):47–60, 2015.

[17] Srinivas Pinisetty, Viorel Preoteasa, Stavros Tripakis, Thierry Jéron, Yliès Falcone, and Hervé Marchand. Predictive runtime enforcement. *Formal Methods in System Design*, 51(1):154–199, 2017.

[18] Yannis Mallios, Lujo Bauer, Dilsun Kaynar, and Jay Ligatti. Enforcing more with less: Formalizing target-aware run-time monitors. In *International Workshop on Security and Trust Management*, pages 17–32. Springer, 2012.

[19] Hala Assal and Sonia Chiasson. Think secure from the beginning: A survey with software developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–13, 2019.

[20] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L Mazurek, and Piotr Mardziel. Build it, break it, fix it: Contesting secure development. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 690–703, 2016.

[21] Daniel Votipka, Kelsey Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *Proceedings of the USENIX Security Symposium (USENIX SEC)*, August 2020.

[22] Glenn Wurster and Paul C Van Oorschot. The developer is the enemy. In *Proceedings of the 2008 New Security Paradigms Workshop*, pages 89–97, 2008.

[23] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[24] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 296–305, 2014.

[25] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391. IEEE, 2018.

[26] Christian Stransky, Yasemin Acar, Duc Cuong Nguyen, Dominik Wermke, Doowon Kim, Elissa M Redmiles, Michael Backes, Simson Garfinkel, Michelle L Mazurek, and Sascha Fahl. Lessons learned from using an online platform to conduct large-scale, online controlled security experiments with software developers. In *10th USENIX Workshop on Cyber Security Experimentation and Test (CSET 17)*, 2017.

[27] Ross Anderson. Why information security is hard-an economic perspective. In *Seventeenth Annual Computer Security Applications Conference*, pages 358–365. IEEE, 2001.

[28] Sathya Chandran Sundaramurthy, Alexandru G Bardas, Jacob Case, Xinming Ou, Michael Wesch, John McHugh, and S Raj Rajagopalan. A human capital model for mitigating security analyst burnout. In *Eleventh Symposium On Usable Privacy and Security (SOUPS 2015)*, pages 347–359, 2015.

[29] Sathya Chandran Sundaramurthy, John McHugh, Xinming Ou, Michael Wesch, Alexandru G Bardas, and S Raj Rajagopalan. Turning contradictions into innovations or: How we learned to stop whining and improve security operations. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, pages 237–251, 2016.

[30] Shreya Agrawal. Monitoring and enforcement of safety hyperproperties. Master's thesis, University of Waterloo, 2015.

[31] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. In *International Conference on Runtime Verification*, pages 190–207. Springer, 2017.

[32] Tzu-Chun Chen, Laura Bocchi, Pierre-Malo Deniélou, Kohei Honda, and Nobuko Yoshida. Asynchronous distributed monitoring for multiparty session enforcement. In *TGC*, volume 11, pages 25–45. Springer, 2011.

[33] Aaron Kane. Runtime monitoring for safety-critical embedded systems. 2015.

[34] David Basin, Felix Klaedtke, and Eugen Zalinescu. Failure-aware runtime verification of distributed systems. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 45. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[35] Hosein Nazarpour, Yliès Falcone, Mohamad Jaber, Saddek Bensalem, and Marius Bozga. Monitoring distributed component-based systems. *arXiv preprint arXiv:1705.05242*, 2017.

[36] Mahesh Viswanathan. Foundations for the run-time analysis of software systems. 2000.

[37] Moonjoo Kim, Sampath Kannan, Insup Lee, Oleg Sokolsky, and Mahesh Viswanathan. Computational analysis of run-time monitoring: Fundamentals of java-mac1. *Electronic notes in theoretical computer science*, 70(4):80–94, 2002.

[38] Shreya Agrawal and Borzoo Bonakdarpour. Runtime verification of k-safety hyperproperties in hyperltl. In *Computer Security Foundations Symposium (CSF), 2016 IEEE 29th*, pages 239–252. IEEE, 2016.

[39] Hugues Chabot, Raphaël Khoury, and Nadia Tawbi. Extending the enforcement power of truncation monitors using static analysis. *computers & security*, 30(4):194–207, 2011.

[40] Fatemeh Imanimehr and Mehran S Fallah. How powerful are run-time monitors with static information? *The Computer Journal*, 59(11):1623–1636, 2016.

[41] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Transactions on Information and System Security (TISSEC)*, 12(3):19, 2009.

[42] Richard Gay, Heiko Mantel, and Barbara Sprick. Service automata. In *International Workshop on Formal Aspects in Security and Trust*, pages 148–163. Springer, 2011.

[43] Srinivas Pinisetty and Stavros Tripakis. Compositional runtime enforcement. In *NASA Formal Methods Symposium*, pages 82–99. Springer, 2016.

[44] Nancy A Lynch and Mark R Tuttle. *An introduction to input/output automata*. Laboratory for Computer Science, Massachusetts Institute of Technology, 1988.

[45] Matthieu Renard, Yliès Falcone, Antoine Rollet, Srinivas Pinisetty, Thierry Jéron, and Hervé Marchand. Enforcement of (timed) properties with uncontrollable events. In *International Colloquium on Theoretical Aspects of Computing*, pages 542–560. Springer, 2015.

[46] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Nguena Timo. Runtime enforcement of timed properties revisited. *Formal Methods in System Design*, 45(3):381–422, 2014.

[47] Raphaël Khoury and Nadia Tawbi. Equivalence-preserving corrective enforcement of security properties. *International Journal of Information and Computer Security*, 7(2-4):113–139, 2015.

[48] Ylies Falcone, Jean-Claude Fernandez, and Laurent Mounier. Runtime verification of safety-progress properties. *RV*, 5779:40–59, 2009.

[49] Edward Chang, Zohar Manna, and Amir Pnueli. Characterization of temporal property classes. In *International Colloquium on Automata, Languages, and Programming*, pages 474–486. Springer, 1992.

[50] Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14(3):349–382, 2012.

[51] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[52] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *2010 IEEE Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.

[53] Aslan Askarov, Stephen Chong, and Heiko Mantel. Hybrid monitors for concurrent noninterference. In *2015 IEEE 28th Computer Security Foundations Symposium*, pages 137–151. IEEE, 2015.

[54] Gurvan Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. In *Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE*, pages 218–232. IEEE, 2007.

[55] Kevin W Hamlen, Greg Morrisett, and Fred B Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(1):175–205, 2006.

[56] George H Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[57] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[58] Bowen Alpern and Fred B Schneider. Defining liveness. *Information processing letters*, 21(4):181–185, 1985.

[59] David R Thomas. A general inductive approach for analyzing qualitative evaluation data. *American journal of evaluation*, 27(2):237–246, 2006.

[60] H Russell Bernard, Amber Wutich, and Gery W Ryan. *Analyzing qualitative data: Systematic approaches*. SAGE publications, 2016.

[61] OWASP top 10 - 2017 the ten most critical web application security risks, 2017. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_(en).pdf.

[62] Michael Howard and Steve Lipner. *The security development lifecycle*, volume 8. Microsoft Press Redmond, 2006.

[63] John Viega. The clasp application security process. *Secure Software*, 2005.

Appendix A: Copyright Permissions

The copyright permission for the content used in Chapters 1, 3, and 4 is shown below.

79

Appendix B: Institutional Review Board Authorization

The IRB approval for the ethnographic study is shown below.

USF UNIVERSITY OF SOUTH FLORIDA

RESEARCH INTEGRITY AND COMPLIANCE
Institutional Review Boards, FWA No. 00001669
12901 Bruce B. Downs Blvd., MDC035 • Tampa, FL 33612-4799
(813) 974-5638 • FAX (813) 974-7091

6/26/2018

Daniel Lende, Ph.D.
Anthropology
Department of Anthropology
4202 E. Fowler Avenue, SOC 107
Tampa, FL 33620

RE: **Expedited Approval for Initial Review**
IRB#: Pro00036117
Title: SaTC: CORE: Medium: Collaborative: Understanding Security in the Software
Development Lifecycle: A Holistic, Mixed-Methods Approach

Study Approval Period: 6/26/2018 to 6/26/2019

Dear Dr. Lende:

On 6/26/2018, the Institutional Review Board (IRB) reviewed and **APPROVED** the above
application and all documents contained within, including those outlined below.

Approved Item(s):
Protocol Document(s):
IRB Protocol Version 1 June 2018.docx

Consent/Assent Document(s)*:
Verbal Consent Form.docx*

*Please use only the official IRB stamped informed consent/assent document(s) found under the
"Attachments" tab. Please note, these consent/assent documents are valid until the consent
document is amended and approved. * Verbal consent does not get stamped.

It was the determination of the IRB that your study qualified for expedited review which
includes activities that (1) present no more than minimal risk to human subjects, and (2) involve
only procedures listed in one or more of the categories outlined below. The IRB may review
research through the expedited review procedure authorized by 45CFR46.110 and 21 CFR
56.110. The research proposed in this study is categorized under the following expedited review

category:

(7) Research on individual or group characteristics or behavior (including, but not limited to, research on perception, cognition, motivation, identity, language, communication, cultural beliefs or practices, and social behavior) or research employing survey, interview, oral history, focus group, program evaluation, human factors evaluation, or quality assurance methodologies.

Your study qualifies for a waiver of the requirements for the documentation of informed consent as outlined in the federal regulations at 45CFR46.117(c) which states that an IRB may waive the requirement for the investigator to obtain a signed consent form for some or all subjects if it finds either: (1) That the only record linking the subject and the research would be the consent document and the principal risk would be potential harm resulting from a breach of confidentiality. Each subject will be asked whether the subject wants documentation linking the subject with the research, and the subject's wishes will govern; or (2) That the research presents no more than minimal risk of harm to subjects and involves no procedures for which written consent is normally required outside of the research context. For the use of the verbal consent only.

As the principal investigator of this study, it is your responsibility to conduct this study in accordance with IRB policies and procedures and as approved by the IRB. Any changes to the approved research must be submitted to the IRB for review and approval via an amendment. Additionally, all unanticipated problems must be reported to the USF IRB within five (5) business days.

We appreciate your dedication to the ethical conduct of human subject research at the University of South Florida and your continued commitment to human research protections. If you have any questions regarding this matter, please call 813-974-5638.

Sincerely,

Kristen Salomon, Ph.D., Chairperson
USF Institutional Review Board